8051 PROGRAMMING (CPE575)

1.0 Computer Overview

Definitions

A computer is defined by:

1. the ability to be programmed to operate on data without human intervention.
2. the ability to store and retrieve data.

Generally, it includes the peripheral devices for communicating with the outside world as well as programs that process data. The equipment is hardware while the program is software. Figure 1.1 is the block diagram of a typical computer.

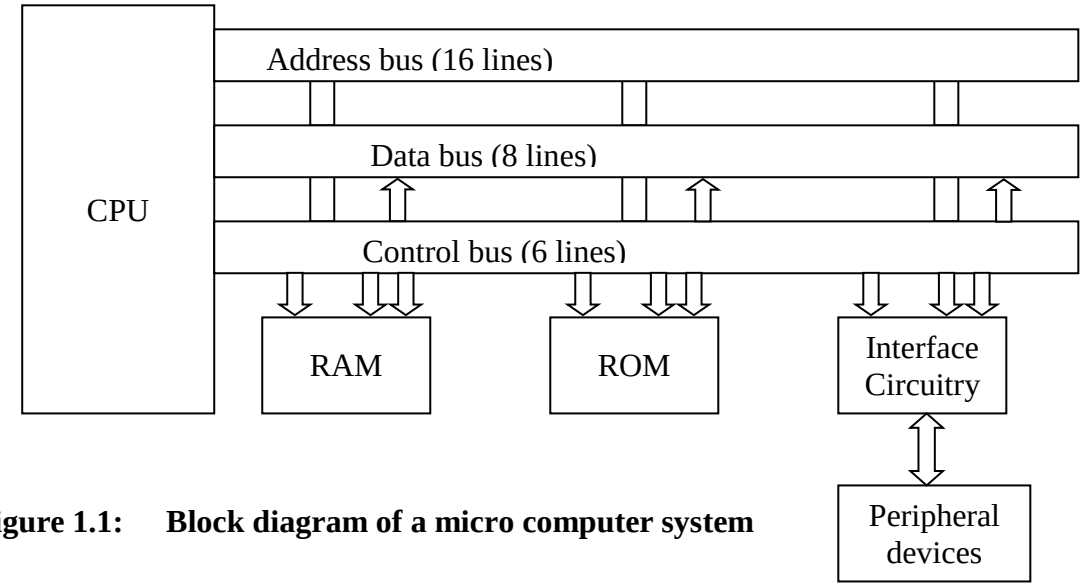


Figure 1.1: Block diagram of a micro computer system

The block diagram of Figure 1.1 is representative of all sizes of computers. It contains a Central Processing Unit (CPU) connected to random access memory (RAM) and read only memory (ROM) via the address bus, data bus and control bus. Interface circuits connect the system buses to peripheral devices.

1.1 THE CENTRAL PROCESSING UNIT

The CPU administers all activity in the system and performs all operations on data. The CPU consists of logic circuits that continuously perform two operations.

1. Fetching instructions and
2. Executing instructions

The CPU has the ability to understand and execute instructions based on a set of binary codes, each representing a simple operation. These instructions are usually:

1. Arithmetic (add, subtract, multiply, divide)

2. Logical (AND, OR, NOT, XOR, etc)
3. Data movement
4. Branch operations

All these instructions are represented by a set of binary codes called instruction set.

Figure 1.2 presents a typical view of a simplified internal structure of a CPU.

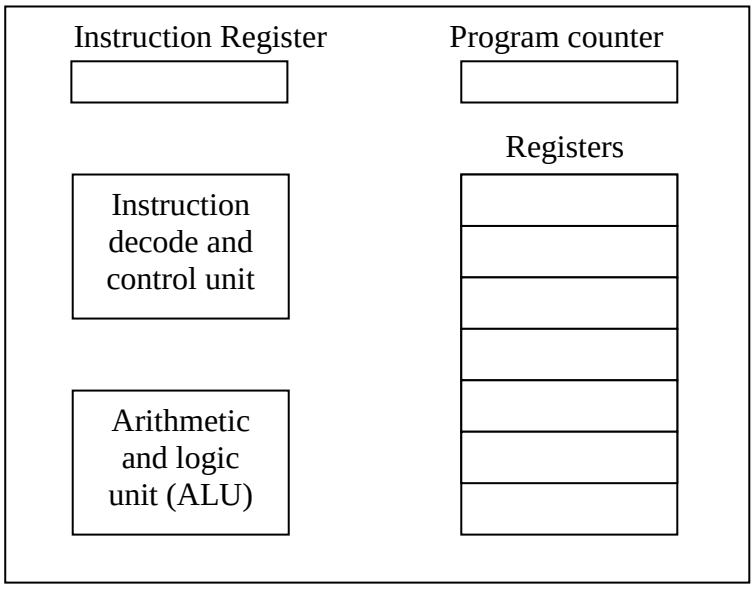


Figure 2: The Central Processing Unit (CPU)

The CPU consists of

1. **Registers –** for temporary storage of information
2. **ALU –** for performing operations on the information
3. **Instruction decode and control unit –** that determines the operation to perform and sets in motion the necessary actions to perform it.
4. **The instruction register –** that holds the binary code for each instruction as it is executed and
5. **The program counter (PC) –** that holds the memory address of the next instruction to be executed.

1.1.1 Fetching Instructions

Fetching instructions involves the following steps:

1. the contents of the program counter are placed on the address bus.
2. a READ control signal is activated.
3. data (the instruction opcode) are read from RAM (or memory) and placed on the data bus.
4. the opcode is latched into the CPU's internal instruction register and

5. the program counter is incremented to prepare for the next fetch from memory.

Figure 1.3 illustrates the flow of information for an instruction fetch.

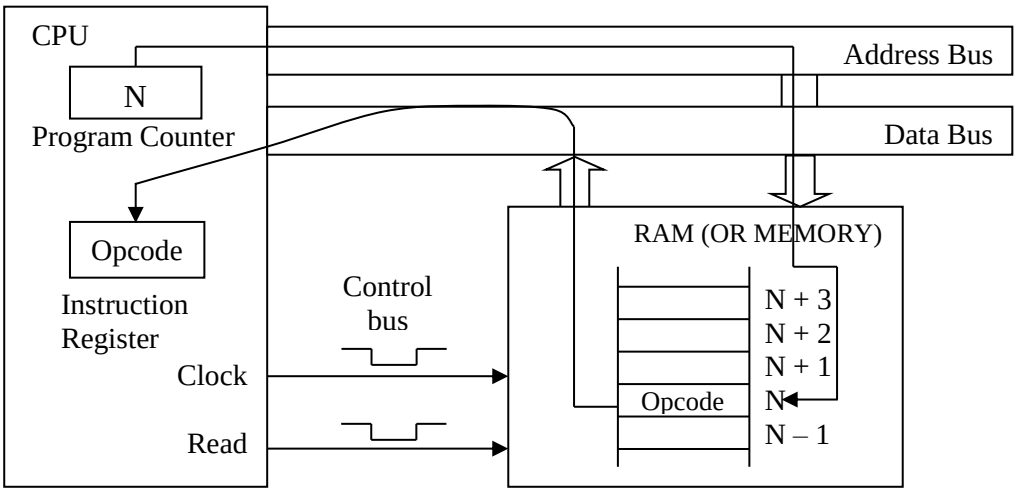


Figure 3:Bus activity for an opcode fetch cycle

1.1.2 Executing Instructions

Instruction execution involves decoding (or deciphering) the opcode and generating control signals to gate internal registers in and out of the ALU and to signal the ALU to perform the specified operation. This step may require more steps such as reading a second and third byte as data for the operation.

1.1.3 Program

A series of instructions combined to perform a meaningful task is called a ‘program’ or software. The degree to which tasks are efficiently and correctly carried out is determined mostly by the quality of software and not by the sophistication of the CPU.

1.1.4 Semiconductor Memory: RAM and ROM

Programs and data are stored in memory. The memory devices directly accessible by the CPU consists of semiconductor ICs called RAM and ROM.

There are features that differentiate RAM and ROM.

1. RAM is read/write memory while ROM is read only memory.
2. RAM is volatile while ROM is non-volatile

Most computer systems have a disk drive and a small amount of ROM, just enough to hold the short, frequently used software routines that perform input/output operations. User programs and data are stored on disk and are loaded into RAM for execution.

1.2 THE BUSES: ADDRESS, DATA AND CONTROL

A bus is a collection of wires carrying information with a common purpose. Access to the circuitry around the CPU is provided by three buses:

1. The Address Bus
2. The Data Bus and
3. The Control Bus

For each read or write operation, the CPU specifies the location of the data (or instruction) by placing an address on the address bus and then activates a signal on the control bus, indicating whether the operation is a read or write.

Read operations retrieve a byte of data from memory at the location specified and place it on the data bus. The CPU reads the data and places it in one of its internal registers.

For a write operation, the CPU outputs data on the data bus. Because of the control signal, memory recognizes the operation as a write cycle and stores the data in the location specified.

Most small computers have 16 or 20 address lines. Given n address lines each with a possibility of being high (1) or low (0) 2^n locations can be accessed. For n = 16, we have 2^{16} = 65536 and since 2^{10} = 1024 = 1K, 16 bits can address 2^6 x 2^{10} = 64k locations and 20 bits can address 1024K, i.e. 2^{10} x 2^{10} = 1,048,576 = 1M.

The data bus carries information between the CPU and memory or between the CPU and I/O devices. Research has shown that computers spend $^2/_3$ of their time just moving data. Since the majority of the move operations are between a CPU register and external RAM or ROM, the number of lines (the width) of the data bus is important for overall performance. Hence, computers are sometimes classified based on the size of the data bus e.g. 4-bit, 8-bit, 16-bit etc.

Control bus is a collection of signals, each having a specific role in the orderly control of system activity. Control signals are timing signals supplied by the CPU to synchronize the movement of information on the address and data buses. Although, there are usually three signals such as Clock, Read and Write, for data movement between the CPU and memory, the names and operations of these signals are highly dependent on the specific CPU.

1.3 Input/Output Devices

I/O devices or computer peripherals provide the path for communication between the computer system and the real world. Three classes of I/O devices are:

1. Mass Storage

These are also built on memory technology and hold large quantities of information (programs or data) that can not fit into the computer's relatively small RAM or 'main memory'. This information must be loaded into main memory before the CPU accesses it. Mass storage devices

are either ‘online’ or ‘archival’. Online storage, usually on magnetic disk, is available to the CPU without human intervention upon request of a program while archival storage holds data that are rarely needed and require manual loading onto the system. Archival storage used to be held on magnetic tapes or disks but now optical disks, such as CD-ROM technology are used.

2. Human Interface Devices

These devices require human intervention for their use. The most common of these devices are video display unit/terminal (VDT) and printer. While printers are strictly output devices that generate hardcopy output, VDTs are two devices: they contain a keyboard for input and a CRT (cathode ray tube) for output. Other human interface devices include the joystick, light pen, mouse, microphone and loudspeaker.

3. Control/monitor Devices

These devices help computers to perform a variety of control-oriented tasks unceasingly without fatigue far beyond human capabilities. Applications such as temperature control of a building, home security, elevator control, home appliance control and even welding parts of an automobile are all made possible by using these devices.

Control devices are outputs or actuators, that can affect the world around them when supplied with a voltage or current (e.g. motors and relays). Monitoring devices are inputs or sensors, which are stimulated by heat, light, pressure, motion, etc and convert this energy to a voltage or current read by the computer (e.g. photo transistor. thermistors and switches). The interface circuitry converts the voltage or current to binary data or vice versa, and through software an orderly relationship between inputs and outputs is established.

These three groups of input/output devices are the most frequently used with microprocessors and microcontrollers.

1.4 Programs: Big and Small

The early days of computing witnessed the materials, manufacturing and maintenance costs of computer hardware far surpassing the software costs. Today, however, with mass produced large-scale integrated (LSI) chips, hardware costs are less dominant. It is the labor-intensive job writing, documenting, maintaining, updating and distributing software that constitutes the bulk of the expense in automating a process using computers.

There are different types of software. Figure 1.4 illustrates three levels of software between the user and the hardware of a computer system.

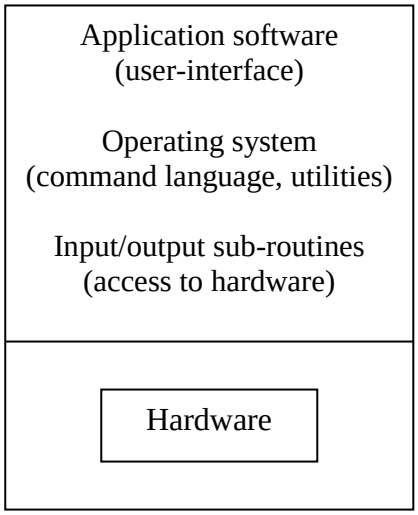


Figure 1.4: Levels of software

The three levels of software between the user and the hardware of a computer system are:

1. The application software
2. The operating system
3. The input/output subroutines

At the lowest level, the input/output subroutines directly manipulate the hardware of the system, reading characters from the keyboard, writing characters to the CRT, reading blocks of information from the disk, and so on. Since these subroutines are so intimately linked to the hardware, they are written by the hardware designers and are usually stored in ROM. They are the BIOS – basic input/output system on the desktop system, for example.

To provide close access to the system hardware for programmers, explicit entry and exit conditions are defined for the input/output subroutines. One only needs to initialize value in CPU registers and call the subroutine. The action is carried out with results returned in CPU registers or left in system RAM. Apart from the input/output subroutines, the ROM contains a start-up program that executes when the system is powered up or is reset manually by the operator. ROM is essential here since this program must exist upon power-up.

'House keeping' chores such as checking for options, initializing memory, performing diagnostic checks, etc, are all performed by the start-up program.

A bookshop loader routine reads the first tract (a small program) from the disk into RAM and passes control to it. This program then loads the RAM-resident portion of the operating system (a large program) from the disk and passes control to it; thus completing the start-up of the system.

The operating system is a large collection of programs that come with the computer system and provides the mechanism to access, manage, and effectively utilize the computer's resources. These abilities exist through the operating system's command language and utility programs which in turn facilitate the development of applications software. If the applications software is well designed, the user interacts with the computer with little or no knowledge of the operating system. Providing an effective, meaningful, and safe user interface is a prime objective in the design of applications software.

1.5 Micros, Minis and Mainframes

Using their size and power as a starting point, computers are classified as microcomputers, minicomputers or mainframe computers.

The key property of micro computers is the size and packaging of the CPU. It is contained within a single integrated circuit, the Microprocessor. Mini computers and mainframe computers, apart from being more complex in every architectural detail, have CPUs consisting of multiple ICs, ranging from several ICs (minicomputers) to several circuit boards of ICs (mainframes). This increased capacity is necessary to achieve the high speeds and computational power of larger computers.

Another feature separating micros from mini and mainframes is that microcomputers are single-user, single-task system, i.e., they interact with one user and they execute one program at a time. Minis and mainframes are multiuser, multitasking systems, i.e., they can accommodate many users and programs simultaneously. Simultaneous executing of programs is actually an illusion resulting from "time slicing" CPU resources. Multiprocessing systems, however, use multiple CPUs to execute tasks simultaneously.

1.5.1 Microprocessors Vs Microcontrollers

The distinguishing features between microprocessors and microcontrollers can be addressed from three perspectives.

1. Hardware architecture

Figure1.5 shows a more detailed diagram of internal view of the microcomputer system

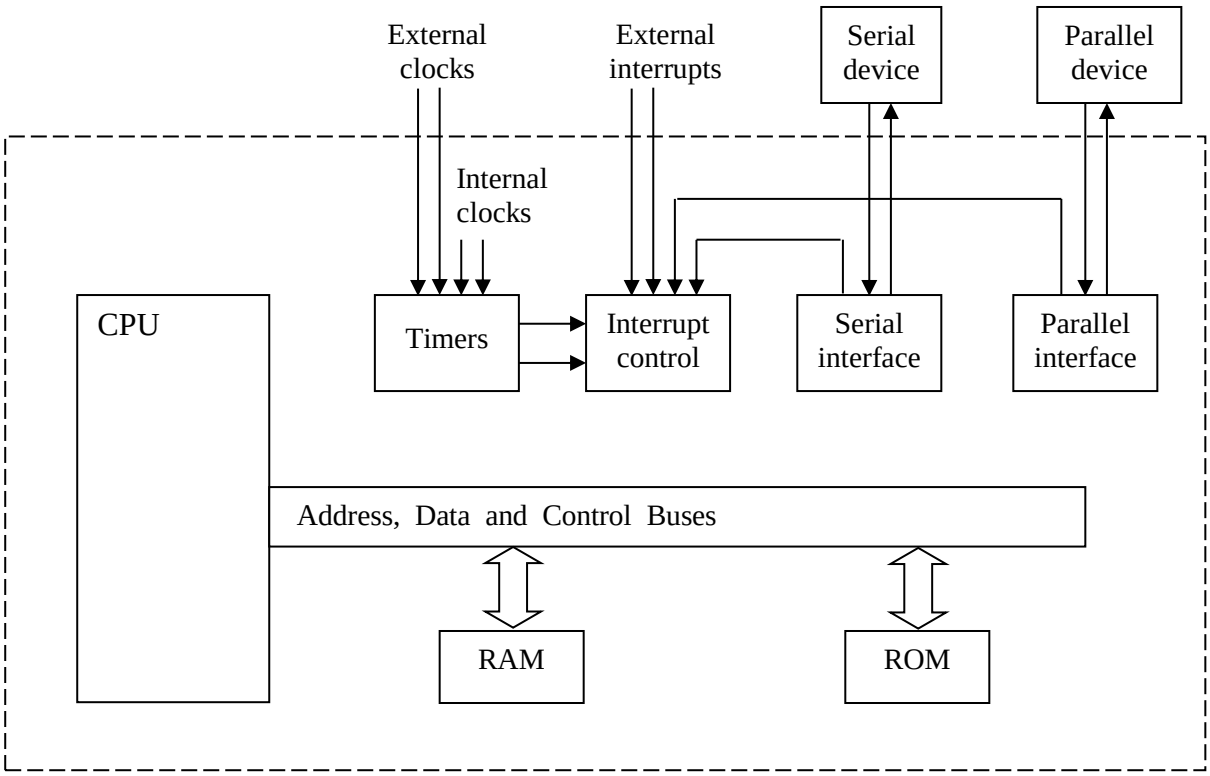


Figure 1.5: Detailed block diagram of a microcomputer system

The microprocessor is a single-chip CPUM while a microcontroller contains in a single IC a CPU and much of the remaining circuitry of a complete microcomputer system. Microcontrollers also include RAM, ROM, serial interface, a parallel interface, timer, interrupt scheduling circuitry, all within the same IC.

An important feature of microcontrollers is the built-in interrupt systems. As control-oriented devices, they are often called upon to respond to external stimuli (interrupts) in real time. They must perform fast context switching, suspending pre process while executing another in response to an 'event'. The opening of a microwave oven's door is an example of an event that might cause an interrupt in a microcontroller-based product.

Whereas a microprocessor requires external components to implement an interrupt scheme, a microcontroller's on-chip circuitry includes all the interrupt handling circuitry necessary.

2. Applications

Microprocessors are mostly used as the CPU in microcomputer systems. This function is what they are designed for and this is where their strength lies. Microcontrollers however, are found in small, minimum component design performing control-oriented activities. A microcontroller can aid in reducing overall component count in a circuit that would normally require dozens or hundreds of digital ICs. A microcontroller only requires a small number of support components and a control program in ROM. Microcontrollers are suited to control of I/O device, in design

requiring a minimum component count whereas microprocessors are suited to "processing" information in computer systems.

3. Instruction set features

Microprocessor instruction sets are processing intensive implying that they have powerful addressing modes with instruction catering for operations on large volumes of data. Their instructions operate on nibbles, bytes, words or even double words. Addressing modes provide access to large arrays of data, using address pointers and offsets. Auto-increment and auto-decrement modes simplify stepping through arrays on bytes, word or double-word boundaries. The bit goes on.

Microcontrollers, on the other hand, have instruction sets catering for the control of inputs and outputs. The interface to many inputs and outputs uses a single bit. For example, a motor may be turned on and off by a solenoid energized by a 1-bit output port. Microcontrollers have instructions to set and clear individual bits and perform other bit-oriented operations such as logically ANDing, ORing, or EXORing bits, jumping if a bit is set or clear and so on. This feature is rarely present in microprocessors, which, are usually designed to operate on bytes or larger units of data.

1.5.2 I/O PORT STRUCTURE

The internal circuitry for the port pins is as shown in Figure 1.6.

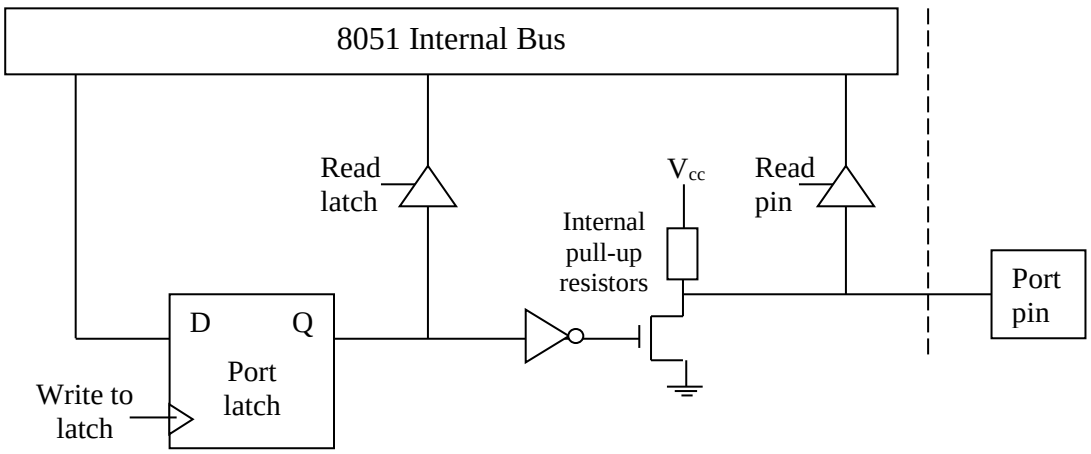


Figure 1.6: Circuitry for I/O ports

Writing to a port pin loads data into a port latch that drives a field-effect transistor connected to the port pin. The drive capability is four low-power schottky TTL loads for ports 1, 2 and 3, and eight loads for port 0. The pull up resistor is absent on port 0 when functioning as the

external address/data bus. An external pull-up resistor may be needed depending on the input characteristics of the device driven by the port pin.

There is both a 'read latch' and 'read pin' capability. Instructions that require a read-modify operation (e.g. CPL P1.5) read the latch to avoid misinterpreting the voltage level in the event. The pin is heavily loaded (e.g. when driving the base of a transistor). Instructions that input a port bit (e.g. MOV C, P1.5) read the pin. The port latch must contain a 1, in this case, otherwise the FET driver is ON and pulls the output low. A system reset sets all port latches, so port pins may be used as inputs without explicitly setting the port latches. If however, a port latch is cleared (e.g. CLR P1.5) then it cannot function subsequently as an input unless the latch is set first (e.g. SETB P1.5).

When the alternate functions of port 0, 2 and 3 (not shown in the diagram for I/O circuitry, Figure 1.6) is in effect, the output drivers are switched to an internal address (port 2), address/data (port 0), or control (port 3) signal as appropriate.

1.5.3 TIMING AND THE MACHINE CYCLE

The 8051's on-chip oscillator is driven by an external quartz crystal through pins 18 and 19. This crystal has a typical frequency 12MHz. These oscillator clock cycles form the basis of the 8051's timing and synchronization.

With the oscillator as reference, the 8051 requires two clock cycles to perform a single discrete operation, which is either fetching an instruction, decoding, or executing it. This duration of two clock cycles is also called a state. Therefore, in order to fully process an instruction, the 8051 would generally require 6 such states, or 12 clock cycles since it would have to first fetch and decode the instruction before it goes to execute it. This duration of six states is also known as one 'machine cycle'. More complex instructions would take more than one machine cycle to be carried out. This number ranges from one to four machine cycles. The Figure 1.7 shows the relationship between clock cycles (P), states (S) and a machine cycle.

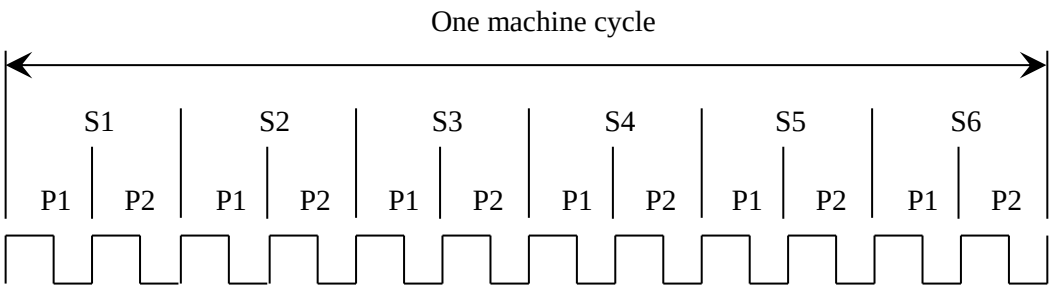


Figure 1.7: Relationship between oscillator clock cycles, states and machine cycle

Typically, the 8051's on-chip oscillator, f_{osl} is driven by a 12MHz crystal, so the period of one clock pulse,

$$T_{clock} = \frac{1}{f_{osl}} = \frac{1}{12MHz} = 83.33ns$$

One machine cycle consists of 12 such clock pulses, hence its duration is 83.33ns x 12 = 1ms

1.5.4 MEMORY ORGANIZATION

Most microprocessors implement a shared memory space for data and programs. Programs are usually stored on disk and loaded into RAM for execution. Thus, both data and programs reside in the system RAM. Microcontrollers are rarely used as the CPU in computer systems. Instead they are employed as the central component in control-oriented designs. There is limited memory and there is no drive or disk operating system. The control program must reside in ROM.

The 8051 implements a separate memory space for programs (code) and data. The internal memory consists of on-chip ROM (8051/8052) and on-chip data RAM. The on-chip RAM contains a rich management of general purpose storage, bit-addressable storage, register banks and special function registers.

Two notable features are:

1. The registers and input/output ports are memory mapped and accessible like any other memory location.
2. The stack resides within the internal RAM, rather than in external RAM as typical of microprocessors.

Any location in the general purpose RAM can be accessed freely using the direct or indirect addressing modes. MCS-51 is a family of which 8051 was the first commercially provided member. Table 1.1 compares the family.

Table1.1: MCS-51 Family

Port Number	On-chip Code Memory	On-chip Data Memory	Timers
8051	4K ROM	128 bytes	2
8031	0K	128 bytes	2
8751	4K EPROM	128 bytes	2
8052	8K ROM	256 bytes	2
8032	0K	256 bytes	2
8752	8K EPROM	256 bytes	2

The term 8051 refers to the MCS-51 family of microcontrollers.

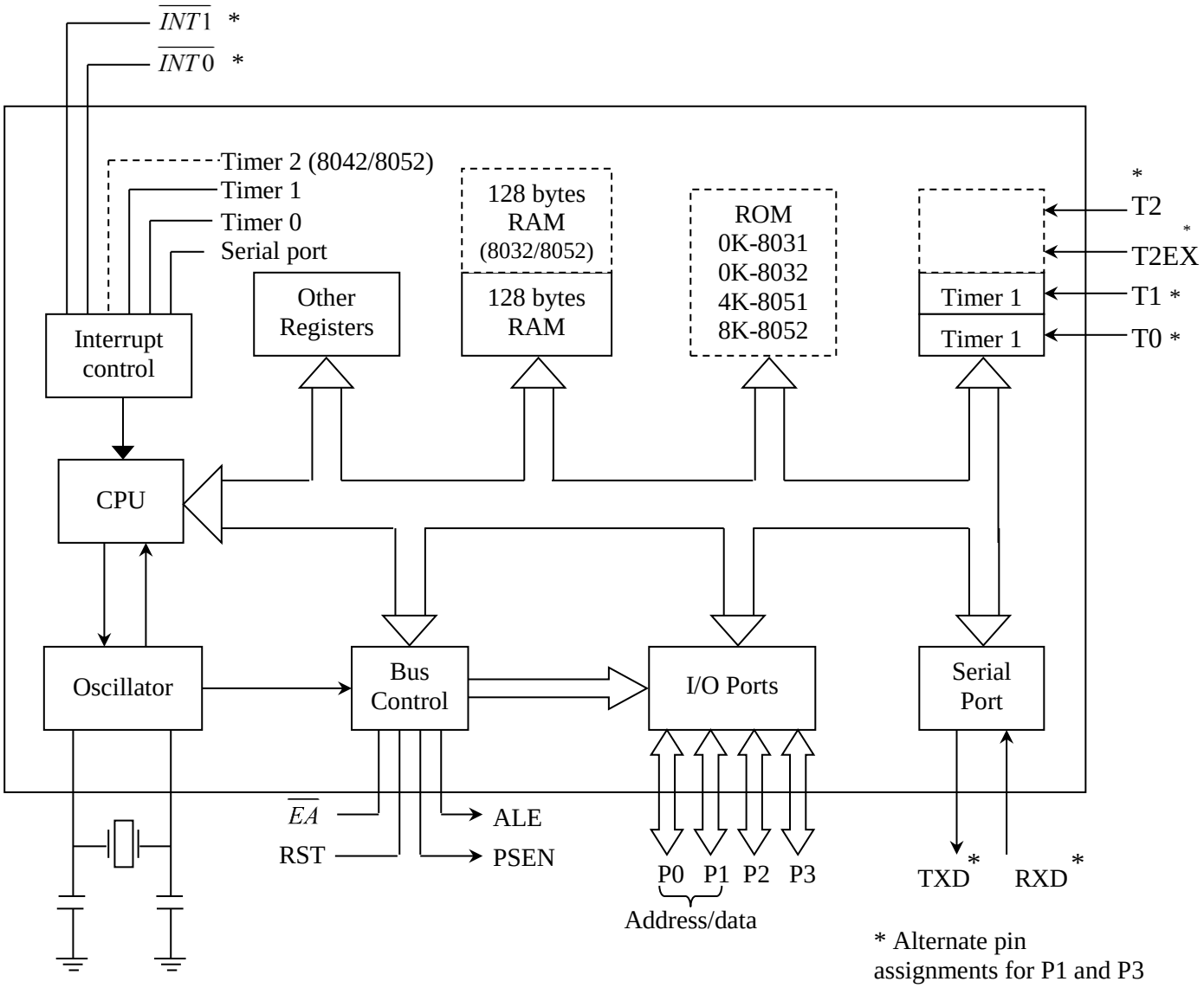


Figure 1. 8: 8051 MCS-51 Block Diagram

The figure 8 and Figure 9 show the 8051 hardware architecture from an external perspective, i.e., the pinouts.

The 8051 has 40 pins out of which 32 function as I/O port lines. However 24 of these are dual purpose (26 on the 8032/8052). Each can operate as I/O or as a control line or part of the address or data bus.

Designs requiring a minimum of external memory or other external components use these ports for general purpose I/O. The eight lines in each port can be treated as a unit in interfacing to parallel devices such as printer, digital-to-analog converters and so on. Each line can also operate independently in interfacing to single-bit devices such as switches, CEDS, transistors, solenoids, motors and loudspeakers.

(I) Port 0

Port 0 is a dual purpose port on pins 32 – 39 of the 8057 IC. In minimum component design, it is used as a general purpose I/O port. For larger designs with external memory, it becomes a multiplexed address and data bus.

(II) Port 1

Port 1 is a dedicated I/O port on pins 1- 8. the pin designated as P1.0, P1.1, P1.2, etc, are available for interfacing to external devices as required. No alternate functions are assigned for port 1 pins. They are used solely for interfacing to external devices.

(III) Port 2

Port 2 (pins) 21-28) is a dual-purpose port serving by general purpose I/O, or as the high byte of the address bus for design with external code memory or more than 256 bytes of external data memory.

(IV) Port 3

Port 3 is a dual-purpose port on pins 10-17. Apart of being used as general-purpose I/O, these pins are multifunctional with each having an alternate purpose related to special features of the 8051. The alternate purpose of the port 3 and port 1 pin is summarized in Table 1.2.

Table 1.2: Alternate pin functions for port pins

	Bit	Name	Bit address	Alternate function
	P3.0	RXD	BOH	Receive data for serial port
	P3.1	TXD	B1H	Transmit data for serial port
	P3.2	$\overline{INT0}$	B2H	External interrupt 0
	P3.3	$\overline{INT1}$	B3H	External interrupt 1
	P3.4	TO	B4H	Timer/counter 0 external input
	P3.5	T1	B5H	Timer/counter 1 external input
	P3.6	$\overline{WR}$	B6H	External data memory write strobe
	P3.7	$\overline{RD}$	B7H	External data memory read strobe
8032/8052 only	P1.0	T2	90H	Timer/counter 2 external input
8032/8052 only	P1.1	T2EX	91H	Timer/counter 2 capture/reload

(V) $\overline{PSEN}$ (Program store enable)

The 8051 has four dedicated bus control signals. Program store enable ($\overline{PSEN}$) in an output signal on pin 29. it is a control signal that enables external program (code) memory. It usually connects to an EPROM Output Enable ($\overline{OE}$) pin to permit reading of program bytes. The $\overline{PSEN}$ signal pulses low during the fifth stage of an instruction which is stored in external

program memory. The binary code of a program (OPCode) are read from EPROM, travel across the data bus, and are catched into the 8051s instruction register for decoding. When executing a program from internal ROM, $\overline{PSEN}$ remains in the inactive (high) state.

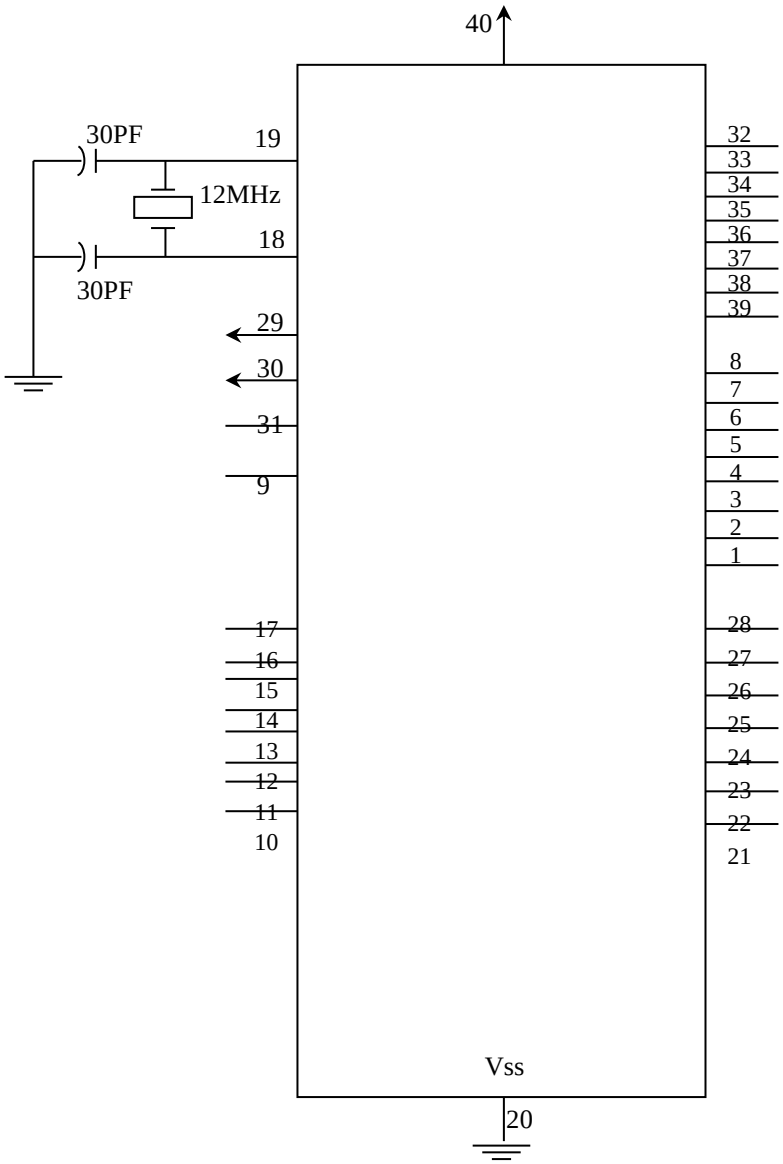


Figure 1.9: 8051 Pin Assignments

The ALE output signal on pin 30 is used by the 8051 for demultiplexing the address and data bus. When port O is used in alternate mode, on the data bus and low byte of the address bus, ALE in the signal that catches the address into an external register during the first half of a memory cycle. This done, the port lines are then available for data input or output during the second half of the memory cycle, when the data transfer takes place.

The ALE signal pulses at $^1/_6{}^{th}$ the on-chip oscillator frequency can be used on a general purpose clock for the rest of the system. If the 8051 is clocked from a 12MHz crystal, the ALE signal oscillates at 2MHz. the only exception is during the MOVx instruction, when one ALE pulse is missed. This pin is also used for the programme input pulse for EPRom versions of 8051.

(VI) $\overline{EA}$ **(External Access)**

The $\overline{EA}$ input signal on pin 31 in generally tied high (+5V) or low (ground). If high, the 8051/8052 executes programs from internal ROM when executing in the lower 4K/8K of memory. If low, programs execute from external memory only (and $\overline{PSEN}$ pulses low accordingly).

$\overline{EA}$ must be tied low for 8031/8032 ICs, since there is no on-chip program memory. If $\overline{EA}$ is tied low on an 8051/8052, internal ROM is disabled and programs execute from external EPROM. The EPROM versions of the 8051 also use the $\overline{EA}$ line for the +21 volt supply (Vpp) for programming the internal EPRom.

(VII) RST (Reset)

The RST input on pin 9 is the master reset for the 8051. When the signal is brought high for at least two machine cycles, the 8051 internal registers are loaded with appropriate values for an orderly system start-up. For normal operation, RST is low.

1.5.5 ON-CHIP OSCILLATOR INPUTS

The 8051 features an on-chip oscillator that is typically driven by a crystal connected to pins 18 and 19. Stabilizing capacitors are also required as shown in the sketch for the 8051 pin outs.

The nominal crystal frequency is 12MHz for most ICs in the MCS-51 family although the 80C 51 BH-1 can operate with crystal frequencies or up to 16MHz. the on-chip oscillator need not be driven by a crystal. As shown in Figure1. 10 a TTL clock source can be connected to XTAL1and XTAL2.

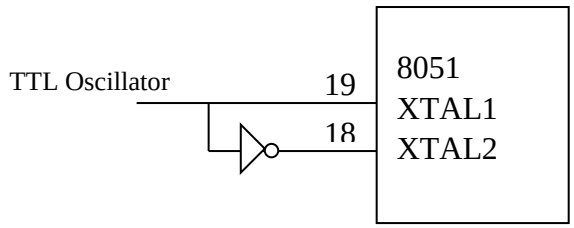


Figure 1.10: Driving the 8051 from a TTL Oscillator

1.5.6 POWER CONNECTIONS

The 8051 operates from a single +5 volt supply the Vcc connection is on pin 40, and Vss (ground).

1.6 8051 Memory

The 8051 has three very general types of memory. To effectively program the 8051 it is necessary to have a basic understanding of these memory types. The memory types are as shown in Figure 1. 11 and include On-Chip Memory, External Code Memory, and External RAM.

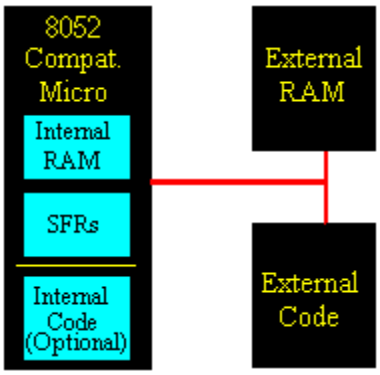


Figure 1.11: 8051 Memory Types

(I) On-Chip Memory

This refers to any memory (Code, RAM, or other) that physically exists on the microcontroller itself. On-chip memory can be of several types, but we'll get into that shortly.

(II) External Code Memory

This is code (or program) memory that resides off-chip. This is often in the form of an external EPROM.

(III) External RAM

This is RAM memory that resides off-chip. This is often in the form of standard static RAM or flash RAM.

1.6.1 Code Memory

Code memory is the memory that holds the actual 8051 program that is to be run. This memory is limited to 64K and comes in many shapes and sizes: Code memory may be found on-chip, either burned into the microcontroller as ROM or EPROM. Code may also be stored completely off-chip in an external ROM or, more commonly, an external EPROM. Flash RAM is also another popular method of storing a program. Various combinations of these memory types may also be used--that is to say, it is possible to have 4K of code memory on-chip and 64k of code memory off-chip in an EPROM.

When the program is stored on-chip the 64K maximum is often reduced to 4k, 8k, or 16k. This varies depending on the version of the chip that is being used. Each version offers specific capabilities and one of the distinguishing factors from chip to chip is how much ROM/EPROM

space the chip has. However, code memory is most commonly implemented as off-chip EPROM. This is especially true in low-cost development systems and in systems developed by students.

1.6.2 External RAM

The 8051 also supports External RAM. Which is any random access memory which is found off-chip. Since the memory is off-chip it is not as flexible in terms of accessing, and is also slower. For example, to increment an Internal RAM location by 1 requires only 1 instruction and 1 instruction cycle. To increment a 1-byte value stored in External RAM requires 4 instructions and 7 instruction cycles, implying in this case ,that external memory is 7 times slower.

However while Internal RAM is limited to 128 bytes (256 bytes with an 8052), the 8051 supports External RAM up to 64K.

1.6.3 On-Chip Memory

On-chip memory is one of two types: Internal RAM and Special Function Register (SFR) memory.

CHAPTER TWO

8051 MEMORY AND PROGRAMMING MODEL

2.1 8051 INTERNAL RAM STRUCTURE

The layout of the 8051's internal memory is presented in the memory map of Figure 2.1 and Figure 2.2. The internal data memory space is divided between internal RAM (00H - 7FH) and special function registers (80H - FFH). The internal data memory has the range from 00H - FFH (256 bytes), out of which 00H - 7FH is for general data while 80H - FFH is mostly for specific purposes and not for general data. Hence, 00H - 7FH is considered to be internal RAM.

The internal RAM is further subdivided into register banks (00H - 1FH), bit addressable RAM (20H - 2FH), and general-purpose RAM (30H - 7FH).

General purpose RAM is from addresses 30H to 7FH. Locations 00H – 2FH can similarly be used but they also have other purposes.

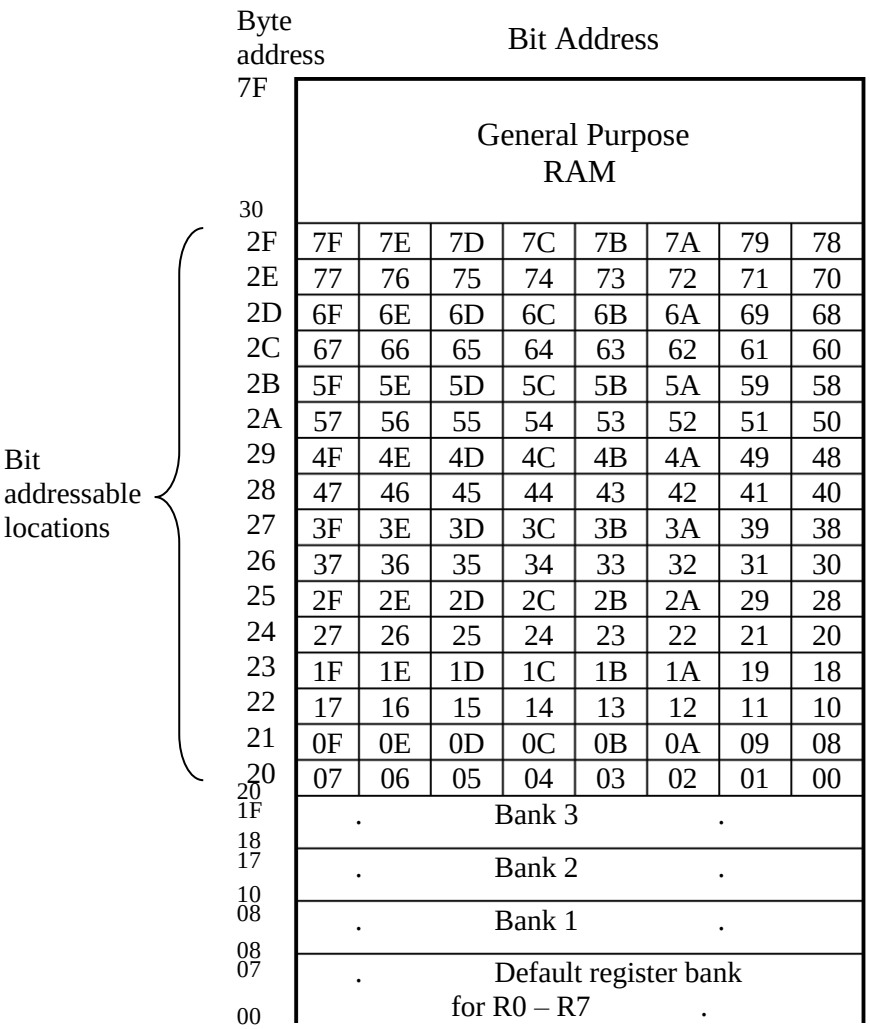


Figure 2.1: Internal 128 RAM structure of the 8051.

Byte Address	Bit Address								
FF	.								
F0	F7	F6	F5	F4	F3	F2	F1	F0	B
	.								
E0	E7	E6	E5	E4	E3	E2	E1	E0	ACC
	.								
D0	D7	D6	D5	D4	D3	D2	D1	D0	PSW
	.								
B8	-	-	-	BC	BB	BA	B9	B8	IP
	.								
B0	B7	B6	B5	B4	B3	B2	B1	B0	P3
	.								
A8	AF	-	-	AC	AB	AA	A9	A8	IE
	.								
A0	A7	AC	A5	A4	A3	A2	A1	A0	P2
	.								
99	. Not Bit addressable								SBUF
98	9F	9E	9D	9C	9B	9A	99	98	SCON
	.								
90	97	96	95	94	93	92	91	90	P1
8D	. Not Bit addressable								TH1
8C	. “								TH0
8B	. “								TL1
8A	. “								TL0
89	. “								TMOD
88	8F	8E	8D	8C	8B	8A	89	88	TCON
87	. Not Bit addressable								PCON
	.								
83	Not Bit addressable								DPH
82	Not Bit addressable								DPL
81	Not Bit addressable								SP
80	87	86	85	84	83	82	81	80	P0

Figure 2.2:Special Function Registers

2.1.1 Register Banks

The first 32 bytes of internal RAM (00H – 1FH) are structured into Register Banks.. The first 8 bytes (00h - 07h) are "register bank 0". The next 8 bytes (08H – 0FH) constitute “register bank 1” while the following 8 bytes (10H – 17H) and the last 8 bytes (18H – 1FH) form “register bank 2” and “register bank 3” respectively.

The 8051 uses 8 "R" registers which are used in many of its instructions. These "R" registers are numbered from 0 through 7 (R0, R1, R2, R3, R4, R5, R6, and R7). The 8051 set supports registers R0 through R7, and by default, after a system reset; these registers are at addresses OOH-07H. However by manipulating certain SFRs, a program may choose to use register banks 1, 2, or 3.

The registers are generally used to assist in manipulating values and moving data from one memory location to another. For example, to add the value of R4 to the Accumulator, we would execute the following instruction:

ADD A,R4

Also consider,

MOV A, R5

The instruction is a 1-byte instruction using register addressing but could also have the same effect by use of a 2-byte direct addressing.

MOV A, 05H

Instruction using registers R0 to R7 are shorter than the equivalent instructions using direct addressing. Data values used frequently should use one of these registers.

The active register Bank may be altered by changing the register bank select bits in the program status word (PSW). Assuming that register bank 3 is active, the following instruction invites the contents of the accumulator into location 184.

2.1.2 BIT MEMORY (BIT-ADDRESSABLE RAM)

The 8051, being a communications-oriented microcontroller, gives the user the ability to access a number of *bit variables*. These variables may be either 1 or 0.

The 8051 contains 210 bit-addressable locations, of which 128 are at byte addresses 20H through 2FH; and the rest are in the special function in registers. Individual bit accessibility through software is a powerful feature of most micro controllers Bits can be set, cleared, ANDed, ORed, etc, with a single instruction. Most microprocessors require a read-modify-write sequence of instructions to achieve the same effect. Furthermore, the 8051 I/O ports are Bit addressable, simplifying the software interface to single bit inputs and outputs

The user may make use of these variables with commands such as SETB and CLR. For example, to set bit number 24 (hex) to 1 you would execute the instruction:

SETB 24h

It is important to note that Bit Memory is really a part of Internal RAM. In fact, the 128 bit variables occupy the 16 bytes of Internal RAM from 20h through 2Fh. Thus, if you write the value FFh to Internal RAM address 20h you have effectively set bits 00h through 07h. That is to say that:

MOV 20h,#0FFh

is equivalent to:

SETB 00h
SETB 01h
SETB 02h
SETB 03h
SETB 04h
SETB 05h
SETB 06h
SETB 07h

As illustrated, bit memory is not really a new type of memory but a subset of Internal RAM. But since the 8051 provides special instructions to access these 16 bytes of memory on a bit by bit basis it is useful to think of it as a separate type of memory.

2.1.3 General purpose RAM

The 80 bytes remaining of Internal RAM, from addresses 30h through 7Fh, may be used by user variables that need to be accessed frequently or at high-speed. This area is also utilized by the microcontroller as a storage area for the operating stack. This fact severely limits the 8051s stack since, as illustrated in the memory map, the area reserved for the stack is only 80 bytes--and usually it is less since these 80 bytes have to be shared between the stack and user variables.

2.1.4 Special Function Registers (SFRs)

The 8051 is a flexible microcontroller with a relatively large number of modes of operations. A program may inspect and/or change the operating mode of the 8051 by manipulating the values of the 8051's Special Function Registers (SFRs). SFRs are accessed as if they were normal Internal

RAM. The only difference is that Internal RAM is from address 00h through 7Fh whereas SFR registers exist in the address range of 80h through FFh. Each SFR has an address (80h through FFh) and a name, Figure 2.2. The figure 2.2 also shows which SFRs are bit addressable or not.

As can be seen, although the address range of 80h through FFh offer 128 possible addresses, there are only 21 SFRs in a standard 8051. All other addresses in the SFR range (80h through FFh) are considered invalid. Writing to or reading from these registers may produce undefined values or behavior.

2.15 SFR Descriptions

A quick overview each of the standard SFRs found in Figure 2.2 is given below. Full explanation the functionality of each SFR is not given as this will be done as each SFR is encountered in the course of the class under the relevant topics.

(i) P0 (Port 0, Address 80h, Bit-Addressable)

This is input/output port 0. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 0 is pin P0.0, bit 7 is pin P0.7. Writing a value of 1 to a bit of this SFR will send a high level on the corresponding I/O pin whereas a value of 0 will bring it to a low level.

(ii) SP (Stack Pointer, Address 81h)

This is the stack pointer of the microcontroller. This SFR indicates where the next value to be taken from the stack will be read from in Internal RAM. If you push a value onto the stack, the value will be written to the address of SP + 1. That is to say, if SP holds the value 07h, a PUSH instruction will push the value onto the stack at address 08h. This SFR is modified by all instructions which modify the stack, such as PUSH, POP, LCALL, RET, RETI, and whenever interrupts are provoked by the microcontroller.

(iii) DPL/DPH (Data Pointer Low/High, Addresses 82h/83h)

The SFRs DPL and DPH work together to represent a 16-bit value called the *Data Pointer*. The data pointer is used in operations regarding external RAM and some instructions involving code memory. Since it is an unsigned two-byte integer value, it can represent values from 0000h to FFFFh (0 through 65,535 decimal).

(iv) PCON (Power Control, Addresses 87h)

The Power Control SFR is used to control the 8051's power control modes. Certain operation modes of the 8051 allow the 8051 to go into a type of "sleep" mode which requires much less

power. These modes of operation are controlled through PCON. Additionally, one of the bits in PCON is used to double the effective baud rate of the 8051's serial port.

(v) TCON (Timer Control, Addresses 88h, Bit-Addressable)

The Timer Control SFR is used to configure and modify the way in which the 8051's two timers operate. This SFR controls whether each of the two timers is running or stopped and contains a flag to indicate that each timer has overflowed. Additionally, some non-timer related bits are located in the TCON SFR. These bits are used to configure the way in which the external interrupts are activated and also contain the external interrupt flags which are set when an external interrupt has occurred.

(vi) TMOD (Timer Mode, Addresses 89h)

The Timer Mode SFR is used to configure the mode of operation of each of the two timers. Using this SFR your program may configure each timer to be a 16-bit timer, an 8-bit autoreload timer, a 13-bit timer, or two separate timers. Additionally, you may configure the timers to only count when an external pin is activated or to count "events" that are indicated on an external pin.

(vii) TL0/TH0 (Timer 0 Low/High, Addresses 8Ah/8Ch)

These two SFRs, taken together, represent timer 0. Their exact behavior depends on how the timer is configured in the TMOD SFR; however, these timers always count up. What is configurable is how and when they increment in value.

(viii) TL1/TH1 (Timer 1 Low/High, Addresses 8Bh/8Dh)

These two SFRs, taken together, represent timer 1. Their exact behavior depends on how the timer is configured in the TMOD SFR; however, these timers always count up. What is configurable is how and when they increment in value.

(ix) P1 (Port 1, Address 90h, Bit-Addressable)

This is input/output port 1. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 1 is pin P1.0, bit 7 is pin P1.7. Writing a value of 1 to a bit of this SFR will send a high level on the corresponding I/O pin whereas a value of 0 will bring it to a low level.

(x) SCON (Serial Control, Addresses 98h, Bit-Addressable)

The Serial Control SFR is used to configure the behavior of the 8051's on-board serial port. This SFR controls the baud rate of the serial port, whether the serial port is activated to receive data, and also contains flags that are set

(xi) SBUF (Serial Control, Addresses 99h)

The Serial Buffer SFR is used to send and receive data via the on-board serial port. Any value written to SBUF will be sent out the serial port's TXD pin. Likewise, any value which the 8051 receives via the serial port's RXD pin will be delivered to the user program via SBUF. In other words, SBUF serves as the output port when written to and as an input port when read from.

(xii) P2 (Port 2, Address A0h, Bit-Addressable)

This is input/output port 2. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 2 is pin P2.0, bit 7 is pin P2.7. Writing a value of 1 to a bit of this SFR will send a high level on the corresponding I/O pin whereas a value of 0 will bring it to a low level.

(xiii) IE (Interrupt Enable, Addresses A8h)

The Interrupt Enable SFR is used to enable and disable specific interrupts. The low 7 bits of the SFR are used to enable/disable the specific interrupts, where as the highest bit is used to enable or disable ALL interrupts. Thus, if the high bit of IE is 0 all interrupts are disabled regardless of whether an individual interrupt is enabled by setting a lower bit.

(xiv) P3 (Port 3, Address B0h, Bit-Addressable)

This is input/output port 3. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 3 is pin P3.0, bit 7 is pin P3.7. Writing a value of 1 to a bit of this SFR will send a high level on the corresponding I/O pin whereas a value of 0 will bring it to a low level.

(xv) IP (Interrupt Priority, Addresses B8h, Bit-Addressable)

The Interrupt Priority SFR is used to specify the relative priority of each interrupt. On the 8051, an interrupt may either be of low (0) priority or high (1) priority. An interrupt may only interrupt interrupts of lower priority. For example, if we configure the 8051 so that all interrupts are of low priority except the serial interrupt, the serial interrupt will always be able to interrupt the system, even if another interrupt is currently executing. However, if a serial interrupt is executing no other interrupt will be able to interrupt the serial interrupt routine since the serial interrupt routine has the highest priority.

(xvi) PSW (Program Status Word, Addresses D0h, Bit-Addressable)

The Program Status Word is used to store a number of important bits that are set and cleared by 8051 instructions. The PSW SFR contains the carry flag, the auxiliary carry flag, the overflow flag, and the parity flag. Additionally, the PSW register contains the register bank select flags which are used to select which of the "R" register banks are currently selected.

(xvii) ACC (Accumulator, Addresses E0h, Bit-Addressable)

The Accumulator is one of the most-used SFRs on the 8051 since it is involved in so many instructions. The Accumulator resides as an SFR at E0h, which means the instruction **MOV A,#20h** is really the same as **MOV E0h,#20h**. However, it is a good idea to use the first method since it only requires two bytes whereas the second option requires three bytes.

(xviii) B (B Register, Addresses F0h, Bit-Addressable)

The "B" register is used in two instructions: the multiply and divide operations. The B register is also commonly used by programmers as an auxiliary register to temporarily store values.

(xix) Other SFRs

The chart above is a summary of all the SFRs that exist in a standard 8051. All derivative microcontrollers of the 8051 must support these basic SFRs in order to maintain compatibility with the underlying MSCS51 standard. A common practice when semiconductor firms wish to develop a new 8051 derivative is to add additional SFRs to support new functions that exist in the new chip. For example, the Dallas Semiconductor DS80C320 is upwards compatible with the 8051. This means that any program that runs on a standard 8051 should run without modification on the DS80C320. This means that all the SFRs defined above also apply to the Dallas component. However, since the DS80C320 provides many new features that the standard 8051 does not, there must be some way to control and configure these new features. This is accomplished by adding additional SFRs to those listed here. For example, since the DS80C320 supports two serial ports (as opposed to just one on the 8051), the SFRs SBUF2 and SCON2 have been added. In addition to all the SFRs listed above, the DS80C320 also recognizes these two new SFRs as valid and uses their values to determine the mode of operation of the secondary serial port. Obviously, these new SFRs have been assigned to SFR addresses that were unused in the original 8051. In this manner, new 8051 derivative chips may be developed which will run existing 8051 programs.

CHAPTER THREE

ADDRESSING MODES

3.0 Introduction

Addressing defines the ways in which operands are located in an instruction or the various ways in which memory is addressed. Addressing modes are integral part of each computer's instruction set. They allow specifying the source or destination of rate in different ways, depending on the programming situation.

The 8051 has light addressing modes including

1. Register
2. Direct
3. Indirect
4. Immediate
5. Relative
6. Absolute
7. Long
8. Indexed

3.1 Register Addressing

The 8051 has access to eight working registers numbered R1 through R7. Instructions using register addressing are encoded using the three least-significant bits of the instruction opcode to specify a register within this logical address space. Thus, a function code and operand address can be combined to form a short (1-byte) instruction, Figure 3.1.

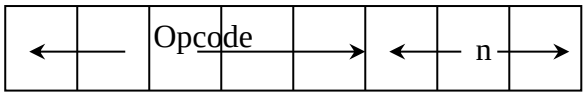


Figure 3.1: Opcode for register addressing

In Figure 3.1, n is from 0 to 7. Thus, to add the contents of Register 7 to the accumulator, the following instruction is used.

ADD A, R7

and the opcode is 00101111B. The upper five bits 00101 indicate the instruction and the lower three bits, 1111 the register. Generally, the 8051 assembly language indicates register addressing with the symbol R_n where n is from 0 to 7. Consider the instruction,

Mov A, R_n

The binary opcode appears as 11101ααα. The low-order three bits identify the source registers.

There are four banks of working registers' but only one is active at a time with the program status word (PSW) bits 4 and 3 determining the active bank.

A hardware reset enables bank 0 but a different bank is selected by modifying PSW bits 4 and 3 accordingly. E.g.

MOV PSW, #00011000B

OR MOV PSW, #18H

activities register bank 3 by setting the register bank select bits (R50 and R51) in PSW bit positions 4 and 3. Some instructions are specific to a certain register such as the accumulator, data printer, etc so address bits are not needed. The opcode itself indicates the register. Other such registers include program counter (PC), carry flag (C), accumulator-B register pair (AB). E.g.

INC DPTR

is a 1-byte instruction that adds to the 16-bit data printer.

3.2 Direct Addressing

Direct addressing can access any on-chip variable or hardware register. An additional byte is added to the opcode specifying the location to be used, Figure 3.2.

Figure 3.2: OPcode for direct addressing

e.g. ADD A, 55H

Depending on the high-order bit of direct address, one of two on-chip memory spaces is selected. When bit 7=0, address 0 - 127 (00H - 7FH) is referenced while 128 - 256 (80H – FFH) is referenced when bit 7=1. 128 - 256 refer to the special function registers including the I/O ports, control and status registers. For example, ports 0 and 1 have addresses 80H and 90H respectively. Using direct address here should require memorizing the address of these registers. This is however not necessary because the assembler allows for and understands the mnemonic abbreviations P0 for port 0, TMOD for timer mode register etc. For example, the following instruction using direct addressing transfers the content of the accumulator to port 1.

MOV P1, A

The direct address of port 1 (90H) is determined by the assembler and inserted as byte 2 of the instruction. The source of the data, accumulator, is specified implicitly in the opcode as mentioned before.

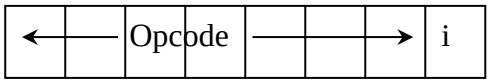
MOV SCON, #55H

moves the numeric value 55H to the SCON special function register whose direct address value is added by the assembler.

3.3 Indirect Addressing

When a program is running an address of a variable may be determined, computed or modified. This is common when manipulating sequential memory locations, indexed entries within tables in RAM, multiple-precision numbers or character strings. Register or direct addressing cannot be used since they require operand address to be known at assemble-time.

The 8051 addresses the situation by indirect addressing. R0 and R1 may operate as 'pointer' registers with their contents indicating an address in internal RAM where data are written or read. The least significant bit of the instruction opcode determines which register (R0 or R1) is used as a pointer, figure 3.3.



i ≡ defines whether R0 or R1 is used.

Figure 3.3: Opcode for indirect addressing

Example: If R1 contains 40H and internal memory address 40H contains 55H, the following instruction moves 55H into the accumulator.

```
MOV A, @R1
```

Indirect addressing is essential when stepping through sequential memory locations. For example, the following instruction sequence clears internal RAM from address 60H to 7FH.

```
          Mnemonic
          MOV R0, #60H
Loop:     MOV @R0, #0
          INC R0
          CJNE R0, #80H, LOOP
          Continue
```

The first instruction MOV R0, #60H initializes R0 with the starting address of the block of memory. The second Loop: MOV @R0, #0 uses indirect addressing to clear location 60H by moving 00H into it. The third instruction increments the pointer (R0) to the next address while the last instruction tests the pointer to see if the end of the block has been reached. The text uses 80H to ensure that 7FH is written into before terminating.

3.4 Immediate Addressing

When a source operand is a constant, rather than a variable, the constant is incorporated into the instruction as a byte of 'immediate' data, Figure 3.4.

Figure 3.4: Opcode for immediate addressing

In assembly language, immediate operands are preceded by a number sign (#). The operand may be a numeric constant, a symbolic variable, or an arithmetic expression using constants, symbols and operators. The assembler computes the value and substitutes the immediate data into the instruction. Example,

MOV A, #12

loads the value 12(0CH) into the accumulator.

All instructions using immediate addressing use an 8-bit data constant for the immediate data. When initializing the data pointer, a 16-bit constant is required. Example,

MOV DPTR, #8000H

is a 3-byte instruction that loads 16-bit constant 8000H into the data pointer.

Generally, immediate addressing instructions are of the form

ADD A, # data for an arithmetic add operation.

3.5 Relative Addressing

Relative addressing is used only with certain jump instructions. A relative address (or offset) is an 8-bit signed value, which is added to the program counter to form the address of the next instruction executed. Since an 8-bit offset is used, the range of jumping is 128 to 127 locations. The relative offset is appended to the instruction as an additional byte, figure 3.5.

Figure 3.5: Opcode for relative addressing

e.g. SJMP AHEAD or SJMP BACKWARD

Prior to the addition, the program counter is incremented to the address following the jump instruction. Thus, the new address is relative to the next instruction, not the address of the jump instruction, Figure 3.6.

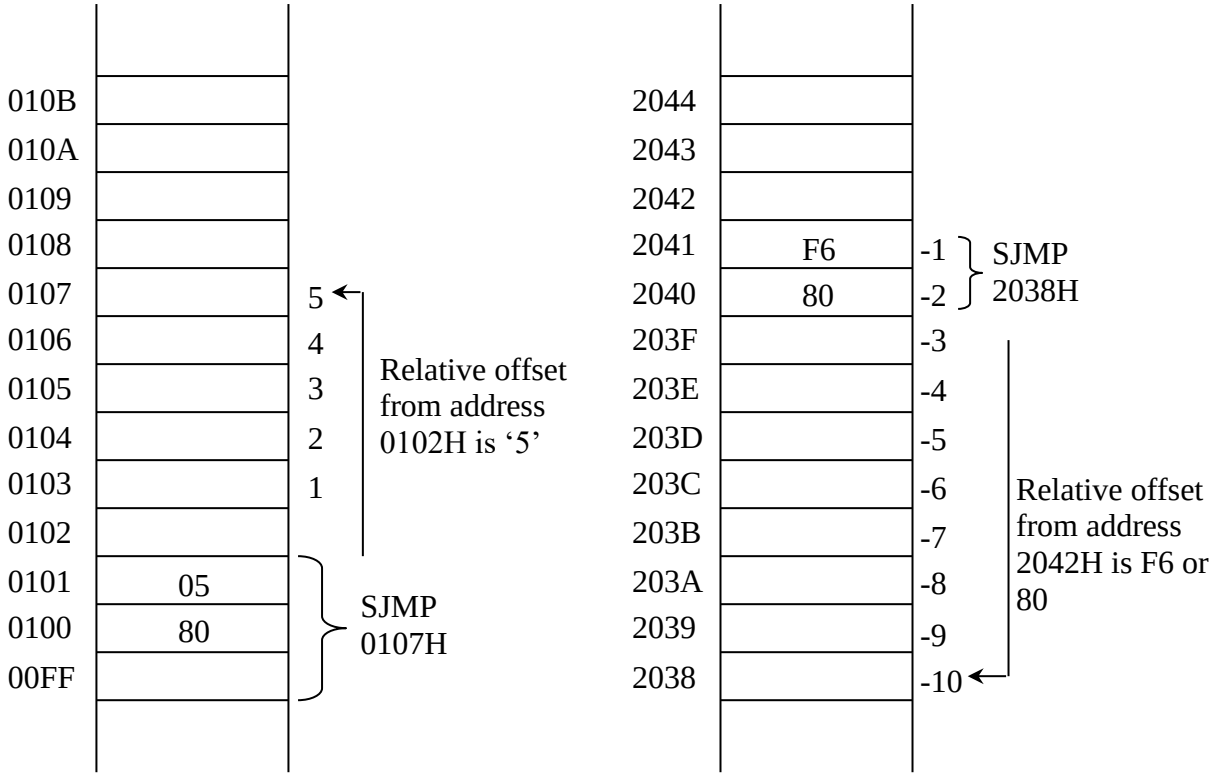

010B
010A
0109
0108
0107
0106
0105
0104
0103
0102
0101
0100
00FF
05
80
5
4
3
2
1
Relative offset from address 0102H is '5'
SJMP 0107H
2044
2043
2042
2041
2040
203F
203E
203D
203C
203B
203A
2039
2038
F6
80
-1
-2
-3
-4
-5
-6
-7
-8
-9
-10
SJMP 2038H
Relative offset from address 2042H is F6 or 80

Figure 3.6 (a) and (b): Illustration of relative addressing

Normally, this detail does not concern a programmer since jump destinations are usually specified as labels and the assembler determines the relative offset accordingly. For example, if the label LAB represents an instruction at location 1040H, and the instruction

SJMP LAB

is in memory at location 1000H and 1001H, the assembler will assign a relative offset of 3EH as byte 2 of the instruction, i.e. 1002H + 3EH = 1040H.

The advantage of relative addressing is that it provides position – independent code while the disadvantage is that the jump destinations are limited in range.

3.6 Absolute Addressing

Absolute addressing is used only with the ACALL and AJUMP instructions. These 2-byte instructions allow branching within the current 2K page of code memory by providing the 11 least significant bits of the destination address in the opcode (A10 – A8) and byte 2 of the instruction (A7 – A0), figure 3.7

.

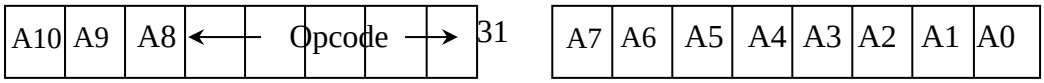


Figure 3.7: Opcode for absolute addressing

The upper five bits of the destination address are the current upper bits in the program counter, so the instruction following the branch instruction and the destination for the branch instruction must be within the same 2k page, since A15 – A11 do not change, Figure 3.8.

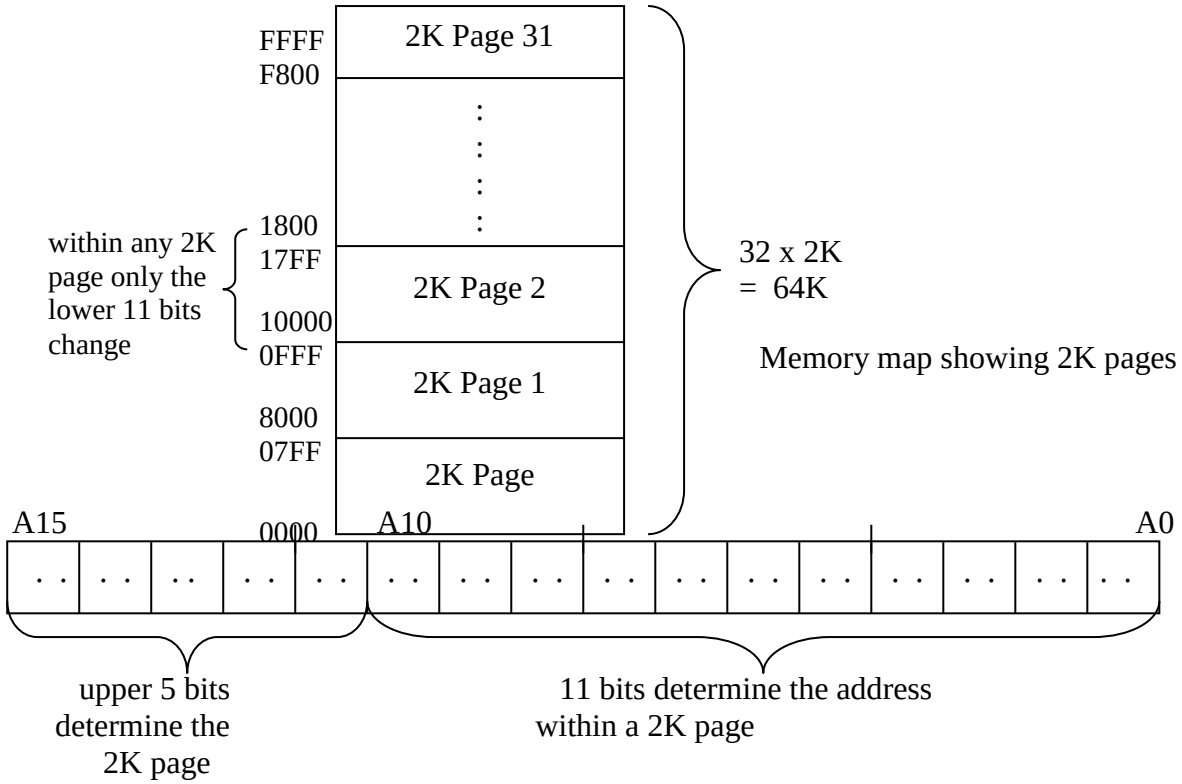


Figure 3.8: Bits definition within any 2K page

Example

If the label LAB represents an instruction at address of 46H and the instruction

AJUMP LAB

is in memory location 0900H and 0901H, the assembler will encode the instruction as

<u>111</u> 00001 – 1st byte (A10 – A8 + opcode)

<u>01000110</u> - 2nd byte (A7 – A0)

The underlined bits are the low-order 11 bits of the destination address, 0F46H = 0000111101000110B. The upper five bits in the program counter will not change when this instruction executes. It is pertinent to note that both the AJMP instruction and the destination are within the 2K page bounded by 0800H and 0FFFH (see figure on memory map showing 2K pages)

and therefore have the upper five address bits in common. (i.e. 0900H, 0901H, 0902H and 0F46H).

Absolute addressing offers the advantage of short (2-byte) instructions but has the disadvantages of limiting the range for the destination and providing position-dependent code. Generally, the encoding for AJMP is as shown in Figure 3.9.

a	a	a	0	0	0	0	1	a	a	a	a	a	a	a	a

a a a = A10 – A8, a a a a a a a a = A7 – A0 of the destination address

Figure 3.9: Illustration of absolute addressing

Example

An ACALL instruction is in memory locations 1024H and 1025H. The subroutine to which the call is directed begins in memory location 17A6H. What are the machine language bytes for the ACALL instructions.

Solution

From the instruction set, the general form for encoding ACALL is

Aaa10001aaaaaaaa

But aaa and aaaaaaaa must be the low-order 11 bits of the destination address which is 17A6H. hence the bytes are

10001aaa aaaaaaaa

But 17A6H = 00010111 10100110B

∴ The underlined bits are the low-order 11 bits of the destination address which must be address to the general form of the machine language bytes.

11110001 10100110B = F1A6H

Note that absolute addressing can only be used if the high-order five bits are the same in both the source and destination addresses since these bits form the identifier that show both the source and destination addresses as falling within the same 2K page.

3.7 Long Addressing

Long addressing is used only with the LCALL and LJMP instructions. These 3-byte instructions include a full 16-bit destination address as byte 2 and 3 of the instruction, Figure 3.10.

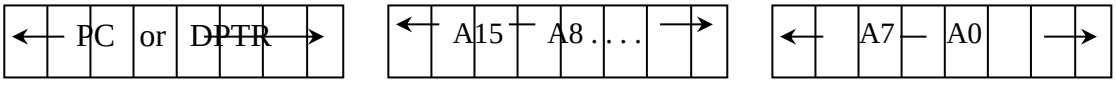


Figure 3.10: Long addressing

The advantage is that the full 64K code space may be used, but the disadvantage is that the instructions are three bytes long and are position-dependent. Position-dependence is a disadvantage because the program cannot execute at different addresses. If for example, a program begins at 2000H and an instruction such as LJMP 2040H appears, then the program cannot be moved to say, 4000H. The LJMP instruction will still jump to 2040H which is not the correct location after the program has been moved.

Example: What are the machine language bytes for the instruction?

LJMP 8AF2H

From the instruction set, LJMP is a 3-bytes instruction with the opcode of 02H in the first byte and high-order byte of destination address in the second byte and low-order byte of destination address in the third byte. Hence the machine language bytes are 02H, 8AH, F2H.

3.8 Indexed Addressing

Indexed addressing uses a tase register (either the program counter or the data pointer) and an offset (the accumulator) in forming the effective address for a JMP or MOV instruction, Figure 3.12.

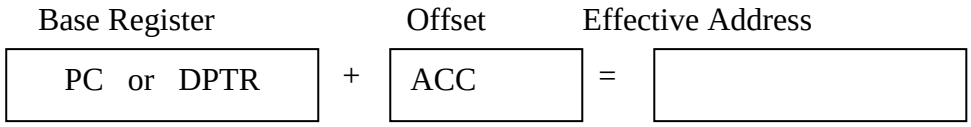


Figure 3.12: Indexed addressing

JUMP tables or look-up tables are easily created using indexed addressing.

MOVC A, @ A + <base-register>
JMP @ A + DPTR

Example

What is the opcode for the instruction?

MOVC A, @ A + DPTR

Solution

From the instruction set, the instruction is a 1-byte instruction with 93H, which specify both the operation and the addressing mode. This instruction moves a byte of data from code memory to the accumulator. The address in code memory is found by adding the index (the present of the accumulator) to the base register (the data pointer). When the instruction finishes executing, the index is lost because it has overwritten with the value moved from code memory

CHAPTER FOUR

8051 INSTRUCTION TYPES

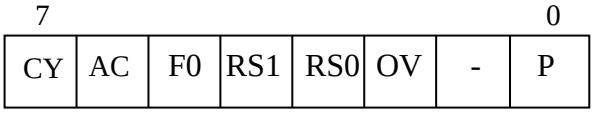
The 8051 instructions are divided among five functional groups

1. Arithmetic
2. Logical
3. Data transfer
4. Boolean variables
5. Program branching

4.1 Arithmetic Instructions

In discussing these instructions, it is expedient that the Program Status Word (PSW) is first understood.

The Program Status Word (PSW) at address DOH contains status bits as shown in Figure 4.1.



Bit	Address	Bit Description
PSW.7	D7H	Carry Flag
PSW.6	D6H	Auxiliary carry flag
PSW.5	D5H	Flag 0
PSW.4	D4H	Register bank select 1
PSW.3	D3H	Register bank select 0 00 = bank 0; addresses 00H-07H 01 = bank 1; addresses 08H-0FH 10 = bank 2; addresses 10H-17H 11 = bank 3; addresses 18H-1FH
PSW.2	D2H	Overflow flag
PSW.1	D1H	Reserved
PSW.0	D0H	Even parity flag

Figure 4.1: Program Status Word

1. The carry flag (C or CY) is dual-purpose. It is set if there is a carry out of bit 7 during an add, or set if there is a borrow into bit 7 during a subtract. Example, if the accumulative FFH contains

   ADD A, #1

   will leave the result 00H in the accumulator and set the carry flag in the PSW

The carry flag is also the Boolean accumulator' serving as a 1-bit register for Boolean instructions operating on bits. Example, the following instruction ANDs bit 25H with the carry flag and places the result back in the carry flag.

```
ANL C, 25H
```

2. **Auxiliary Carry Flag (AC)**

When working with BCD values, the AC flag is set if a carry was generated out of bit 3 into bit 4 or if the result in the lower nibble is in the range 0AH-0FH. For BCD values, after the addition instruction, the DAA (decimal adjust accumulator) must follow to bring results greater than 9 back into range.

3. **Flag 0**

This is a general purpose flag bit available for user-applications

4. **Register bank select bits**

The register bank select bits (RS0 and RS1) determine the active register bank. They are cleared after a system reset and are changed by software as needed.

Example, the following instructions enable register bank 3 and then move the content of R7 to the accumulator.

```
SETB RS1
SETB RS0
MOV A, R7
```

At assembly time, the correct bit addresses are substituted for the symbols 'RS1' and 'RS0'. Thus,

```
SETB RS1
```

is the same as

```
SETB 0D4H
```

5. **Overflow flag (OV)**

The OV flag is set after an addition or subtraction operation if there was an arithmetic overflow. When signed numbers are added or subtracted, software can examine this bit to determine if the result is in the proper range when unsigned numbers are added, the OV bits can be ignored. Results greater than + 127 or less than -128 will set the OV bit.

6. Parity Bit

The parity bit (P) is automatically set or cleared each machine cycle to establish even parity with the accumulator. The number of 1 bit in the accumulator plus the P is at ways even.

Example, if the accumulator contains 10101101, P will contain 1 (to make it an even no of 1s). The parity bit is most commonly used in conjunction with serial port routines to include a parity bit before transmission or to check for parity after reception.

Example: The accumulator contains 63H, R3 contains 23H, and the PSW contains 00H. What is the hex content of the accumulator and the PSW after executing the instruction.

```
        ADD A, R3
Soln: ACC = 86H, PSW = 05H
CY = 0, OV = 0, P = 1, AC = 0
```

If signed numbers were assumed, then OV = 1

4.1.1 Arithmetic Instructions and Addressing Modes

Various addressing modes are possible with the arithmetic group of instructions. Example, the ADD A instruction can be written in the following ways.

```
ADD A, 07H        (Direct addressing)
ADD A, @R0        (Indirect addressing)
ADD A, R7         (Register addressing)
ADD A, #35H (Immediate addressing)
```

All arithmetic instructions execute one machine cycle except the INC DPTR instruction (two machine cycles) and MULAB and DIVAB instructions (four machine cycles).

Example

Write an instruction sequence to subtract the content of R6 from R7 and leave the result in R7.

Solution

```
Mnemonic
MOV A, R7
CLR C
SUBB R6
MOV R7, A
```

The data pointer (DPTR) is a 16-bit register that generates a 16-bit addresses for external memory thus incrementing it in one operation is a useful feature. However, decrementing the data pointer instruction is not provided for in 8051 and so a sequence of instructions are required.

Example:

```
        Mnemonic
        DEC DPL                 ; Decrement low-byte of DPTR
        MOV R7, DPL             ; move to R7
        CJNE R7, #0FFH, SKIP    ; if underflow to FF
        DEC DPH                 ; decrement high-byte too
SKIP: continue
```

It is observed that the high - and low-bytes of the DPTR must be decremented separately. However, the high byte (DPH) is only decremented if the low-byte (DPL) underflows from 00H to FFH.

The MUL AB instruction multiplies the accumulator by the data in the B register and puts the 16-bit product into the concatenated B (high byte) and accumulator (low-byte) registers. DIV AB divides the accumulator by the data in the B register, leaving the 8-bit quotient in the accumulator and the 8-bit remainder in the B register.

4.2 Logical Operations

The 8051 logical instructions perform Boolean operations (AND, OR, Exclusive OR and NOT) on bytes of data on a bit-by-bit basis. The same addressing modes used for arithmetic instructions are also used for logical instructions. Thus,

```
ANL A, 55H          (Direct addressing)
ANL A, @R0          (Indirect addressing)
ANL A, R6           (Register addressing)
ANL A, #33H (Immediate addressing)
```

All logical instructions using the accumulator as one of the operands execute in one machine cycle. The others take two machine cycles.

Example

```
XRL P1, #0FFH
```

This instruction performs a read-modify-write operation. The eight bits at port 1 are read, then each bit read is exclusively ORed with the corresponding bit in the immediate data. Since the eight bits of immediate data are all 1s the effect is to complement each bit read. The result is written back to port 1.

The rotate instructions (RLA and RRA) shift the accumulator one bit to the left or right. For a left rotation, the MSB rolls into the LSB position. The reverse is true for a right rotation.

The RLL A and RRC A are a 9-bit rotation using the accumulator and the carry flag in the PSW. For example, if the carry flag contains 1 and A contains 00H, then the instruction

```
RRC A
```

leaves the carry flag clear and A equals 80H. The carry bit rotates into ACC.7 and ACVC.0 rotates into the carry flag.

The SWAP A instructions exchanges the high and low nibbles within the accumulator. This is a useful operation in BCD manipulations. For instance, if the accumulator contains a binary number that is known to be less than 100_{10}, it is quickly converted to BCD as follows:

```
MOV B, #10
DIV A B
SWAP A
ADD A, B
```

Dividing the number by 10 in the first two instructions leaves the tens digit in the low nibble of the accumulator and the ones digit in the B register. The SWAP and ADD instructions move the tens digit to the high nibble of the accumulator, and the ones digit to the low nibble.

Example

Write a program to reverse the bits in the accumulator. Bit 7 and bit 0 are swapped, bit 6 and bit 1 are swapped, etc.

Mnemonic

```
      MOV R7, #8
Loop: RLC  A
      XCH A, 0F0H
      RRC  A
      XCH A, 0F0H
      DJNZ R7, LOOP
      XCH A, 0F0H
```

The approach adopted here is to build the new value in the B register by successively shifting a bit out of the accumulator into the carry bit and then shifting the same bit back into the B register. To reverse the bit pattern, the first shift is to the left and second shift is to the right. Because the rotate instruction only operates on the accumulator, the B register and the accumulator are exchanged (XCH) following each rotate. A final XCH positions the correct result in the accumulator. Note that the B register is at direct address 0F0H.

NOTE: 16-bit arithmetic here

4.3 Data Transfer Instructions

Internal RAM:

The instruction format is

MOV <destination> <source>

This format allows data to be transferred between any two internal RAM or SFR locations without going through the accumulator. The stack residues in on-chip RAM for 8051. The PUSH instruction first increments the stack pointer (SP), then copies the byte into the stack. PUSH and

POP use direct addressing to identify the byte being saved or restored, but the stack itself is accessed by indirect addressing using the SP register. For 8051, if the SP is advanced above 7FH (127), the PUSHed bytes are lost and the POPed bytes are indeterminate.

Example

The stack pointer contains 07H, accumulator A contains 55H, and accumulator B contains 4AH. What internal RAM locations are altered and what are their need values after executing the following instructions.

```
PUSH ACC
PUSH 0F0H
```

Solution

Address	Contents
08H	55H
09H	4AH
81H(SP)	09H

4.5 16-Bit Mathematics

The 8051 is an 8-bit microcontroller. This means that each machine language opcode in its instruction set consists of a single 8-bit value. This permits a maximum of 256 instruction codes, however, only 255 are actually used.

The 8051 works almost exclusively with 8-bit values. All the registers except PC and DPTR are 8-bit. The PC internally indicates the next instruction to be executed while the DPTR may be used to access external RAM as well as directly access code memory. Even though the 8051 provides a number of instructions aimed at performing mathematical calculations, they all work with 8-bit input values. But often, we find ourselves working with values that cannot be expressed in 8-bits.

4.5.1 16-BIT ADDITION

16-bit addition is the addition of two 16-bit values. Recognize that the addition of two 16-bit values with result in a value that is at most, 17 bits long. Thus, the addition of two 16-bit values will produce a 3-byte answer. Since only one bit is used in the third byte, the carry bit could be used to hold this bit. Either way is acceptable.

Example

Add the decimal numbers. 6724 and 8923

Solution

First convert the numbers to hexadecimal

6724 = 1A44h

8923 = 22DBh

To understand the addition process, recall your primary school addition process.

SIGN	100s	10s	1s
	1	0	1
+	2	1	2
=	3	1	3

Primary school addition

From the above illustration, we add as follows:

SIGN	256s	1s
	1A	44
+	22	DB
=	3D	1F

Here, we are just working with two columns: the 1's column and the 256's column. In other words, we are dealing with the low-byte (the 1's column) and the high byte (the 256's column). But the process is the same.

The challenge is how to code the above process. Let us use the following table to explain the process.

SIGN	65536's	256's	1's
		R6	R7
+		R4	R5
	R1	R2	R3

Let the first number to be added be held in R6 and R7 with the high byte in R6 and the low byte in R7, while the second value to be added is held in R4 and R5 with the high byte in R4 and the low byte in R5. Also, let the answers be left in R1, R2 and R3. Note that only 1 bit of the answer will be held in R1.

Outline of the steps for the addition

1. Add the low bytes R7 and R5, and leave the answer in R3.
2. Add the high bytes R6 and R4, adding any carry from step 1, and leave the answer in R2.
3. Put the any carry from step 2 in the final byte, R1.

The program is as shown below:

Step 1

Add the low bytes R7 and R5, leave the answer in R3.

```
Mnemonic
MOV   A, R7          ;  move low-byte into the accumulator
ADD   A, R5          ;  add the second low byte to accumulator
MOV   R3, A          ;  place result in R3
```

Step 2

Add the R6 and R4, Add carry, leave result in R3

```
Mnemonic
MOV   A, R6          ;  move the high-byte into the accumulator
ADDC  A, R4          ;  add the second high-byte
MOV   R2, A          ;  place result in R2
```

Step 3

Put any carry from step 2 in the final byte, R1

```
Mnemonic
MOV   A, #00h        ;  By default, the highest byte is zero
ADDC A, #00h         ;  Add zero, plus carry from step 2
MOV   R1, A          ;  place result in R1
```

The combined code will therefore be:

```
ADD16 _ 16:
        ; step 1 of the process
        MOV   A, R7          ;  move low-byte into accumulator
        ADD   A, R5          ;  Add 2nd low-byte
        MOV   R3, A          ;  place the result in R3
        ; step 2 of the process
        MOV   A, R6          ;  move high-byte into accumulator
        ADDC A, R4           ;  Add 2nd high byte to acc. Plus carry
        MOV   R2, A          ;  place result in R2
        ; step 3 of the process
        MOV   A, #00h        ;  By default, the highest byte will be zero
        ADDC A, #00h         ;  Add zero, plus carry from step 2
        MOV   R1, A          ;  move the answer to R1
        ; Return _ answer now resides in R1, R2 and R3
        RET
```

NOW, to call our routine to add the two values used in the example, we would use the code:

```
        Mnemonic
        ; Load the first value into R6 and R7
        MOV   R6, #1Ah
        MOV   R7, #44h
```

```
; load the second value into R4 and R5
MOV   R4, #22h
MOV   R5, #0DBh

; call the 16-bit addition routine
LCALL   ADD 16 _ 16
```

4.5.2 16-BIT SUBTRACTION

16-bit subtraction is the subtraction of one 16-bit value from another. A subtraction of this nature results in another 16-bit value.

Example

Consider the subtraction: 8923 – 6905

The first step is to convert the values to hexadecimal

8923 = 22DBh

6905 = 1AF9h

Again, our primary school subtraction

SIGN	256's	1's
	22	DB
-	1A	F9
=	07	E2

First, we subtract the second value in the 1's column (low byte): DB – F9. since F9 is greater than DB, we need to borrow from the 256's column. Thus, we are actually subtracting 1DB – F9 = E2. Now 22 – 1A subtraction is carried out. Recall that we borrowed 1 from the 256's column, so we must subtract an additional 1. So the subtraction becomes 22 – 1A – 1 = 07, which is left in 256's column.

Let's illustrate the process again with a table.

SIGN	256's	1's
	R6	R7
-	R4	R5
=	R2	R3

Let R6 and R7 hold the value to be subtracted from while the value to be subtracted is held in R4 and R5. Let the answer be left in R2 and R3.

Outline of steps for the subtraction

1. Subtract the low byte R5 from R7 and leave result in R3.
2. Subtract the high byte R4 from R6, less any borrow and leave the answer in R2.

Converting the above process to assembly language, step by step gives:

Step 1

Subtract the low byte R5 from R7, leave the answer in R3.

```
Mnemonic
MOV   A, R7        ;  move the low-byte into accumulator
CLR   C            ;  always clear carry before first subtraction
SUBB  A, R5        ;  subtract 2nd low-byte from the accumulator
MOV   R3, A        ;  leave result in R3
```

Step 2

Subtract the high byte R4 from R6, less any borrow and leave the answer in R2.

```
Mnemonic
MOV   A, R6        ;  move the high byte into the accumulator
SUBB  A, R4        ;  subtract the second high byte from the acc.
MOV   R2, A        ;  leave the answer in R2
```

Combining the code from the two steps above, we have the following subroutine:

```
SUBB 16_16:

    ; step 1 of the process
    MOV   A, R7
    CLR   C
    SUBB  A, R5
    MOV   R3, A

    ; step 2 of the process
    MOV   A, R6
    SUBB  A, R4
    MOV   R2, A
    ; Return_answer now resides in R2 and R3
    RET
```

Now, to call our routine to subtract the two values used in the example, we could use the code,

```
    ; Load the first value into R6 and R7
    MOV   R6, #22h
    MOV   R7, #0DBH

    ; Load the second value into R4 and R5
    MOV   R4, #1Ah
    MOV   R5, #0F9h

    ; call the 16-bit subtraction routine
    LCALL   SUBB 16_16
```

4.5.3 16-BIT MULTIPLICATION

16-bit multiplication is the multiplication of two 16-bit value from another. Such a multiplication results in a 32-bit value.

Note that any multiplication results in an answer which is the sum of the bits in the two multiplicands.

Example

Let us multiply 25, 136 by 17, 198. The answer is 432, 288, 928.

First, we convert the values to hexadecimal.

25,136 = 6230h

17,198 = 432Eh

Once again, bet us full back on our primary school multiplication.

SIGN	Byte 4	Byte 3	Byte 2	Byte 1
			62	30
*			43	2E
			08	A0
		11	9C	
		OC	90	
	19	A6		
	10	C4	34	A0

The process in assembly language will be identical to the multiplication table. Let us illustrate it with a table once again.

SIGN	Byte 4	Byte 3	Byte 2	Byte 1
			R6	R7
*			R4	R5
=	R0	R1	R2	R3

The first number is contained in R6 and R7 while the second number is contained in R4 and R5. The result of the multiplication is held in R0, R1, R2 and R3. At 8-bits per register, these four registers give the 32-bits needed to handle the largest possible multiplication.

Outline of the steps for the process

1. Multiply R5 by R7, leaving the 16-bit result in R2 and R3.
2. Multiply R5 by R6, adding the 16-bit result into R1 and R2.
3. Multiply R4 by R7, adding the 16-bit result to R1 and R2.
4. Multiply R4 by R6, adding the 16-bit result to R0 and R1.

Let us now convert the above process to assembly language, step by step.

Step 1

Multiply R5 by R7, leaving the 16-bit result in R2 and R3.

```
MOV   A, R5       ;  move the R5 into the accumulator
MOV   B, R7       ;  move R7 into B
MUL   AB          ;  multiply the two values
MOV   R2, B       ;  move high byte of result into R2
MOV R3, A         ;  move low byte of result into R3
```

Step 2

Multiply R5 by R6, adding the 16-bit result to R1

```
MOV   A, R5       ;  move R5 into the accumulator
MOV   B, R6       ;  move R6 into B
MUL   AB          ;  multiply the two values
ADD   A, R2       ;  add the low byte to result already in R2
MOV   R2, A       ;  move the result back into R2
MOV   A, B        ;  move high byte into the accumulator
ADD   A, #00h     ;  add zero (plus carry if any)
MOV   R1, A       ;  move the result into R1
MOV   A, #00h     ;  load accumulator with zero
ADDC A, #00h      ;  add zero (plus carry, if any)
MOV   R0, A       ;  move the resulting answer to R0
```

Step 3

Multiply R4 by R7, adding the 16-bit result to R1 and R2.

```
MOV A, R4         ;  Move R4 into the accumulator
MOV   B, R7       ;  Move R7 into B
MUL   AB          ;  Multiply the two values
ADD   A, R2       ;  Add the low-byte to the value already in R2
MOV   R2, A       ;  Move the resulting answer back into R2
MOV   A, B        ;  Move the high-byte into the accumulator
ADDC A, R1        ;  Add the current value of R1 (plus any carry)
MOV   R1, A       ;  Move the resulting answer into R1
MOV   A, #00h     ;  Load the accumulator with zero
```

```
ADDC A, R0          ;  Add the current value of R0 (plus any carry)
MOV  R0, A          ;  Move the resulting answer to R0
```

Step 4

Multiply R4 by R6, adding the 16-bit result to R0 and R1

```
MOV  A, R4          ;  move R4 back into the accumulator
MOV  B, R6          ;  move R6 into B
MOV  AB             ;  multiply the two values
ADD  A, R1          ;  add the low-byte the value already in R1
MOV  R1, A          ;  move the resulting value back into R1
MOV  A, B           ;  move the high byte into the accumulator
ADD C A, R0         ;  add it to the value already in R0 plus
                    ;  any carry
MOV  R0, A          ;  move the resulting answer back to R0
```

Combining the code from the two steps above, the following subroutine results:

```
MUL 16_16:
        ; multiply R5 by R7
        MOV  A, R5
        MOV  B, R7
        MUL  AB
        MOV  R2, B
        MOV  R3, A

        ; multiply R5 by R6
        MOV  A, R5
        MOV  B, R6
        MUL  AB
        ADD A, R2
        MOV  R2, A
        MOV  A, B
        ADDC A, #00h
        MOV  R1, A
        MOV  A, #00h
        ADDC A, #00h
        MOV  R0, A

        ; multiply R4 by R7
        MOV  R, R4
        MOV  B, R7
        MUL  AB
        ADD  A, R2
        MOV  R2, A
        MOV  A, B
        ADDC A, R1
        MOV  R1, A
        MOV  A, #00h
```

```
ADDC A, R0
MOV   R0, A

; multiply R4 by R6
MOV   A, R4
MOV   B, R6
MUL   AB
ADD   A, R1
MOV   R1, A
MOV   A, B
ADD C A, R0
MOV   R0, A

; Return-answer now in R0, R1, R2 and R3
RET
```

Now to call routine to multiply the two values used in the example, we would use the code.

```
; Load the first value in R6 and R7
MOV   R6, #62h
MOV   R7, #30h

; Load the second value into R4 and R5
MOV R4, #43h
MOV   R5, #2Eh

; call the 16-bit multiplication routine
LCALL   MUL16_16
```

CHAPTER FIVE

TIMER OPERATION

5.0 Timers

A timer is a series of divide-by-2 flip-flops that receive an input signal as a clocking source. The clock is applied to the first flip-flop which divides the clock frequency by 2. The output of the first flip-flop clocks the second flip-flop, which also divides by 2, and so on. Since each successive stage divides by 2, a timer with n stages divides the input clock frequency by 2^n. The output of the last stage clocks a timer overflow flip-flop, or flag, which is tested by software or generates an interrupt. The binary value in the timer flip-flops can be thought of as a 'count' of the number of clock pulses (or events) since the timer was started. A 16-bit timer, for example, would count from 0000H to FFFFH. The overflow flag is set on the FFFFH to – 0000H overflow of the count.

The operation of a simple timer is illustrated in Figure 5.1 for a 3-bit timer.

Timer flip-flops (3)

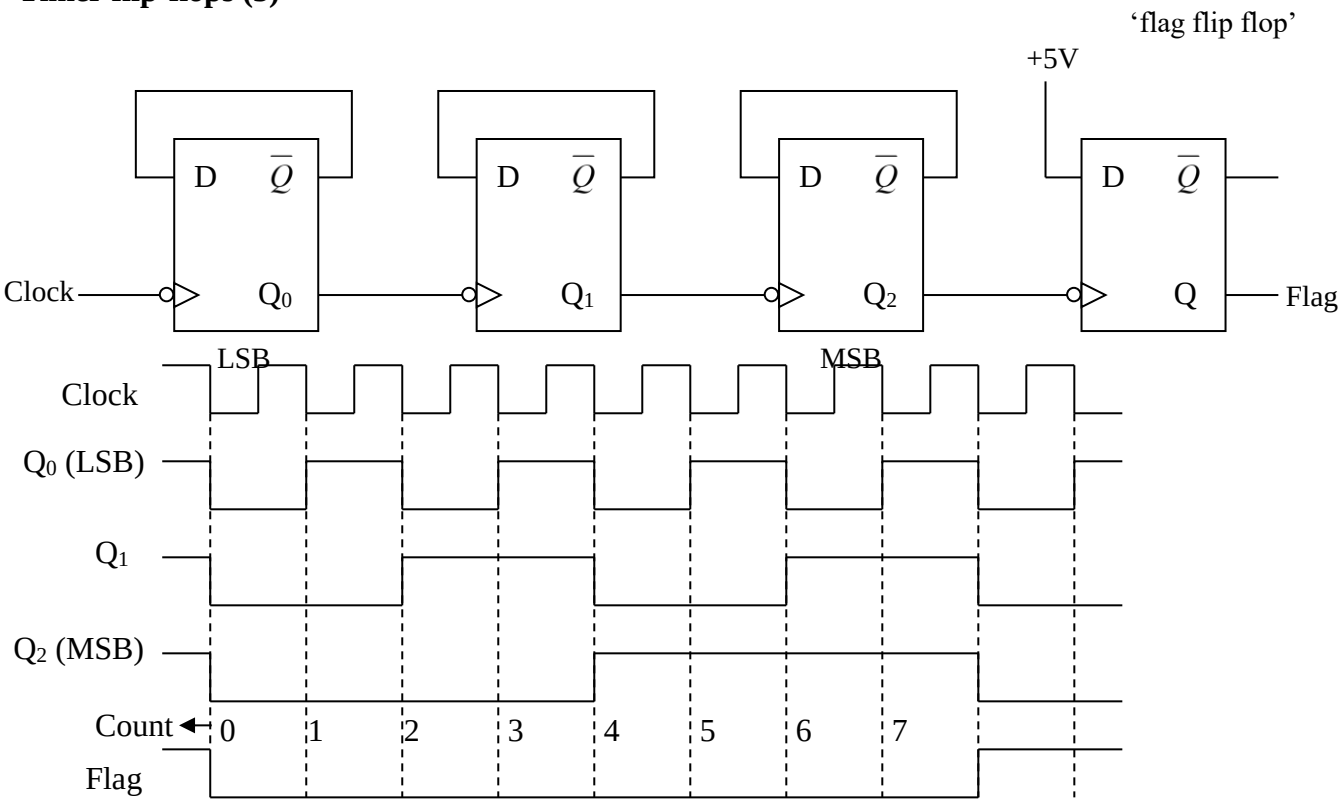


Figure 5.1:3-Bit Timer Operation

Each stage is shown as a type-D negative edge-triggered flip-flop operating in divide-by-2 mode (i.e the $\overline{Q}$ output connects to the D input). The flag flip-flop is simply a type-D latch, set by the

last stage in the timer. It is evident in the timing diagram that the first stage (Q_0) toggles at ½ the clock frequency, the second stage at ¼ the clock frequency and so on. The count is shown in decimal and is easily verified by examining the state of the three flip-flops. For example, the count 4 occurs when $Q_2 = 1$, $Q_1 = 0$, $Q_0 = 0$ ($4_{10} = 100_2$).

Times are used in virtually all control-oriented applications, and the 8051 timers are no exception. There are two 16-bit timers each with four modes of operation (A third 16-bit timer with 3 modes of operation is added on the 8052). The timers are used for

(a) interval timing

(b) event counting or

(c) baud rate generation for the built-in serial port.

Since each is a 16-bit timer, the 16^{th} or last stage divides the input clock frequency by

$$2^{16} = 65536$$

5.1.1 INTERVAL TIMING

In interval timing applications, a timer is programmed to overflow at a regular intervals and set the timer overflow flag. The flag is used to synchronize the program to perform an action such as checking the state of the inputs or sending data to outputs. Other applications can use the regular clocking of the timer to measure the elapsed time between two conditions (e.g. pulse width measurements).

5.1.2 EVENT COUNTING

Event counting is used to determine the number of occurrences of an event. An event is any external stimulus that provides a 1-to-0 transition to a pin on the 8051IC. The timers can also provide the baud rate clock for the 8051s internal serial port.

The 8051 timers are accessed using six special function registers, able 5.1.

TABLE 5.1: TIMER SPECIAL FUNCTION REGISTERS

TIMER SFR	PURPOSE	ADDRESS	BIT ADDRESSABLE
TCON	CONTROL	88H	YES
TMOD	MODE	89H	NO
TL0	TIMER 0 LOW BYTE	8AH	NO
TL1	TIME 1 LOW BYTE	8BH	NO
TH0	TIMER 0 HIGH BYTE	8CH	NO
TH1	TIMER 1 HIGH BYTE	8DH	NO

5.2 TMOD REGISTER (TMOD)

The TMOD register contains two groups of four bits that set the operating mode for Timer 0 and Timer 1 as shown in Table 5.2 and Table 5.3 respectively. TMOD is loaded once by software at the beginning of a program to initialize the timer mode. Thereafter, the timer can be stopped, started and so on by accessing the other timer SFRs.

TABLE 5.2: TMOD REGISTER

BIT	NAME	TIMER	DESCRIPTION
7	GATE	1	Gate bit n when set, timer only runs while $\overline{INT1}$ is high
6	C/$\overline{T}$	1	Counter select bit 1 = event counter 0 = interval timer
5	M1	1	Mode bit 1
4	M0	1	Mode bit 0
3	GATE	0	Timer 0 gate bit
2	C/$\overline{T}$	0	Timer 0 counter/select bit
1	M1	0	Timer 0 M1 bit
0	M0	0	Timer 0 M0 bit

TABLE 5.3: TIMER MODES

M1	M0	MODE	DESCRIPTION
0	0	0	13-bit timer mode (8048 mode)
0	1	1	16-bit timer mode
1	0	2	8-bit auto-reload mode
1	1	3	Split timer mode: Timer 0: TL0 is an 8-bit timer controlled by timer 0 mode bits. TH0 is the same except controlled by timer/mode bits. Timer 1: stopped

5.3 TIMER CONTROL REGISTER (TCON)

The TCON register contains status and control bits for Timer 0 and Timer 1 as shown in the Table 5.4.

Table 5.4: TCON summary

BIT	SYMBOL	BIT ADDRESS	DESCRIPTION
TCON.7	TF1	8FH	Timer 1 overflow flag. Set by hardware upon overflow. Cleared by software or by hardware when processor vectors to interrupt service routines.
TCON.6	TR1	8FH	Timer 1 run-control bit. Set/cleared by software to turn timer on/off
TCON.5	TF0	8DH	Timer 0 overflow flag

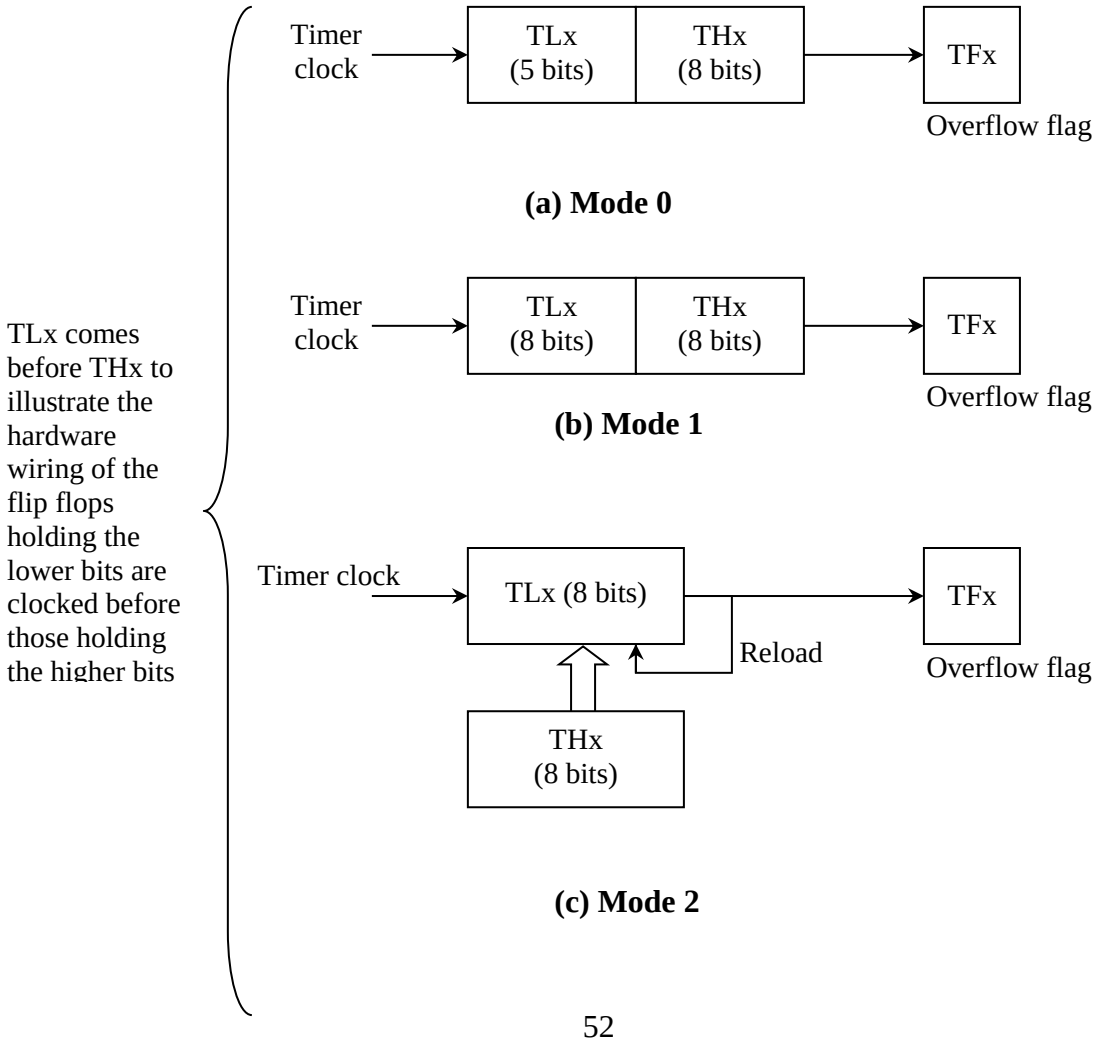
TCON.4	TR0	8CH	Timer 0 run-control bit
TCON.3	1E1	8BH	External interrupt 1 edge flag. Set by hardware when a falling edge is detected on $\overline{INT1}$. Cleared by software or by hardware when CPU vectors to interrupt service routine
TCON.2	1T1	8AH	External interrupt/type flag. Set/cleared by software for falling edge/low-level activated external interrupt.
TCON.1	1E0	89H	External interrupt 0 edge flag
TCON.0	1TO	88H	External interrupt 0 type flag

The upper four bits in TCON (TCON.4 – TCON.7) are used to turn the timers on and off (TR0, TR1), or signal a timer overflow (TF0, TF1).

The lower four bits in TCON (TCON.0 – TCON.3) have nothing to do with the timers. They are used to detect and initiate external interrupts.

5.4 TIMER MODES AND THE OVERFLOW FLAG

Since the 8051 use two timers, 'x' is used to imply either the arrangement of timer registers TLx and THx and the timer overflow flags TFx is shown in Figure 5.2 for each mode.



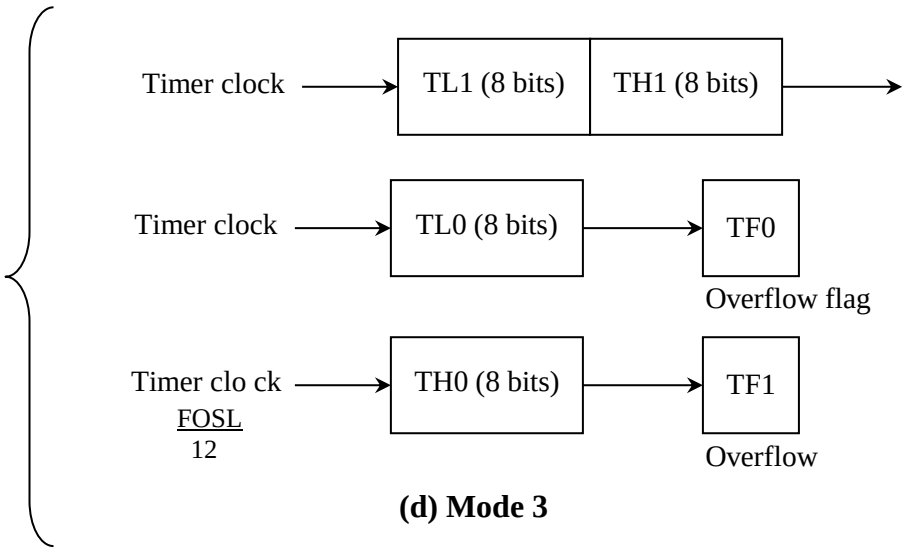


Figure 5.2: Timer registers TLx and THx and the timer overflow flags TFx

5.2.1 13-BIT TIMER MODE (MODE 0)

Mode is a 13-bit timer mode that provides compatibility with the 8051's predecessor, the 8048. It is generally not used in new designs. The timer high-byte (THx) is cascaded with the five least significant bits of the timer low-byte (TLx) to form a 13-bit timer. The upper three bits of TLx are not used.

5.2.2 16-BIT TIMER MODE (MODE 1)

Mode 1 is a 16-bit timer mode (the same as mode 0) except the timer is operating as a full 16-bit timer. The clock is applied to the combined high and low timer registers (TLx/THx). As clock pulses are received, the timer counts up from 0000H, 0001H, 0002H, etc. An overflow occurs on the FFFFH-to-0000H transition of the count and sets the timer overflow flag. The timer continues to count. The overflow flag is the TFx bit in TCON that is read or written by software. The MSB of the value in the timer registers is THx bit 7 while the LSB is TLx bit 0. The LSB toggles at the input clock frequency divided by 2 while the MSB toggles at the input clock frequency divided by 65536 (i.e. 2^{16}). The timer registers may be read or written at any time by software.

5.2.3 8-BIT AUTO-RELOAD MODE (MODE 2)

The timer low-byte (TLx) operates as an 8-bit timer while the timer high-byte (THx) holds a reload value. When the count overflows from FFH not only is the timer flag set, but the value in THx is loaded into TLx. Counting continues from this value up to the next FFH overflow, and so on. This mode is convenient since timer overflows occur at specific periodic intervals once TMOD and THx are initialized. If TLx contains 4FH, for example, the timer counts continuously from 4FH to FFH.

5.2.4 SPLIT TIMER MODE (MODE 3)

Mode 3 is the split timer mode and is different for each timer. Timer 0 in mode 3 is split into two 8-bit timers. TL0 and TH0 act as separate timers with overflows setting the TF0 and TF1 bits respectively.

Timer 1 is stopped in mode 3 but can be started by switching it into one of the other modes. The only limitation is that the usual Timer 1 overflow flag, TF1 is not affected by Timer 1 overflows since it is connected to TH0.

Mode 3 essentially provides an extra 8-bit timer. The 8051 appears to have a third timer. When Timer 0 is in mode 3, Timer 1 can be turned on and off by switching it out of and into its own mode 3. It can still be used by the serial port as a band rate generator, or it can be used in any way not requiring interrupts (since it is no longer connected to TF1).

5.3 CLOCKING SOURCES FOR THE TIMERS

There are two possible clock sources, selected by writing to the counter/timer ($C/\overline{T}$) bit in TMOD when the timer is initialized. One clocking source is used for interval timing, the other for event counting.

5.3.1 INTERVAL TIMING

If $C/\overline{T}$ = 0, continuous timer operation is selected and the timer is clocked from the on-chip oscillator. A divide-by-12 stage is added to reduce the clocking frequency to a value reasonable for most applications.

When continuous timer operation is selected, the timer is used for ‘interval timing’. The timer registers (TLx/THx) increment at a rate of $^{1}/_{12}{}^{th}$ the frequency of the on-chip oscillator, Figure 5.3. Thus, a 12MHz crystal would yield a clock rate of 1MHz. Timer overflows occur after a fixed number of clocks, depending on the initial value loaded into the timer registers (TLx/THx).

5.3.2 EVENT COUNTING

If $C/\overline{T}$ = 1, the timer is clocked from an external source. In most applications, this external source supplies the timer with a pulse upon the occurrence of an ‘event’, the timer is ‘event counting’. The number of events is determined in software by reading the timer registers TLx/THx, since the 16-bit value in these registers increments for each event.

The external clock source comes by way of the alternate functions of the port 3 pins. Port 3 bit 4 (P3.4) serves as the external clocking input for Timer 0 and is known as ‘T0’. Thus P3.5 or T1 is the clocking input for Timer 1.

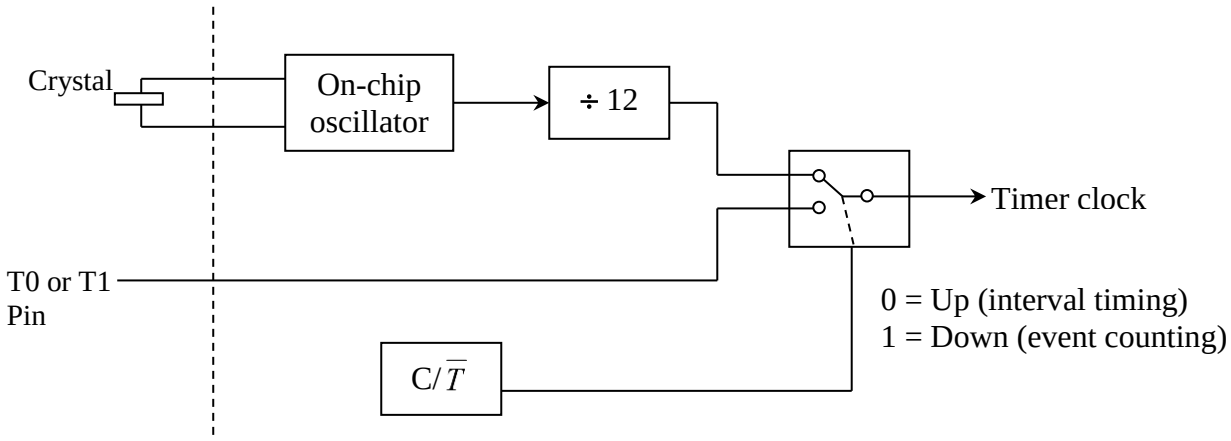


Figure 5.3: Clocking Sources

In counter applications, the timer registers are incremented in response to a 1-to-0 transition at the external input, Tx. The external input is sampled during S5P2 of every cycle (check the SP/machine cycle). Thus, when the input, shows a high in one cycle and a low in the next, the count is incremented. The new value appears in the timer registers during S3P1 of the cycle following the one in which the transition is detected. Since it takes 2 machine cycles (2μs) to recognize a 1-to-0 transition, the maximum external frequency is 500KHz (assuming 12MHz operation).

5.4 STARTING, STOPPING AND CONTROLLING THE TIMERS

The simplest method for starting and stopping the timers is with the run-control bit, TR_x, Figure 5.4, in TCON. TR_x is clear after a system reset; thus the timers are disabled (stopped) by default. TR_x is set by software to start the timers.

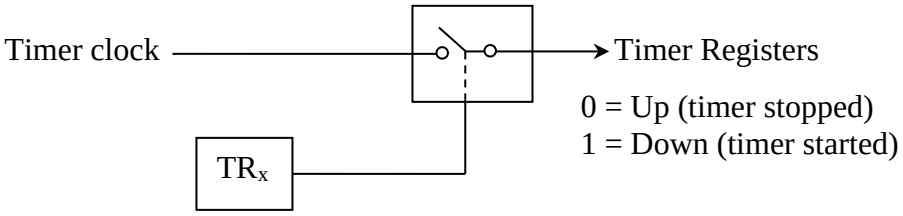


Figure 5.4: Starting and Stopping the timers

Since TRx is in the bit-addressable register TCON, it is easy to start and stop the timers within a program. For example, Timer 0 is started by

```
SETB TR0
```

and stopped by

```
CLR TR0
```

The assembler evaluates the correct bit address. SETB TR0 is exactly the same as SETB 8CH. Another method for controlling the timers is with the GATE bit in TMOD and the external input $\overline{INT_X}$. Setting GATE = 1 allows the timer to be controlled by $\overline{INT_X}$. This is useful for pulse width measurements. Example, assume $\overline{INT0}$ is low but pulses high for a period of time to be measured. Initialize Timer 0 for mode 1, 16-bit timer mode, with TL0/TH0 = 0000H, GATE = 1, and TR0 = 1. When $\overline{INT0}$ goes high, the timer is 'gated on' and is clocked at a rate of 1MHz. When $\overline{INT0}$ goes low, the timer is 'gated off' and the duration of the pulse in microseconds is the count in TL0/TH0. ($\overline{INT0}$ can be programmed to generate an interrupt when it returns low).

The figure below illustrates Timer 1 operating in mode 1 as a 16-bit timer as well as the timer registers TL1/TH1 and the overflow flag TF1. The Figure 5.5 shows the possibilities for the clocking source and for starting, stopping, and controlling the timer.

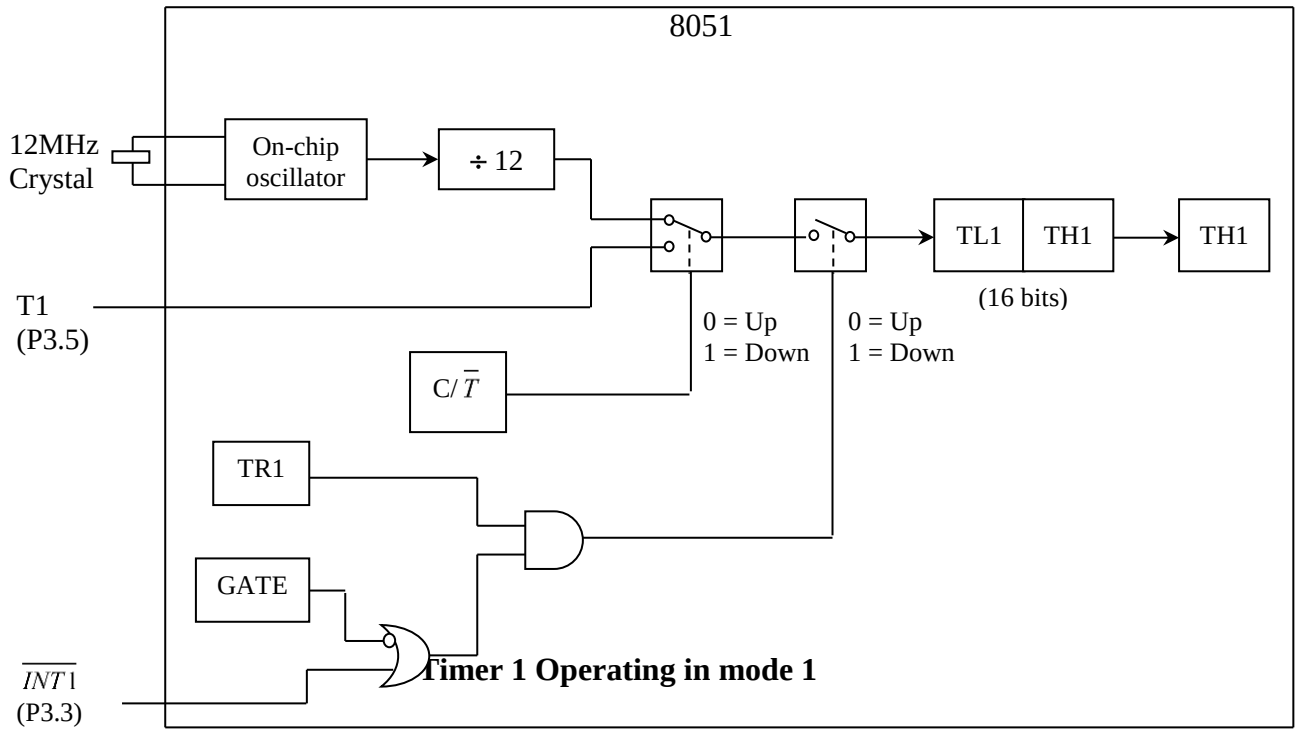


Figure 5.5: Clocking Source, Starting and Stopping and Controlling the Timer

5.5 INITIALIZING AND ACCESSING TIMER REGISTERS

The timers are usually initialized once at the beginning of a program to set the correct operating mode. Thereafter within the body of the program, the timers are started, stopped, flag bits tested and cleared, timer registers read or updated, and so on, as required in the application.

TMOD is the first register initialized, since it sets the mode of operation. For example, the following instruction initializes Timer 1 as a 16-bit timer (mode 1) clocked by the on-chip oscillator (interval timing)

```
MOV TMOD, # 00010000B  OR MOV TMOD, # 10H
```

The effect of this instruction is to set M1 = 0 and M0 = 1 for mode 1, leave C/$\overline{T}$ = 0 and GATE = 0 for internal clocking, and clear the Timer 0 mode bits. The timer does not actually begin timing until its run control bits TR1, is set.

If an initial count is necessary, the timer registers TL1/TH1 must also be initialized. Remembering that the timers count up and set the overflow on an FFFFH-to-0000H transition, a 100μs interval could be timed by initializing TL1/TH1 to 100 counts less than 0000H. The correct value is -100 or FF9CH. The following instructions do the job.

```
MOV TL1, # 9CH
MOV TH1, # 0FFH
```

The timer is then started by setting the run control bit as follows:

```
SETB TR1
```

The overflow flag automatically set 100μs later. Software can sit in a 'wait loop' for 100μs using a conditional branch instruction that returns to itself as long as the overflow flag is not set.

```
Wait: JNB TF1, WAIT
```

When the timer overflows, it is necessary to stop the timer and clear the overflow flag in software:

```
CLR TR1
CLR TF1
```

Example: 10KHz square wave

Write a program using Timer 0 to create a 10KHz square wave on P1.0.

Solution

```
8100            6       ORG     8100H
8100    758902  7       MOV     TMOD, #02H  ;   8-bit auto-reload
8103    758CCE  8       MOV     TH0, #-50   ;   -50 reload value in TH0
8106    D28C    9       SETB    TR0         ;   start timer
```

```
8108    308DFD    10 Loop:  JNB    TF0, LOOP     ;   wait for overflow
810B    C28D      11        CLR    TF0           ;   clear timer overflow flag
810D    B290      12        CPL    P1.0          ;   toggle port bit
810F    80F7      13        SJMP   LOOP          ;   repeat
                  14        END
```

Example

Write a program that creates a square wave on P1.0 with a frequency of 10KHz and a square wave on P2.0 with a frequency of 1KHz.

Solution

```
8100              6         ORG    8100H
8100    758912    7         MOV    TMOD, #12H    ;   timer 1 in mode 1
                  8                               ;   timer 0 in mode 2
8103    758CCE    9         MOV    TH0, #-50     ;   -50 reload value in TH0
                  10                              ;   in TH0
8106    D28C      11        SETB   TR0           ;   start timer 0
8108    758DFE    12  LOOP: MOV    TH1, #0FEH    ;   -500 (high byte)
810B    758B0C    13        MOV    TL1, #0CH     ;   -500 (low byte)
810E    D28E      14        SETB   TR1           ;   start timer 1
8110    308D04    15  WAIT: JNB    TF0, NEXT     ;   timer 0 overflow?
                  16                              ;   no; check timer 1
8113    C28D      17        CLR    TF0           ;   Yes, clear Timer 0
                  18                              ;   overflow flag
8115    B290      19        CPL    P1.0          ;   toggle P1.0
8117    308FF6    20  NEXT: JNB    TF1, WAIT     ;   timer 1 overflow?
                  21                              ;   no: check timer 0
811A    C28E      22        CLR    TR1           ;   stop timer 1
811C    C28F      23        CLR    TF1           ;   clear timer 1
                  24                              ;   overflow flag
811E    BZA0      25        CPL    P2.0          ;   toggle P2.0
8120    80E6      26        SJMP   LOOP          ;   repeat
                  14        END
```

Both timers 0 and 1 are used in this case to simultaneously generate two square waves on P1.0 and P2.0 respectively. The value written into TMOD initializes both timers at the same time. Even though the timers are running simultaneously, the testing for overflow has to be done in sequence. Timer 0 is checked first since its period is smaller in order to avoid running its overflows. Notice that Timer 0 is set to operate in mode 2 auto-reload mode so there is no need to reload the count after ever overflow. Meanwhile, Timer 1 operates in mode 1 so its count must be reloaded every time an overflow occurs.

Example

A buzzer is connected to P1.7 and a debounced switch is connected to P1.6, Figure 5.7.

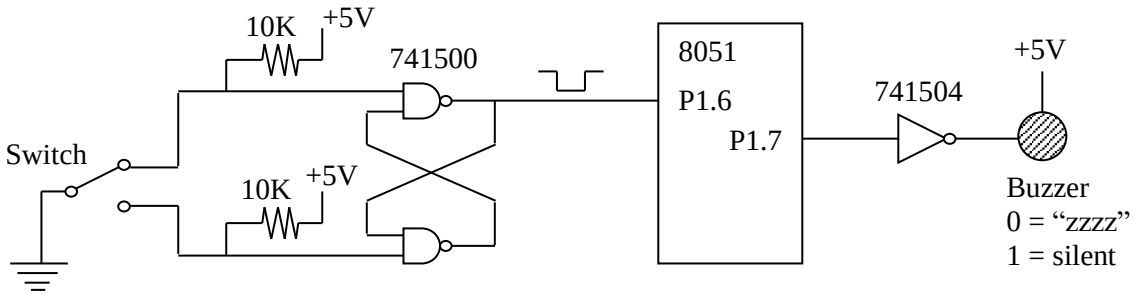


Figure 5.7: The buzzer and debounced switch connection

Write a program that reads the logic level provided by the switch and sounds the buzzer for 1 second for each 1-to-0 transition detected.

Solution

```
0064             6   HUNDRED  EQU   100          ;  100 x 10000 μs = 1 sec
D8F0             7   COUNT    EQU   -10000
8100             8            ORG   8100H
8100  758901      9            MOV   TMOD, #01H  ;  use timer 0 in mode 1
8103  3096FD     10   LOOP:   JNB   P1.6, LOOP   ;  wait for 1 input
8106  2096FD     11   WAIT:   JB    P1.6, WAIT   ;  wait for 0 input
8109  D297       12           SETB  P1.7         ;  turn buzzer on
810B  128112     13           CALL  DELAY        ;  wait 1 second
810E  C297       14           CLR   P1.7         ;  turn buzzer off
8110  80F1       15           SJMP  LOOP
                 16   ;
8112  7F64       17   DELAY:  MOV R7, #HUNDRED
8114  758CD8     18   AGAIN:  MOV TH0, #HIGH COUNT
8117  758AF0     19           MOV TL0, #LOW COUNT
811A  D28C       20           SETB TR0
811C  308DFD     21   WAIT2:  JNB  TF0, WAIT 2
811F  C28D       22           CLR TF0
8121  C28C       23           CLR TF0
8123  DFEF       24           DJNZ R7, AGAIN
8125  22         25           RET
                 26           END
```

5.7 USING TIMERS AS EVENT COUNTERS

The 8051 also allows the use of the timers to count events. Let us say you had a sensor placed across a road that would send a pulse every time a car passed over it. This could be used to

determine the volume of traffic on the road. We could attach this sensor to one of the 8051's I/O lines and constantly monitor it, detecting when it pulsed high and then incrementing our counter when it went back to a low state. Let the sensor be connected to P1.0.

The 8051 counts 1-to-0 transitions on the P3.4 line. This means that when a car first runs over the sensor it will raise the input to a high ("1") condition. At that point the 8051 will not count anything since this is a 0-to-1 transition. However, when the car has passed the sensor will fall back to a low ("0") state. This is a 1-to-0 transition and at that instant the counter will be incremented by 1.

It is important to note that the 8051 checks the P3.4 line each instruction cycle (12 clock cycles). This means that if P3.4 is low, goes high, and goes back low in 6 clock cycles it will probably not be detected by the 8051. This also means the 8051 event counter is only capable of counting events that occur at a maximum of 1/24th the rate of the crystal frequency. That is to say, if the crystal frequency is 12.000 MHz it can count a maximum of 500,000 events per second (12.000 MHz * 1/24 = 500,000). If the event being counted occurs more than 500,000 times per second it will not be able to be accurately counted by the 8051.

CHAPTER SIX

SERIAL PORT OPERATION

6.1 8051 SERIAL PORT

The 8051 includes an on-chip serial port that can operate in several modes over a wide range of frequencies. The essential function of serial port is to perform parallel-to-serial conversion for output data and serial-to-parallel conversion for input data. Hardware access to the serial port is through the TXD and RXD pins. These pins are the alternate functions for two port 3 bits, P3.1 on pin 11 (TXD), and P3.0 on pin 10 (RXD).

The serial port features full duplex operation and receive buffering, allowing one character to be received and held in a buffer while a second character is received. If the CPU reads the first character before the second is fully received, data are not lost.

The serial port frequency of operation, or band rate can be fixed (derived from the 8051 on-chip oscillator) or variable. If a variable band rate is used, Timer 1 supplies the band rate clock and must be programmed accordingly.

Two special function registers, the serial port buffer register (SBUF) and the serial port control register (SCON), provide software access to the serial port.

6.2 SERIAL PORT BUFFER REGISTER (SBUF)

The serial port buffer register (SBUF) at address 99H is really two buffers. Writing to SBUF loads data to be transmitted and reading SBUF accesses received data. These are two separate and distinct registers, the transmit write-only register, and the receive read-only register, Figure 6.1.

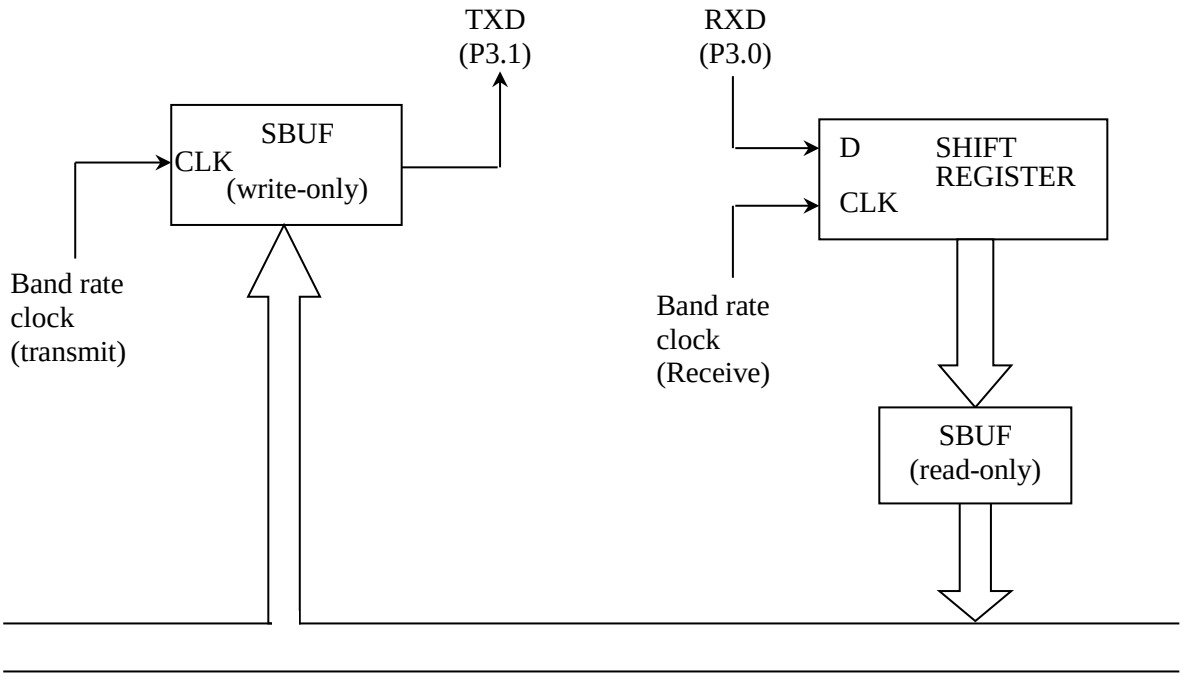


Figure 6.1: Serial Port block diagram

Note in the figure that a serial-to-parallel shift register is used to clock in the received data before it is transferred to the receive read-only register. This shift register is the key elements in providing receive buffering. Only when all 8 bits of the incoming data are received will they be transferred to the receive read-only register. This ensures that while the incoming data and being received, the previous received data are still intact in the receive read-only register.

6.3 SERIAL PORT CONTROL REGISTER (SCON)

SCON.7	SCON.6	SCON.5	SCON.4	SCON.3	SCON.2	SCON.1	SCON.0
SM0	SM1	SM2	REN	TB8	RB8	T1	R1

Figure 6.2: SCON Register

The serial port control register (SCON) at address 98H is a bit-addressable register containing status bits and control bits, Figure 6.2. Status bits indicate the end of a character transmission or reception and are tested in software or programmed to cause an interrupt. Meanwhile, writing to the control bits would set the operating mode for the 8051 serial port, Table 6.1 and table 6.2.

Table 6.1: 8051 SCON SFR

SCON Summary

Bit	Symbol	Bit Address	Description
SCON.7	SMO	9FH	Serial port mode bit 0
SCON.6	SM1	9EH	Serial port mode bit 1
SCON.5	SM2	9DH	Serial port mode bit 2 Enables multiprocessor communication in modes 2 & 3; R1 will not be activated if received 9^{th} bit is 0
SCON.4	REN	9CH	Receiver enable. Must be set to receive characters
SCON.3	TB8	9BH	Transmit bit 8, 9^{th} bit transmitted in modes and 3; set 1 cleared by software
SCON.2	RB8	9AH	Receive bit 8, 9^{th} bit received
SCON.1	T1	99H	Transmit interrupt flag: set at end of character transmission; cleared by software
SCON.0	R1	98H	Receive interrupt flag. Set at end of character reception; cleared by software.

Table 6.2: 8051 Serial Port Modes

SM0	SM1	MODE	DESCRIPTION	BAUD RATE
0	0	0	Shift register	Fixed (oscillator frequency ÷12)
0	1	1	8-bit UART	Variable (set by timer)
1	0	2	9-bit UART	Fixed (oscillator frequency ÷ 12 or ÷ 64)
1	1	3	9-bit UART	Variable (set by timer)

The band rate is fixed at $^{1}/_{12}{}^{th}$ the on-chip oscillator frequency. The RXD line is used for both data input and output and the TXD line serves as the clock.

Transmission is initialized by any instruction that writes data to SBUF. Data are shifted out on the RXD line (P3.0) with clock pulses sent out the TXD (P3.1) line. Each transmitted bit

is valid on the RXD pin for one machine cycle. During each machine cycle, the clock signal goes low on S3P1 and returns high on S6P1. The timing for output data is shown in Figure 6.3.

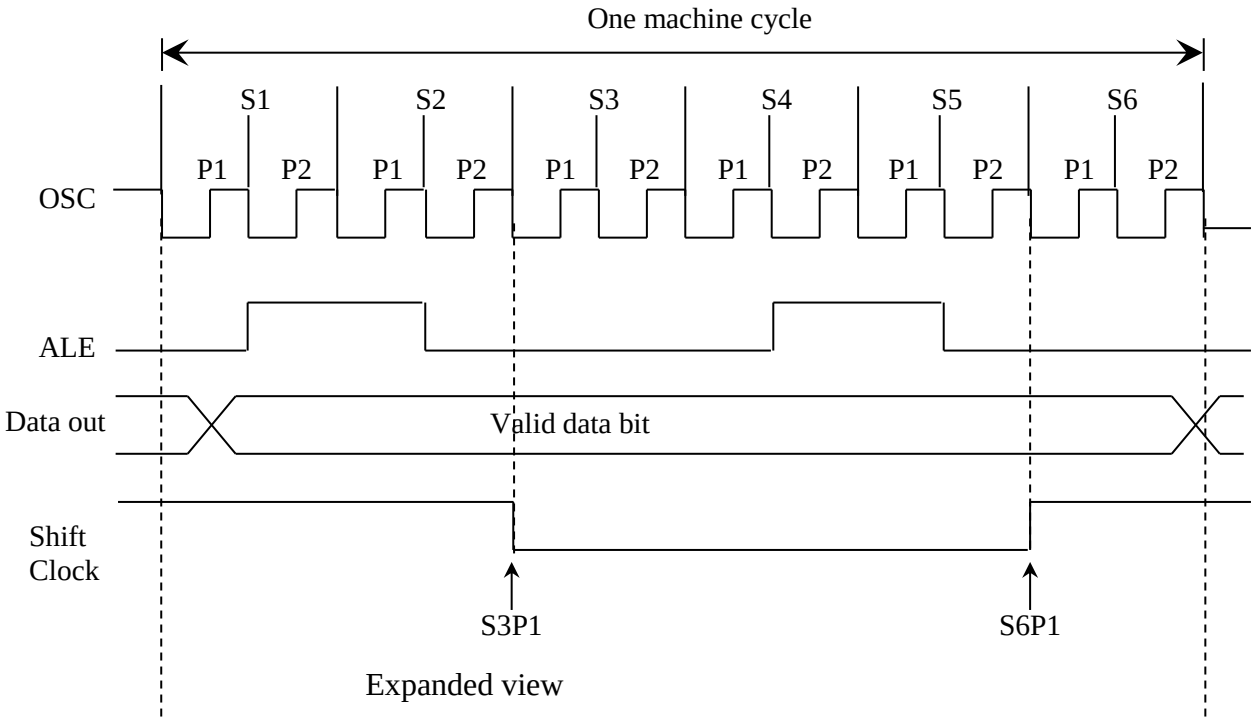


Figure 6.3: Serial port transmit timing for mode 0 (1 bit transmission)

Reception is initiated when the receiver enable bit (REN) is 1 and the receive interrupt bit (R1) is 0. The general rule is to set REN at the beginning of a program to initialize the serial port and then clear R1 to begin a data input operation. When R1 is cleared, clock pulses are written out the TXD line, beginning the following machine cycle, and data are clocked in the RXD line. Obviously, it is up to the attached circuitry to provide data on the RXD line as synchronized by the clock signal on TXD. The clocking of data into the serial port occurs on the positive edge of TXD. Notice that in this mode of operation, the data transfers between the 8051 and the attached circuitry are via synchronous communication where both parties are synchronized to the clock signal on TXD.

6.4 MODE 0 (8-bit Shift Register)

In this mode, serial data enter and exit through RXD and TXD outputs the shift clock. Eight bits are transmitted or received with the least significant bit (LSB) first. The band rate is fixed at $1/12^{th}$ the on-chip oscillator frequency. The RXD line is used for both data input and output, and the TXD line serves as the clock.

Transmission is initiated by any instruction that writes data to SBUF. Data are shifted out on the RxD line with clock pulses sent out the TxD line. Each transmitted bit is valid on the RxD line/pin for one machine cycle.

Reception is initiated when the receiver enable bit (REN) is 1 and the receiver interrupt bit (R1) is 0. The general rule is to set REN at the beginning of a program to initialize the serial port and then clear R1 to begin a data input operation. When R1 is cleared, clock pulses are written out the TXD line, beginning the following machine cycle, and data are clocked in the RXD line. It is up to the attached circuitry to provide data on the RXD line as synchronized by the clock signal on TXD. The clocking of data into the serial port occurs on the positive edge of TXD. The transfer of data between the 8051 and the attached circuitry is done via synchronous communication where both parties are synchronized to the clock signal on TXD.

One possible application of shift register mode is to expand the output capability of the 8051. a serial-to-parallel shift register IC can be connected to the 8051 TXD and RXD lines to provide an extra eight output lines. Additional shift registers may be cascaded to the first for further expansion.

6.5 8-BIT UART WITH VARIABLE BAND RATE (MODE 1)

A UART (Universal Asynchronous Receiver/Transmitter) is a device that receives and transmits serial data with each data character preceded by a start bit (low) and followed by a stop bit (high). A parity bit is sometimes inserted between the last data and the stop bit. In mode 1, 10 bits are transmitted on TXD or received on RXD. These consist of a start bit (always D) eight data bits (LSB first), and a stop bit (always 1). For a receive operation, the stop bit goes into RB8 in SCON. In 8051, the band rate is set by the Timer 1 overflow rate.

In mode 1, the 8051 serial port operates as an 8-bit UART with variable band rate. A UART, or ‘universal asynchronous receiver/transmitter’, is a device that receives and transmits serial data with each data character preceded by a start bit (low) and followed by a stop bit (high). A parity bit is sometimes inserted between the last bit and the stop bit. The essential operation of a UART is parallel-to-serial conversion of output data and serial-to-parallel conversion of input data.

In mode 1, 10 bits are transmitted on TXD or received on RXD. These consist of a start bit (always 0), eight data bits (LSB first), and a stop bit (always 1), for a receive operation, the stop bit goes into RB8 in SCON. In the 8051, the band rate is set by the Timer 1 overflow rate.

Clocking and synchronizing the serial port shift registers in modes 1, 2 and 3 is established by a 4-bit divide-by-16 counter, the output of which is the band rate clock, Figure 6.4.

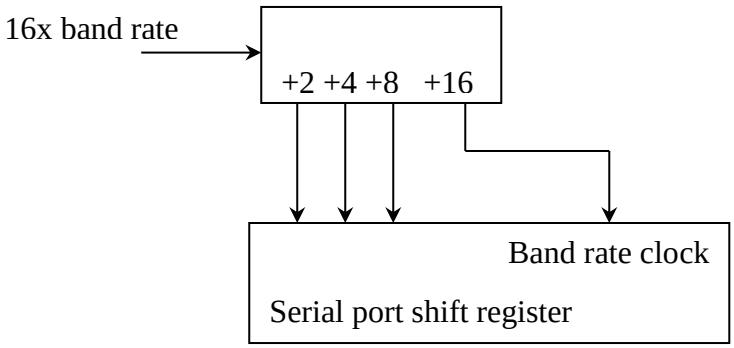


Figure 6.4: Serial port clocking

Transmission is initiated by writing to SBUF but does not actually start until the next rollover of the divide-by-16 counter supplying the serial port band rate. Shifted data are outputted on the TXD line beginning with the start bit followed by the eight data bits, then the stop bit. The period for each bit is the reciprocal of the band rate as programmed in the timer. The transmit interrupt flag (T1) is set as soon as the stop bit appears on TXD.

Reception is initiated by R 1-to-0 transition on RXD. The divide-by-16 counter is immediately reset to align the counts with the incoming bit stream (the next bit arrives on the next divide-by-16 rollover, and so on). The incoming bit stream is sampled in the middle of the 16 counts.

The receiver includes ‘false start bit detection by requiring a 0 state eight counts after the first 1-to-0 transition. If this does not occur, it is assumed that the receiver was triggered by noise rather than by a valid character. The receiver is reset and returns to the idle state, looking for the next 1-to-0 transition.

Assuming a valid start bit was detected, character reception continues. The start bit skipped and eight data bits are clocked into the serial port shift register. When all eight bits have been clocked in, the following occurs.

1. The ninth bit (stop bit) is clocked into RB8 in SCON.
2. SBUF is loaded with the eight data bits and
3. The receiver interrupt flag (R1) is set.

These only occur, however, if the following conditions exist:

1. R1 = 0 and
2. SM2 = 1 and the received stop bit = 1 or SM2 = 0

The requirement that R1 = 0 ensures that software has read the previous character (and cleared R1). The second condition sounds complicated but applies only in multiprocessor communication mode. It implies 'Do not set R1 in multiprocessor communications mode when the ninth data bit is 0'.

6.6 9-BIT UART WITH FIXED BAND RATE (MODE 2)

When SM1 = 1 and SM0 = 0, the serial port operates in mode 2 as a 9-bit UART with a fixed band rate. Eleven bits are transmitted or received: a start bit, eight data bits, a programmable ninth data bit, and a stop bit. On transmission, the ninth bit is whatever has been put in TB8 in SCON (perhaps a parity bit). On reception, the ninth bit received is placed in RB8. The band rate in mode 2 is either $1/32^{nd}$ or $1/64^{th}$ the on-chip oscillator frequency.

6.7 9-BIT UART WTH VARIABLE BAND RATE (MODE 3)

Mode 3, 9-bit UART with variable band rate, is the same as mode 3 except the band rate is programmable and provided by the timer. In fact, modes 1, 2 and 3 are very similar. The difference lie in the band rates (fixed in mode 2, variable in modes 1 and 3) and in the number of data bits (eight in mode 1, nine in mode 2 and 3).

6.8 CLOCKING AND SNYCHRONIZING SERIAL PORT SHIFT REGISTER

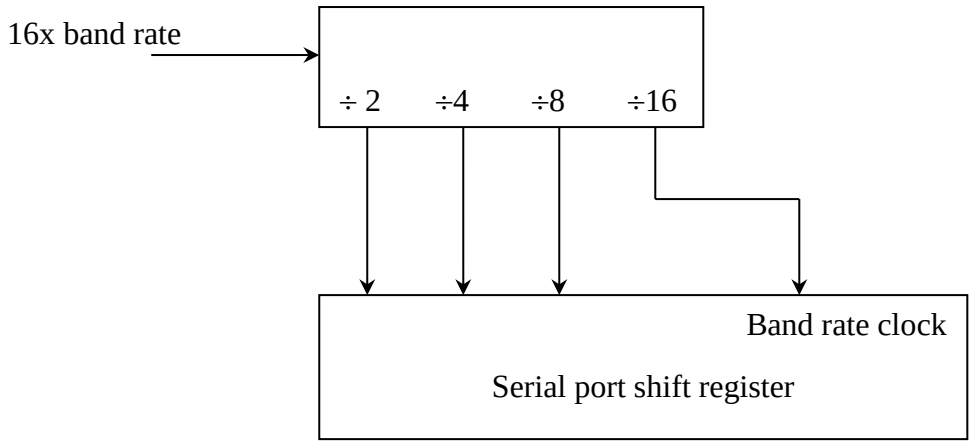


Figure 6.5: Clocking Synchronising serial port shift register

Clocking and synchronizing the serial port shift registers in mode 1, 2, and 3 is established by a 4-bit divide-by-16 counter, the output of which is the band rate clock, Figure 6.5. The input to this counter is selected through software.

Transmission is initiated by writing to SBUF but does not actually start until the next rollover of the divide-by-16 counter supplying the serial port band rate. Shifted data are outputted on the TXD line beginning with the start bit followed by the eight data bits, then the stop bit. The period for each bit is the reciprocal of the band rate as programmed in the timer. The transmit interrupt flag (T1) is set as soon as the stop bit appears on TXD.

Reception is initiated by 1-to-0 transition on RXD. The divide-by-16 counter is immediately reset to align the counts with the incoming bit stream (the next bit arrives on the next divide-by-16 rollover, and so on).

The receiver includes 'false start bit detection' by requiring a 0 state eight counts after the first 1-to-0 transition. If this does not occur, it is assumed that the receiver was triggered by noise rather than by a valid character. The receiver is reset and returns to the idle state, looking for the next 1-to-0 transition.

Assuming a valid start bit was detected, character reception continues. The start bit skipped and eight data bits are clocked into the serial port shift register. When all eight bits have been clocked in, the following take place.

1. The ninth bit (the stop bit) is clocked into RB8 in SCON.
2. SBUF is loaded with the eight data bits and
3. The receiver interrupt flag (R1) is set.

These only occur, however if the following conditions exist:

1. R1 = 0 and
2. SM2 = 1 and the received stop bit = 1, or SM2 = 0

The requirement that R1 = 0 ensures that software has read the previous character and cleared R1. The second condition applies only in multiprocessor communication mode. It means R1 should not be set in multiprocessor communications mode when the ninth data bit is 0.

6.9 INITIALIZING AND ACCESSING SERIAL PORT REGISTERS

(I) RECEIVER ENABLE

The receiver enable bit (REN) in SCON must be set by software to enable the reception of characters. This is usually done at the beginning of a program when the serial port, timers, etc are initialized. This can be done in 2 ways;

1. SETB REN; explicitly sets REN

2. MOV SCON, #xxx1xxxxB; sets REN and sets or clears the other bits in SCON.

(II) The Ninth (9th) Data Bit

The 9th data bit transmitted in modes 2 and 3 must be loaded into TB8 by software. The 9th data bit received is placed in RB8. Software may or may not require a ninth data bit, depending on the specifications of the serial device with which communications are established.

(III) Adding a parity bit

A common use for the ninth data bit is to add parity to a character. The parity bit, P, in the PSW is set or cleared every machine cycle to establish even parity with the eight bits in the accumulator. E.g. if a communication require data bits plus even parity, the following instructions could be used to transmit the eight bits in the accumulator with even parity added in the ninth bit.

```
MOV C, P        ;  Put even parity bit in TB8
MOV TB8, C      ;  This becomes the 9th data bit
MOV SBUF, A     ;  Move 8 bits from accumulator to SBUF
```

If odd parity is required, then the instructions must be modified as follows:

```
MOV C, P        ;  Put even parity bit in C flag
CPL  C          ;  Convert to odd parity
MOV TB8, C
MOV SBUF, A
```

In mode 1 where a ninth data bit is not used, the eight data bits transmitted can consist of seven data bits plus a parity bit. In order to transmit a 7-bit ASCII code with even parity in bit 8, the following instructions could be used.

```
CLR  ACC. 7     ;  Ensure MSB is clear
                ;  Even parity is in P
MOV C, P        ;  Copy to C
MOV ACC. 7, C   ;  Put even parity into MSB
MOV SBUF, A     ;  Send character (7 data bits plus even parity)
```

6.10 INTERRUPT FLAGS

The receive and transmit interrupt flags (R1 and T1) in SCON play an important role in 8051 serial communications. Both bits are set by hardware but must be cleared by software.

R1 is set at the end of character receive to indicate that the buffer is full. This condition is tested in software or programmed to cause an interrupt. If software wishes to input a character from the

device connected to the serial port (e.g. a VDU), it must wait until R1 is set, then clear R1 and read the character from SBUF. This is shown below:

```
WAIT :    JNB R1, WAIT      ;  Check R1 unit set
          CLR R1            ;  Clear R1
          MOV A, SBUF       ;  Read character
```

T1 is set at the end of character transmission indicating that the buffer is empty. If software wishes to send a character to the device connected to a serial port, it must first check that the serial port is ready. This means that if a previous character was sent, wait until transmission is finished before sending the next character. The following code transmits the character in the accumulator:

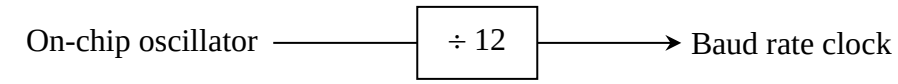
```
WAIT :    JNB   T1, WAIT    ;  Check T1 until set
          CLR   T1          ;  Clear T1
          MOV SBUF, A       ;  Send character
```

The receiver and transmit instruction sequences above are usually part of standard input character and output character subroutine.

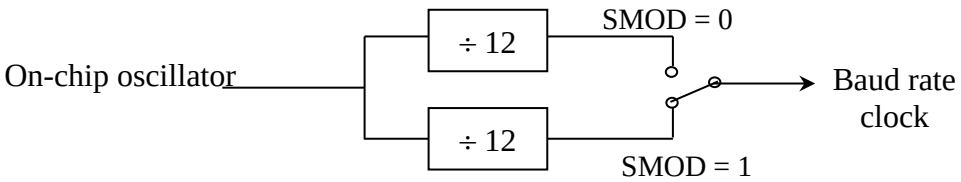
6.11 SERIAL PORT BAUD RATES

Baud rates for modes 0 and 3 are fixed. In mode 0, it is always the on-chip oscillator frequency divide by 12. Usually a crystal drives the 8051's on-chip oscillator, but another clock source can be used as well. Assuming a nominal oscillator frequency of 12MHz, the mode 0 baud rate is 1MHz.

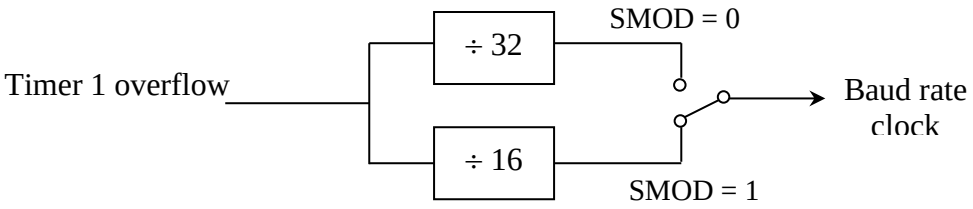
By default following a system reset, the mode 2 band rate is the oscillator frequency divided by 64. The baud rate is also affected by a bit in the power control register, PCON, Figure 6.6.



(a) Mode 0



(b) Mode 2



(c) Mode 1 and 3

Figure 6.6: Serial port clocking sources: (a) Mode 0 (b) Mode 2 (c) Modes 1 and 3

Bit 7 of PCON is the SMOD bit. Setting SMOD has the effect of doubling the baud rate in modes 1, 2 and 3. In mode 2, the baud rate can be doubled from a default value of $1/64^{th}$ the oscillating frequency (SMOD = 0) to $1/32^{nd}$ the oscillating frequency (SMOD = 1).

Since PCON is not bit addressable, setting SMOD without altering the other PCON bits requires a read-modifying-write' operation. The following instructions set SMOD:

```
MOV A, PCON        ;  Get current value of PCON
SETB ACC. 7        ;  Set Bit 7 (SMOD)
MOV PCON. A        ;  Write value back to PCON
```

Baud rates in modes 1 and 3 are determined by the Timer 1 overflow rate. Since the timer operates at a relatively high frequency, the overflow is further divided by 32 (16 if SMOD = 1) before providing the band rate clock to the serial port.

6.12 USING TIMER 1 AS THE BAUD RATE CLOCK

For the 8051, the usual technique for baud rate generation is to initialize TMOD for 8-bit auto-reload mode and put the correct reload value in TH1 to yield the proper overflow rate for the baud rate.

```
MOV TMOD, #0010xxxxB
```

Very low baud rates can be achieved by using 16-bit mode, timer mode 1 with

TMOD = 0001xxxxB

There is a slight overhead since the TH1/TL1 registers must be reinitialized after each overflow. This would be performed in an interrupt service routine.

Another option is to clock Timer 1 externally using T1 (P3.5). regardless the baud rate is the Timer 1 overflow rate divided by 32 (or divided by 16, if SMOD = 1).

The formula for determining the baud rate in modes 1 and 3 therefore is

Baud rate = Timer 1 overflow rate ÷ 3

e.g. 1200 band operation requires an overflow rate calculated as follows:

1200 = Timer 1 overflow rate ÷ 32

Timer 1 overflow rate = 1200 x 32 = 38.4KHz

If a 12MHz crystal drives the on-chip oscillator, Timer 1 is clocked at a rate of 1MHz or 1000KHz. Since the timer must overflow at a rate of 38.4KHz and the timer is clocked at a rate of 1000KHz, an overflow is required every

$$\frac{1000}{38.4} = 26.04 \text{ clocks.} \approx 26 \text{ clocks}$$

Since the timer counts up and overflows on the FF-to-00H transition of the count, 26 counts less than 0 is the required reload value for TH1, or -26. The instruction is

MOV TH1, #-26

OR MOV TH1, #0E6H

Due to rounding, there is a slight error in the resulting band rate. Generally a 5% error is tolerable using asynchronous (start/stop) communications. Exact baud rates are possible using an 11.059MHz crystal.

Most common baud rates using a 12.000MHz or 11.059MHz crystal are shown in Table 6.3.

Table 6.3: Most common baud rates using a 12.000MHz or 11.059MHz crystal

Baud rate	Crystal Frequency	SMOD	TH1 Reload Value	Actual Baud Rate	Error
9600	12.000MHz	1	-7(F9H)	8923	7%
2400	12.000MHz	0	-13(F3H)	2404	0.16%
1200	12.000MHz	0	-26(E6H)	1202	0.16%
19200	11.059MHz	1	-3(FDH)	19200	0
9600	11.059MHz	0	-3(FDH	9600	0
2400	11.059MHz	0	-12(F4H)	2400	0
1200	11.059MHz	0	-24(F8H)	1200	0

Example

Write an instruction sequence to initialize the serial port to operate as an 8-bit UART at 2400 baud. Use Timer 1 to provide the baud rate clock.

Solution

Four registers must be initialized for this example. They are SMOD, TMOD, TCON and TH1. The required values are summarized below.

	SM0	SM1	SM2	REN	TB8	RB8	T1	R1
SCON :	0	1	0	1	0	0	1	0
	GTE	$C/\overline{T}$	M1	M0	GTE	$C/\overline{T}$	M1	M0
TMOD :	0	0	1	0	0	0	0	0
	TF1	TR1	TF0	TR0	1E1	1T1	1E0	1T0
TCON :	0	1	0	0	0	0	0	0
TH1 :	1	1	1	1	0	0	1	1

Setting SM0/SM1 = 0/1 puts the serial port into 8-bit UART mode. REN = 1 enables the serial port to receive characters. Setting T1 = 1 allows transmission of the first character by indicating that the transmit buffer is empty.
For TMOD, setting M1/M0 = 1/0 puts Timer 1 in 8-bit auto-reload mode. Setting TR1 = 1 in TCON turns on Timer 1.

The required TH1 value is that which overflows at the rate of 2400 x 32 = 768KHz. Assuming the 8051 is clocked from a 12MHz crystal, Timer 1 is clocked at a rate of 1MHz or 1000KHz, and the number of clocks for each overflow is

$$\frac{1000}{76.8} = 13.02 \ (\approx 13)$$

The reload value is therefore -13 or 0F3H.
The instruction sequence is as shown below.

```
8100             5          ORG   8100H
8100  759852     6   INIT:  MOV   SCON, #52H   ;  serial port mode 1
8103  758920     7          MOV   TMOD, #20H   ;  timer 1, mode 2
8106  758DF3     8          MOV   TH1, #-13    ;  reload count for 2400 band
8109  D28E       9          SETB  TR1          ;  start timer 1
                 10         END
```

Example: Output character subroutine
Write a subroutine called OUTCHR to transmit the 7-bit ASCII code in the accumulator out the 8051 serial port, with odd parity added as the eighth bit. Return from the subroutine with the accumulator intact, i.e. containing the same value as before the subroutine was called.

Solution

```
8100             5          ORG   8100H
8100  A2D0       6   OUTCHR; MOV   C, P        ;  put parity bit in C flag
8102  B3         7          CPL   C            ;  change to odd parity
8103  92E7       8          MOV   ACC.7, C     ;  add to character code
8105  3099ED     9   AGAIN: JNB   T1, AGAIN    ;  Tx empty? No; check again
8108  C299       10         CLR   T1           ;  Yes, clear flag and
810A  F599       11         MOV   SBUF, A      ;  send character
810C  C2E7       12         CLR   ACC.7        ;  strip off parity bit and
810E  22         13         RET                ;  return
                 14         END
```

The first 3 instructions place odd parity in the accumulator bit 7. since the P bit in the PSW establishes even parity with the accumulator, it is complemented before being placed in ACC.7. The JNB instruction creates a ‘wait loop’ repeatedly testing the transmit interrupt flag (T1) until

it is set. When T1 is set, it is cleared and then the character in the accumulator is written into the serial port buffer (SBUF). Transmission begins on the next rollover of the divide-by-16 counter that clocks the serial ports. Finally the ACC.7 is cleared so that the return value is the same as the 7-bit code passed to the subroutine.

The OUTUAR is a building block and is of little use by itself. At a higher level, this subroutine is called to transmit a single character or a string of characters. E.g. the following instructions transmit the ASCII code for the letter 'Z' to the serial device attached to the 8051's serial port.

```
MOV   A, # 'Z'
CALL  OUTCHR
  :
  :
```

Example

Write a subroutine called INCHR to input a character from the 8051's serial port and return with the 7-bit ASCII code in the accumulator. Expect odd parity in the eight bit received and set the carry flag if there is a parity error.

Solution

```
8100                5     INCHR:  ORG   8100H
8100  3098FD        6             JNB   R1, $        ;  wait for character
8103  C298          7             CLR   R1           ;  clear flag
8105  E599          8             MOV   A, SBUF      ;  read character into A
8107  A2D0          9             MOV   C, P         ;  for odd parity in A
                                                     ;  P should be set
8109  B3           11             CPL   C            ;  complementing correctly
                   12                                ;  indicate if "error"
810A  C2E7         13             CLR   ACC.7        ;  strip off parity
810C  22           14             RET
                   15             END
```

The subroutine begins by waiting for the receive interrupt flag (R1) to be set, indicating that a character is waiting in SBUF to be read. When R1 is set, the JNB instruction falls through to the next instruction. R1 is cleared and the code in SBUF is read into the accumulator. The P bit in the PSW establishes even parity with the accumulator, so it should be set if the accumulator, on its own correctly contains odd parity in bit 7. Moving the P bit into the carry flag and complementing it leaves CY = 0 if there is no error. On the other hand, if the accumulator contains a parity error, then CY = 1, correctly indicating "parity error". Finally ACC.7 is cleared to ensure that only a 7-bit code is returned to the calling program.

Example: Full duplex operation

Write a program that continually transmits characters from a transmit buffer (internal RAM 30H to 4FH). If incoming characters are detected on the serial port, store them in the receive buffer starting at internal RAM location 50H. Assume that the 8051 serial port has already been initialized in mode 1.

Solution

```
8100            1           ORG   8100H
8100  7830       2           MOV   R0, #30H      ;  pointer for Tx buffer
8102  7950       3           MOV   R1, #50H      ;  pointer for Rx buffer
8104  209819     4     LOOP: JB    R1, RECEIVE   ;  character received?
                  5                               ;  yes, process it
8107  209902     6           JB    T1, TX        ;  previous character
                  7                               ;  transmitted
                  8                               ;  yes, process it
810A  80F8       9           SJMP  LOOP          ;  no, continue checking
810C  E6         10    TX:   MOV   A, @R0        ;  get character from Tx
                  11                              ;  buffer
810D  A2D0       12          MOV   C, P          ;  put parity bit in C
810F  B3         13          CPL   C             ;  change to odd parity
8110  92E7       14          MOV   ACC.7, C      ;  add to character code
8112  C299       15          CLR   T1            ;  clear transmit flag
8114  F599       16          MOV   SBUF, A       ;  send character
8116  C2E7       17          CLR   ACC.7         ;  strip off parity bit
8118  08         18          INC   R0            ;  point to next cha-
                  19                              ;  racter in buffer
8119  B850E8     20          CJNE  R0, #50H      ;  loop; end of buffer'
                  21                              ;  no, continue
811C  7830       22          MOV   R0, #30H      ;  yes, recycle
811E  80E4       23          SJMP  LOOP          ;  continue checking
                  24
8120  C298       25    RX:   CLR   R1            ;  checking receive flag
8122  E599       26          MOV   A, SBUF       ;  read character into A
8124  A2D0       27          MOV   C, P          ;  for odd parity in A,
                  28                              ;  P should be set
8126  B3         29          CPL   C             ;  complementing correctly
                  30                              ;  indicates 'error'
                  31
8127  C2E7       32          CLR   ACC.7         ;  strip off parity
8129  F7         33          MOV @ R1, A         ;  store received character
                  34                              ;   in buffer
812A  09         35          INC   R1            ;  point to next location
                  36                              ;  in buffer
812B  80D7       37          SJMP LOOP           ;  continue checking
                  38          END
```

The program first uses two registers R0 and R1 to point to the transmit and receive buffers respectively. It then checks to see if a character has been received or if the previous character to be transmitted has already been sent out. Notice that reception is processed first. This is because reception is more critical since the 8051 is depending on an external device for an incoming character, which must be processed and stored as soon as possible else it might be overwritten by subsequent incoming characters.

Recall that the serial port can hold at most one character in its buffer while a second as being received. If an incoming character has been fully received, it is read from SBUF and checked for parity before being stored into the receiver buffer where current empty location is pointed to by R1. R1 is next incremented to point to the next available location and the program goes back to checking.

When the previous transmission is finished, the program first gets the character to be transmitted from the transmit buffer. R0 is used as the pointer to the next character in the transmit buffer. The odd parity is then placed in bit 7 of the code before it is set to the SBUF for transmission. R1 is then incremented to point to the next character. The program also checks for the end of the transmit buffer, upon which it would recycle back to the beginning of the buffer. Finally, the program goes back to checking for buffer receptions or transmission.

Example

Write a code to move the ASCII character stored in location 32H to the accumulator and check for even parity.

Solution

```
MOV   A, 32H;    moves ASCII character stored at 32H to the ACC
MOV   C, P             ;   copy the parity bit to the carry bit
MOV   ACC.7, C         ;   move this bit to the ACC MSB
MOV   SBUF, A          ;   transmit character with parity bit down the serial line
JNB   T1, $            ;   wait for the byte to be sent
CLR   T1
  :
```

The receiver must be able to count the number of ones in the received data. The code below does that.

```
Wait for Byte:
        JNB  R1, $                  ;  wait for a byte to be received
        CLR    R1
```

```
MOV   A, SBUF              ;  move byte to the accumulator
JB    P, wait for byte     ;  if the parity bit is set, there is
                           ;  transmission error
CLR   ACC.7                ;  valid data, clear the parity bit
```

The code waits for a byte to be received. Once a byte arrives, it moves it into the accumulator. If the number of ones in the accumulator in odd then the parity bit in the PSW will be set. Therefore, the program tests the status of the parity bit. If it is 1, it means a bit in the transmitted data was altered. This should be the case since the transmitter included a parity bit in the MSB of the transmitted byte to ensure the number of ones was even. If the receiver sees an odd number of ones, then something went wrong.

Example

Write a program to send the text abcdef down the serial port line at a band rate of 9600 with a system clock frequency of 12MHz.

Solution

```
CLR   SM0
SETB  SM1    ;  clear SM0 and set SM1 to put the serial port in mode 1
MOV   TMOD, #20H    ;   put Timer 1 in mode 2 interval timing
MOV   TH1, #253     ;   put the reload value in TH1 to produce 9600 band rate
SETB  TR1    ;   start timer 1
MOV   40H, #'a'
MOV   41H, #'b'
MOV   42H, #'c'
MOV   43H, #'d'
MOV   44H, #'e'
MOV   45H, #'f'     ;   store the text in memory starting at address 40F
MOV   46H, #0       ;   terminate the text with 0
MOV   R0, #40H      ;   put the start address in R0
CALL    SendText
```

This is the main program. Below is the Send Text subroutine

```
Send Text:
        MOV   A, @R0        ;   more data pointer to by R0 into A
        JZ      finish ;   if contents of A are equal to 0, jump
                            ;   to finish
Loop:
        MOV   SBUF, A       ;   more contents of A to the serial
                            ;   buffer. This initiates data transmission
        INC     R0          ;   increment R0 to point to next character
        JNB     T1, &       ;   wait for entire character to be sent
        CLR     T1          ;   clear the transmit overflow
        JMP     LOOP        ;   go back to send next character
Finish:
```

RET

NOTE: To find the correct value to load into TH1 for mode 1, use the following formula:

$$TH1 = 256 - \left(\frac{system\ \ frequency}{12\ x\ 32} \Big/ band \right) \quad \text{for SMOD} = 0$$

For 9600 band rate

$$TH1 = 256 - \frac{12MHz}{12\ x\ 32} \Big/ 9600$$

$$= \frac{12\ x\ 10^6}{12\ x\ 32\ x\ 9600}$$

$$= 256 - 3.26 \quad \Rightarrow \quad \text{Error Margin} = 0.26$$

$$TH1 = 253$$

If SMOD = 1, then

$$TH1 = 256 - \frac{12\ x\ 10^6}{12\ x\ 16\ x\ 9600} = 256 - 6.51 \approx 249$$

$$\Leftrightarrow \text{ Error Margin} = 0.49$$

```
ORG ($ + 10H) AND 0FFF0H    ;  set location counter to next 16-byte
                            ;  boundary
ORG 50                      ;  set location counter to 50
```

CHAPTER SEVEN

8051 INTERRUPTS

7.1 INTERRUPTS

An interrupt is an occurrence of a event or condition that results in a temporary suspension of a program while the event or condition is serviced by another program (a section of code) called the interrupt service routine(ISR).Interrupts allow a system to respond asynchronously to an event and deal with it while another program is executing. An interrupt-driven-system gives an illusion of doing many things simultaneously. Actually, however, the CPU cannot execute more than one instruction at a time but it can suspend temporarily the execution of one program, executes another program and then return to the original program. Thus, the CPU handles interrupts in a similar way that it handles subroutines the only difference being that in interrupts, the interruption is a response to an event that occurs asynchronously with the main program and it is not known when the main program will be interrupted.

The interrupt service routine (ISR) or interrupt handler executes in response to the interrupt and generally performs an input or output operation to a device. When an interrupt occurs, the main program temporarily suspends execution and branches to the ISR, which executes, performs the operation and terminates with a “return from interrupt, INTI” instruction while the main program continues from where it left off The main program is said to be executing at base-level or foreground while the interrupt executes at the interrupt-level or background.

7.2 8051 INTERRUPTS

There are five sources of interrupts in the 8051:

(I) Two external interrupts provided through the 8051’s pins INTO-bar and INT1-bar which are respectively port 3 pin 2 (P3.2) and port 3 pin 3 (P3.3).

(II) Two internal interrupts generated by timer 0 and timer 1 overflows

(III) The serial port interrupts when a byte has been transmitted or received.

The interrupt flag bits are shown in Table 7.1

Table 7.1: Interrupt flag bits

Interrupt	**Flag**	**Location**
External 0	IE0	TCON.1
External 1	IE1	TCON.3
Timer 0	TF0	TCON.5
Timer 1	TF1	TCON.7
Serial port Receive	RI	SCON.0
Serial Port Transmit	TI	SCON.1

7.3 Interrupt Vectors

When an interrupt occurs the address of the interrupt service routine is loaded into the PC. This address is called the interrupt vector. Table 7.3 shows the 8051 interrupt vectors.

Table 7.3: 8051 interrupt vectors

Interrupt	**Flag**	**Location**
System reset	RST	0000H
External 0	IE0	0003H
External 1	IE1	0013H
Timer 0	TF0	000BH
Timer 1	TF1	001BH
Serial port	RI OR TI	0023H

The system reset is a special type of interrupt with interrupt vector 0000H. When this interrupt occurs the main program is interrupted and the PC is loaded with added 0000H which is the address the PC is first loaded with on system power up. Therefore reset interrupt is imply powering down and then powering up the system.

When vectoring to an interrupt the flag that caused the interrupt is automatically cleared by hardware. However, for serial port interrupt, there are two possible sources of the interrupt which makes it impractical for the interrupt flags to be cleared by hardware. These bits must therefore be tested in the ISR to determine the source of the interrupt and then the interrupting flag is cleared by software. Usually also a branch occurs to the appropriate action (whether transmit or receive) depending on the source of the interrupt.

It should be noted from Table 7.3 that each ISR is allocated only 8 memory locations meaning that each ISR should not exceed 8 bytes. However, if an ISR becomes more than 8 bytes the simple solution is to put a jump instruction at the appropriate ISR location to somewhere else in code memory and then put the ISR there.

7.4 ENABLING AND DISABLING INTERRUPTS

On power up or reset all interrupts are disabled. Each of the interrupt sources is then enabled or disabled individually through the bit addressable Interrupt Enable (IE) SFR at address A8H, Table 7.4. There is a global enable/disable bit that is cleared to disable all interrupts or set to turn on interrupts. Two bits must therefore be set to enable any interrupt: the individual enable bit and the global enable (EA) bit.

Table 7.4: Interrupt Enable (IE) SFR

Symbol	Bit Number	Bit Address	Description(1=Enable, 0=Disable)
EA	IE.7	AFH	Global Enable/Disable
-	IE.6	AEH	Undefined
-	IE.5	ADH	Undefined
ES	IE.4	ACH	Enable serial port interrupt
ET1	IE.3	ABH	Enable/Disable timer 1 overflow interrupt
EX1	IE.2	AAH	Enable/Disable external 1 interrupt
ET0	IE.1	A9H	Enable/Disable Timer 0 overflow interrupt
EX0	IE.0	A8H	Enable/Disable External 0 interrupt

To enable Timer 0 overflow interrupt the following code can be used.

```
SETB EA
SETB ET0
```

To disable this interrupt, the following instruction is used.

```
CLR ET0
```

To enable Timer 1 overflow interrupt would require a code like:

```
SETB EA
SETB ET1
```

OR generally,

```
MOV IE, #10001000B
```

Here two approaches are used to achieve the same purpose of enabling Timer 1 overflow interrupt. However whereas the first approach affects only the bits concerned the second one affects all the bits the IE register. Therefore using the second approach must take into cognizance what is intended for the other interrupt bits in order not to disable them when there were intended to be enabled.

7.5 Polling Sequence

The 8051 automatically evaluates whether an interrupt should occur after every instruction. When checking for interrupt conditions, it checks them in the following order:

- External 0 Interrupt
- Timer 0 Interrupt
- External 1 Interrupt
- Timer 1 Interrupt
- Serial Interrupt

This means that if a Serial Interrupt occurs at the exact same instant that an External 0 Interrupt occurs, the External 0 Interrupt will be executed first and the Serial Interrupt will be executed once the External 0 Interrupt has completed.

7.6 Interrupt Priorities

The 8051 offers two levels of interrupt priority: high and low. By using interrupt priorities you may assign higher priority to certain interrupt conditions. For example, you may have enabled Timer 1 Interrupt which is automatically called every time Timer 1 overflows. Additionally, you may have enabled the Serial Interrupt which is called every time a character is received via the serial port. However, you may consider that receiving a character is much more important than the timer interrupt. In this case, if Timer 1 Interrupt is already executing you may wish that the serial interrupt itself interrupts the Timer 1 Interrupt. When the serial interrupt is complete, control passes back to Timer 1 Interrupt and finally back to the main program. You may accomplish this by assigning a high priority to the Serial Interrupt and a low priority to the Timer 1 Interrupt.

Interrupt priorities are controlled by the Interrupt Priority (**IP)** SFR (B8H). The IP SFR has the format shown in Table 7.5.

Table 7.5: IR SFR

Bit	Name	Bit Address	Explanation of Function
7	-	-	Undefined
6	-	-	Undefined
5	-	-	Undefined
4	PS	BCh	Serial Interrupt Priority
3	PT1	BBh	Timer 1 Interrupt Priority
2	PX1	BAh	External 1 Interrupt Priority
1	PT0	B9h	Timer 0 Interrupt Priority
0	PX0	B8h	External 0 Interrupt Priority

When considering interrupt priorities, the following rules apply:

- Nothing can interrupt a high-priority interrupt--not even another high priority interrupt.
- A high-priority interrupt may interrupt a low-priority interrupt.

- A low-priority interrupt may only occur if no other interrupt is already executing.
- If two interrupts occur at the same time, the interrupt with higher priority will execute first. If both interrupts are of the same priority the interrupt which is serviced first by polling sequence will be executed first.

7.7 Actions Triggered When an Interrupt Occurs

When an interrupt is triggered, the following actions are taken automatically by the microcontroller:

- The current Program Counter is saved on the stack, low-byte first.
- Interrupts of the same and lower priority are blocked.
- In the case of Timer and External interrupts, the corresponding interrupt flag is cleared.
- Program execution transfers to the corresponding interrupt handler vector address.
- The Interrupt Handler Routine executes.

Take special note of the third step: If the interrupt being handled is a Timer or External interrupt, the microcontroller automatically clears the interrupt flag before passing control to your interrupt handler routine. This means it is not necessary that you clear the bit in your code.

7.8 When an Interrupt Ends

An interrupt ends when your program executes the RETI (Return from Interrupt) instruction. When the RETI instruction is executed the following actions are taken by the microcontroller:

- Two bytes are popped off the stack into the Program Counter to restore normal program execution.
- Interrupt status is restored to its pre-interrupt status.

7.9 Serial Interrupts

Serial Interrupts are slightly different than the rest of the interrupts. This is due to the fact that there are two interrupt flags: RI and TI. If either flag is set, a serial interrupt is triggered. As you will recall from the section on the serial port, the RI bit is set when a byte is received by the serial

port and the TI bit is set when a byte has been sent. This means that when the serial interrupt is executed, it may have been triggered because the RI flag was set or because the TI flag was set--or because both flags were set. Thus, service routine must check the status of these flags to determine what action is appropriate. Also, since the 8051 does not automatically clear the RI and TI flags the interrupt handler must clear these bits.

Example :

```
INT_SERIAL: JNB RI,CHECK_TI ;If the RI flag is not set, we jump to check TI
            MOV A,SBUF      ;If we got to this line, its because the RI bit *was* set
            CLR RI          ;Clear the RI bit after weve processed it
CHECK_TI:   JNB TI,EXIT_INT ;If the TI flag is not set, we jump to the exit point
            CLR TI          ;Clear the TI bit before we send another character
            MOV SBUF,#A     ;Send another character to the serial port
EXIT_INT:   RETI
```

As can be seen, the code checks the status of both interrupts flags. If both flags were set, both sections of code will be executed. Also note that each section of code clears its corresponding interrupt flag. If you forget to clear the interrupt bits, the serial interrupt will be executed over and over until you clear the bit. Thus it is very important that you always clear the interrupt flags in a serial interrupt.

7.10 Important Interrupt Consideration: Register Protection

One very important rule applies to all interrupt handlers: Interrupts must leave the processor in the same state as it was in when the interrupt initiated. Remember, the idea behind interrupts is that the main program is not aware that they are executing in the "background." However, consider the following code:

```
CLR C ;Clear carry
MOV A,#25H ;Load the accumulator with 25H
ADDC A,#10H ;Add 10H, with carry
```

After the above three instructions are executed, the accumulator will contain a value of 35H.

But what would happen if right after the MOV instruction an interrupt occurred. During this interrupt, the carry bit was set and the value of the accumulator was changed to 40H. When the interrupt finished and control was passed back to the main program, the ADDC would add 10H to 40H, and additionally add an additional 1H because the carry bit is set. In this case, the accumulator will contain the value 51H at the end of execution.

In this case, the main program has seemingly calculated the wrong answer by assigning 51H to 25H + 10H. A programmer that was unfamiliar with interrupts would be convinced that the microcontroller was damaged in some way, provoking problems with mathematical calculations. What has happened, in reality, is the interrupt did not protect the registers it used. An interrupt must leave the processor in the same state as it was in when the interrupt initiated.

Therefore if an interrupt uses the accumulator, it must ensure that the value of the accumulator is the same at the end of the interrupt as it was at the beginning. This is generally accomplished with a PUSH and POP sequence. For example:

```
PUSH ACC
PUSH PSW
MOV A,#0FFh
ADD A,#02h
POP PSW
POP ACC
```

The core of the interrupt is the MOV instruction and the ADD instruction. However, these two instructions modify the Accumulator (the MOV instruction) and also modify the value of the carry bit (the ADD instruction will cause the carry bit to be set). Since an interrupt routine must guarantee that the registers remain unchanged by the routine, the routine pushes the original values onto the stack using the PUSH instruction. It is then free to use the registers. Once the interrupt has finished its task, it pops the original values back into the registers. When the interrupt exits, the main program will never know the difference because the registers are exactly the same as they were before the interrupt executed.

In general, your interrupt routine must protect the following registers:

- PSW

- DPTR (DPH/DPL)
- PSW
- ACC
- B
- Registers R0-R7

7.11 Common Problems with Interrupts

Interrupts are a very powerful tool available to the 8051 developer, but when used incorrectly they can be a source of a huge number of debugging hours. Errors in interrupt routines are often very difficult to diagnose and correct. If you are using interrupts and your program is crashing or does not seem to be performing as you would expect, always review the following interrupt-related issues:

- **Register Protection**: Make sure you are protecting all your registers, as explained above. If you forget to protect a register that your main program is using, very strange results may occur. In our example above we saw how failure to protect registers caused the main program to apparently calculate that 25h + 10h = 51h. If you witness problems with registers changing values unexpectedly or operations producing "incorrect" values, it is very likely that you have forgotten to protect registers.
- **Forgetting to restore protected values**: Another common error is to push registers onto the stack to protect them, and then forget to pop them off the stack before exiting the interrupt. For example, you may push ACC, B, and PSW onto the stack in order to protect them and subsequently pop only ACC and PSW off the stack before exiting. In this case, since you forgot to restore the value of "B", an extra value remains on the stack. When you execute the RETI instruction the 8051 will use that value as the return address instead of the correct value. In this case, your program will almost certainly crash. Always make sure you pop the same number of values off the stack as you pushed onto it.

- **Using RET instead of RETI**: Remember that interrupts are always terminated with the RETI instruction. It is easy to inadvertently use the RET instruction instead. However, the RET instruction will not end your interrupt. Usually, using a RET instead of a RETI will cause the illusion of your main program running normally, but your interrupt will only be executed once.

If it appears that your interrupt mysteriously stops executing, verify that you are exiting with RETI.

7.12 Programs using interrupts

The suggested framework for a self-contained program using interrupts is as shown;

```
        ORG   0000H       ; ENTRY POINT
        LJMP  MAIN
        .                 ;ISR ENTRY POINT
        .
        ORG   0030H       ; MAIN PROGRAM ENTRY POINT
MAIN:   .                 ; MAIN PROGRAM BEGINS
        .
        .
```

The first instruction jumps to 0030H just above the vector locations where the ISRs begin while the main program begins at address 0030H.

7.12.1 Timer Interrupts

The timer interrupts occur when the timer overflow flag, TFx, is set consequent upon the overflow of the timer registers, THx/TLx. When the 8051 goes to service this interrupt, the timer overflow flag,TFx, is automatically cleared by hardware.

Example

Write a program using timer 0 and interrupt to create a 10KHz square wave on P2.0.

Solution

```
0000            5       ORG 0             ; Reset Entry Point
0000  020030    6       LJMP MAIN         ; Jump Above Interrupt Vectors
000B            7       ORG 000BH         ;Timer 0 Interrupt Vector
000B  B290      8  TOISR: CPL P2.0        ;Toggle Port Bit
000D  32        9       RETI
0030            10      ORG 0030H         ;Main Program Entry Point
0030  758902    11 MAIN: MOV TMOD, #02H   ;Timer 0 Mode 2
0033  758CCE    12      MOV TH0, #-50     ; 50 µs Delay
0036  D28C      13      SET TR0           ; Start Timer
```

```
0038  75A882      14          MOV IE, #82H        ; Enable 0 Interrupt
003B  80FE        15          SJMP $              ; Do Nothing
```

With the timer interrupt enabled, the event that generates the interrupt is the setting of the TF0 upon the overflow of TH0/TL0. When the first instruction(LJMP MAIN) is executed the program jumps over the timer ISR to address 0030H in code memory. The next three instructions initialize timer 0 for 8-bit auto-reload mode with overflow every 50µs. The MOV instruction in line 14 enables timer 0 interrupt so each overflow of the timer generates an interrupt. The first overflow does not occur until after 50µs so the main program executes line 15 and does nothing (or wait for the overflow).

Example

Write a program using interrupts to simultaneously create 7KHz and 500Hz square waves on P1.1 and p1.2.

Solution

```
0000                5                   ORG 0
0000  020030        6                   LJMP MAIN
000B                7                   ORG 000BH           ; Timer 0 Vector Address
000B  02003F        8                   LJMP TOISR
001B                9                   ORG 001BH           ;Timer 1 Vector Address
001B  020042        10                  LJMP T1ISR
0030                11                  ORG 0030H
0030  758912        12      MAIN:       MOV TMOD, #12H      ; Timer1 Mode1, Timer0
                    13                                      ;  Mode2
0033  758CB9        14                  MOV TH0, #-71       ; 7KHz using Timer 0
0036  D28C          15                  SET TR0
0038  D28F          16                  SET TF1             ; Force Timer 1 Interrupt
003A  75A88A        17                  MOV IE, #8AH        ;Enable timers interrupts
003D  80FE          18                  SJMP $
                    19                  ,
003F  B297          20      T0ISR:      CPL P1.1
0041  32            21                  RETI
```

```
0042  C28E      22    T1ISR:   CLR TR1
0044  758DFC    23             MOV TH1, #HIGH (-1000)   ; 1 ms High Time
0047  758B18    24             MOV TL1, #LOW (-1000)    ; 1 ms Low Time
004A  D28E      25             SETB TR1
004C  B296      26             CPL P1.2
004E  32        27             RETI
              28             END
```

7.12.2 Serial Port Interrupts

The serial port interrupt occurs when either the transmit interrupt flag (TI) or the receive interrupt flag (RI) is set. A transmit interrupt occurs when transmission of character previously written to SBUF is finished while a receive interrupt occurs when a character has been completely received and is waiting in SBUF to be used.

Example

Write a program using interrupts to continually transmit the ASCII code set (excluding control code) to a terminal attached to the 8051 serial port.

Solution

There are 128 7-bit codes in the ASCII chart. These consist of 95 graphic codes (20H to 7EH) and 33 control codes (00H to 1FH, and 7FH).

```
0000            5             ORG O
0000  020030    6             LJMP MAIN
0023            7             ORG  0023H          ; Serial port interrupt entry
0023  020042    8             LJMP SPISR
0030            9             ORG 0030H
0030  758920    10    MAIN:   MOV TMOD, #20H      ; Timer 1 Mode 2
0033  758DE6    11            MOV TH1, # -26      ; 12000 baud reload value
0036  D28E      12            SETB TR1            ; Start timer
0038  759842    13            MOV SCON, #42H      ; Mode 1, set TI to force 1st
```

```
                    14                                  ;interrupt, send 1st character
003B  7420          15              MOV A, #20H         ; Send ASCII space first
003D  75A890        16              MOV IE, 90H         ; Enable serial port interrupt
0040  80FE          17              SJMP $              ; Do nothing
                    18      ,
0042  B47F02        19      SPISR:  CJNE A, #7FH, SKIP  ; if finished ASCII set
0045  7420          20              MOV A, #20H         ; reset to SPACE
0047  F599          21      SKIP:   MOV SBUF, A         ;Send character to serial port
0049  04            22              INC A               ; Increment ASCII code
004A  C299          23              CLR TI              ; Clear interrupt flag
004C  32            24              RETI
                    25              END
```

7.12.3 External Interrupts

External interrupts occur as a result of a low-level or negative edge on the INT0-bar(p3.2) or INT1-ba (p3.3)r pin on the 8051.The flags that actually generate these interrupts are bits IE0 and IE1 in TCON. When an external interrupt is generated, the flag that generated it is cleared by hardware when vectoring to the ISR only if the interrupt was transition activated. If the interrupt was level-activated, then the external requesting source controls the level of the request flag rather than the on-chip hardware.

Example

Using interrupts, design an 8051 furnace controller that keeps a building at $20^0C \pm 1^0C$.

Solution

Assuming the furnace solenoid is connected to P1.1 such that

P1.1 engages the solenoid to turn ON the furnace for a logical HIGH and disengages the solenoid to turn off the furnace for a logical low. Temperature sensors are connected to INTO-bar and INT1-bar and provide HOT-bar and COLD-bar signals respectively such that

HOT-bar = 0 if T > 21^0C

COLD-bar = 0 if T < 19^0C

```
0000              5                ORG 0
0000  020030        6              LJMP MAIN
                    7                                  ; EXT 0 Vector at 0003H
0003  C297          8    EX0ISR:   CLR P1.1            ; Turn furnace OFF
0005  32            9              RETI
0013                10              ORG 0013
0013  D297          11   EX1ISR:   SETB P1.1           ; Turn furnace ON
0015  32            12             RETI
0030                13              ORG 30
0030  75A885        14   MAIN:     MOV IE, #85H        ; Enable external interrupts
0033  D288          15             SETB IT0            ; Negative edge triggered
0035  D28A          16             SETB IT1
0037  D297          17             SETB P1.1           ; Turn ON furnace
0039  20B202        18             JB P3.2, SKIP       ; If T> 21 degrees
003C  C297          19             CLR P1.1            ; turn furnace OFF
003E  80FE          20   SKIP:     SJMP $              ; Do nothing
                    21             END
```

Example

Design an intrusion warning system using interrupts that sounds a 400KHz tone for 1 second using a loudspeaker connected to p1.7 whenever a door sensor connected to INT0-bar makes a high-to-low transition.

Solution

The solution uses three interrupts including external 0 (door sensor), timer 0 (1 second timeout) and timer 1 (400 KHz tone)

```
0000              5                ORG 0
0000  020030        6              LJMP MAIN           ;3-byte instruction
0003  02003A        7              LJMP EX0ISR         ;EXT 0 vector address
```

```
000B             8          ORG 000BH          ; Timer 0 vector
000B 020045      9          LJMP T0ISR
001B            10          ORG 001BH          ; Timer 1 vector
001B 020059     11          LJMP T1ISR
0030            12          ORG 0030H
0030 D288       13  MAIN:   SETB IT0           ; Negative edge activated
0032 758911     14          MOV TMOD, #11H     ;16-bit timer mode
0035 75A881     15          MOV IE, #81H       ; Enable EXT 0 only
0038 80FE       16          SJMP $             ; Wait
                17  ;
003A 7F14       18  EX0ISR: MOV R7, #20        ;20 x 50000µs = 1 second
003C D28D       19          SETB TF0           ; Force timer 0 interrupt
003E D28F       20          SETB TF1           ; Force timer 1 interrupt
0040 D2A9       21          SETB ET0           ; begin tone for 1 second
0042 D2AB       22          SETB ET1           ; Enable timer interrupts
0044 32         23          RETI
                24  ;
0045 C28C       25  T0ISR:  CLR TR0            ; Stop timer
0047 DF07       26          DJNZ R7, SKIP      ; if not 20th time, skip
0049 C2A9       27          CLR ET0            ; if 20th disable
004B C2AB       28          CLR ET1            ; disable timer 1 interrupt
004D 020058     29          LJMP EXIT
0050 758C3C     30  SKIP:   MOV TH0, #HIGH(-50000)  ; 0.05 sec delay
0053 758AB0     31          MOV TL0, #LOW(-50000)
0056 D28C       32          SETB TR0
0058 32         33  EXIT:   RETI
                34  ;
0059 C28E       35  T1ISR:  CLR TR1
005B 758DFB     36          MOV TH1, #HIGH(-1250)    ; Count for 400 Hz
005E 758B1E     37          MOV TL1, #LOW (-1250)
0061 B297       38          CPL P1.7
0063 D28E       39          SETB TR1
0065 32         40          RETI
                41          END
```

This program consists of the interrupt vector locations, the main program and the three interrupt service routines. All vector locations contain LJMP instructions to the respective routines. The main program starting at code address 0030H contains only four instructions. SETB IT0 configures the door sensing interrupt input as negative-edge triggered. MOV TMOD, #11H configures both timers for mode 1, 16-bit timer mode. Only the external 0 interrupt is enabled initially (MOV IE, #81H), so a "door-open" condition is needed before any interrupt is accepted. Finally, the LJMP $ instruction puts the main program in a waiting state.

When a door-open condition is sensed (by a 1-to-0 transition of INT0-bar), an external 0 interrupt is generated. EX0ISR begins by putting the constant 20 in R7, then sets the overflow flags for both timers to force timer interrupts to occur.

Timer interrupts will only occur, however, if the respective bits are enabled in the IE register. The next two instructions (SETB ET0) and (SETB ET1) enable timer interrupts. Finally, EX0ISR terminates with a RETI to the main program.

Timer 0 creates a 1 second time out, and timer 1 creates a 400Hz tone. After EX0ISR returns to the main program, timer interrupts are immediately generated (and accepted after one execution of SJMP $). Because of the fixed polling sequence, timer 0 interrupt is serviced first. A 1 second timeout is created by programming 20 repetitions of a 50,000μs timeout. R7 serves as the counter. Nineteen times out of 20, T0ISR operates thus:

Timer 0 is first turned off and R7 is decremented. Then TH0/TL0 is reloaded with -50,000, the timer is turned back on, and the interrupt is terminated. On the 20^{th} Timer 0 interrupt, R7 is decremented to 0 (1 second has elapsed). Both timer interrupts are disabled (CLR ET0, CLR ET1) and the interrupt is terminated. No further timer interrupt will be generated until the next "door-open" condition is sensed.

The 400Hz tone is programmed using timer 1 interrupts, 400 Hz requires a period of 1/400 = 2500 μs or 1250μs high and 1250μs low-time. Each timer ISR simply puts -1250 in TH1/TL1, complements the port bit driving the loudspeaker, then terminates.

CHAPTER EIGHT

PARALLEL COMMUNICATION

8.0 Introduction to LCD Programming

Frequently, an 8051 program must interact with the outside world using input and output devices that communicate directly with a human being. One of the most common devices attached to an 8051 is an LCD display. Some of the most common LCDs connected to the 8051 are 16x2 and 20x2 displays. This means 16 characters per line by 2 lines and 20 characters per line by 2 lines, respectively.

Fortunately, a very popular standard exists which allows us to communicate with the vast majority of LCDs regardless of their manufacturer. The standard is referred to as HD44780U, which refers to the controller chip which receives data from an external source (in this case, the 8051) and communicates directly with the LCD.

8.1 44780 BACKGROUND

The 44780 standard requires 3 control lines as well as either 4 or 8 I/O lines for the data bus. The user may select whether the LCD is to operate with a 4-bit data bus or an 8-bit data bus. If a 4-bit data bus is used the LCD will require a total of 7 data lines (3 control lines plus the 4 lines for the

data bus). If an 8-bit data bus is used the LCD will require a total of 11 data lines (3 control lines plus the 8 lines for the data bus).

The three control lines are referred to as **EN**, **RS**, and **RW**.

The **EN** line is called "Enable." This control line is used to tell the LCD that you are sending it data. To send data to the LCD, your program should make sure this line is low (0) and then set the other two control lines and/or put data on the data bus. When the other lines are completely ready, bring **EN** high (1) and wait for the minimum amount of time required by the LCD datasheet (this varies from LCD to LCD), and end by bringing it low (0) again.

The **RS** line is the "Register Select" line. When RS is low (0), the data is to be treated as a command or special instruction (such as clear screen, position cursor, etc.). When RS is high (1), the data being sent is text data which should be displayed on the screen. For example, to display the letter "T" on the screen you would set RS high.

The **RW** line is the "Read/Write" control line. When RW is low (0), the information on the data bus is being written to the LCD. When RW is high (1), the program is effectively querying (or reading) the LCD. Only one instruction ("Get LCD status") is a read command. All others are write commands--so RW will almost always be low.

Finally, the data bus consists of 4 or 8 lines (depending on the mode of operation selected by the user). In the case of an 8-bit data bus, the lines are referred to as DB0, DB1, DB2, DB3, DB4, DB5, DB6, and DB7.

8.2 AN EXAMPLE HARDWARE CONFIGURATION

The LCD requires either 8 or 11 I/O lines to communicate with. We will be using 11 of the 8051's I/O pins to interface with the LCD.

A sample psuedo-schematic of how the LCD will be connected to the 8051 is shown in Figure 8.1.

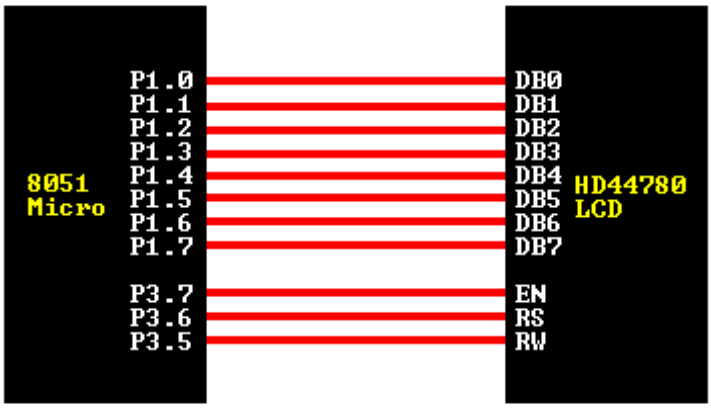


Figure 8.1: A psuedo-schematic LCD connection to the 8051

In Figure 8.1, a 1-to-1 relation between a pin on the 8051 and a line on the 44780 LCD was established. Thus as assembly program to access the LCD is written, constants to equate the 8051 ports will be written so that we can refer to the lines by their 44780 name as opposed to P0.1, P0.2, etc. Let's go ahead and write our initial equates:

```
DB0 EQU P1.0
DB1 EQU P1.1
DB2 EQU P1.2
DB3 EQU P1.3
DB4 EQU P1.4
DB5 EQU P1.5
DB6 EQU P1.6
DB7 EQU P1.7
EN EQU P3.7
RS EQU P3.6
RW EQU P3.5
DATA EQU P1
```

Having established the above equates, we may now refer to our I/O lines by their 44780 name. For example, to set the RW line high (1), we can execute the following instruction:

```
SETB RW
```

8.3 HANDLING THE EN CONTROL LINE

As we mentioned above, the EN line is used to tell the LCD that you are ready for it to execute an instruction that you've prepared on the data bus and on the other control lines. Note that the EN line must be raised/lowered before/after each instruction sent to the LCD regardless of whether that instruction is read or write, text or instruction. In short, you must always manipulate EN when communicating with the LCD. EN is the LCD's way of knowing that you are talking to it. If you don't raise/lower EN, the LCD doesn't know you're talking to it on the other lines.

Thus, before we interact in any way with the LCD we will always bring the **EN** line low with the following instruction:

CLR EN

And once we've finished setting up our instruction with the other control lines and data bus lines, we'll always bring this line high:

SETB EN

The line must be left high for the amount of time required by the LCD as specified in its datasheet. This is normally on the order of about 250 nanoseconds, but check the datasheet. In the case of a typical 8051 running at 12 MHz, an instruction requires 1.08 microseconds to execute so the EN line can be brought low the very next instruction. However, faster microcontrollers (such as the DS89C420 which executes an instruction in 90 nanoseconds given an 11.0592 Mhz crystal) will require a number of NOPs to create a delay while EN is held high. The number of NOPs that must be inserted depends on the microcontroller you are using and the crystal you have selected.

The instruction is executed by the LCD at the moment the EN line is brought low with a final CLR EN instruction.

8.4 CHECKING THE BUSY STATUS OF THE LCD

As previously mentioned, it takes a certain amount of time for each instruction to be executed by the LCD. The delay varies depending on the frequency of the crystal attached to the oscillator input of the 44780 as well as the instruction which is being executed.

While it is possible to write code that waits for a specific amount of time to allow the LCD to execute instructions, this method of "waiting" is not very flexible. If the crystal frequency is changed, the software will need to be modified. Additionally, if the LCD itself is changed for another LCD which, although 44780 compatible, requires more time to perform its operations, the program will not work until it is properly modified.

A more robust method of programming is to use the "Get LCD Status" command to determine whether the LCD is still busy executing the last instruction received.

The "Get LCD Status" command will return to us two tidbits of information; the information that is useful to us right now is found in DB7. In summary, when we issue the "Get LCD Status" command the LCD will immediately raise DB7 if it's still busy executing a command or lower DB7 to indicate that the LCD is no longer occupied. Thus our program can query the LCD until DB7 goes low, indicating the LCD is no longer busy. At that point we are free to continue and send the next command.

Since we will use this code every time we send an instruction to the LCD, it is useful to make it a subroutine. Let's write the code:

```
WAIT_LCD:
        CLR EN ;Start LCD command
        CLR RS ;It's a command
        SETB RW ;It's a read command
        MOV DATA,#0FFh ;Set all pins to FF initially
        SETB EN ;Clock out command to LCD
        MOV A,DATA ;Read the return value
        JB ACC.7,WAIT_LCD ;If bit 7 high, LCD still busy
        CLR EN ;Finish the command
        CLR RW ;Turn off RW for future commands
        RET
```

Thus, our standard practice will be to send an instruction to the LCD and then call our **WAIT_LCD** routine to wait until the instruction is completely executed by the LCD. This will assure that our program gives the LCD the time it needs to execute instructions and also makes our program compatible with any LCD, regardless of how fast or slow it is.

8.5 INITIALIZING THE LCD

Before you may really use the LCD, you must initialize and configure it. This is accomplished by sending a number of initialization instructions to the LCD.

The first instruction we send must tell the LCD whether we'll be communicating with it with an 8-bit or 4-bit data bus. We also select a 5x8 dot character font. These two options are selected by

sending the command 38h to the LCD as a command. As you will recall from the last section, we mentioned that the **RS** line must be low if we are sending a command to the LCD. Thus, to send this 38h command to the LCD we must execute the following 8051 instructions:

```
CLR RS
MOV DATA,#38h
SETB EN
CLR EN
LCALL WAIT_LCD
```

We've now sent the first byte of the initialization sequence. The second byte of the initialization sequence is the instruction 0Eh. Thus we must repeat the initialization code from above, but now with the instruction. Thus the next code segment is:

```
CLR RS
MOV DATA,#0Eh
SETB EN
CLR EN
LCALL WAIT_LCD
```

The last byte we need to send is used to configure additional operational parameters of the LCD. We must send the value 06h.

```
CLR RS
MOV DATA,#06h
SETB EN
CLR EN
LCALL WAIT_LCD
.
```

So, in all, our initialization code is as follows:

```
INIT_LCD:
        CLR RS
        MOV DATA,#38h
        SETB EN
        CLR EN
```

```
        LCALL WAIT_LCD
        CLR RS
        MOV DATA,#0Eh
        SETB EN
        CLR EN
        LCALL WAIT_LCD
        CLR RS
        MOV DATA,#06h
        SETB EN
        CLR EN
        LCALL WAIT_LCD
        RET
```

Having executed this code the LCD will be fully initialized and ready for us to send display data to it.

8.6 CLEARING THE DISPLAY

When the LCD is first initialized, the screen should automatically be cleared by the 44780 controller. However, it's always a good idea to do things yourself so that you can be completely sure that the display is the way you want it. Thus, it's not a bad idea to clear the screen as the very first operation after the LCD has been initialized.

An LCD command exists to accomplish this function. Not surprisingly, it is the command 01h. Since clearing the screen is a function we very likely will wish to call more than once, it's a good idea to make it a subroutine:

```
CLEAR_LCD:
        CLR RS
        MOV DATA,#01h
        SETB EN
        CLR EN
        LCALL WAIT_LCD
        RET
```

How that we've written a "Clear Screen" routine, we may clear the LCD at any time by simply executing an **LCALL CLEAR_LCD**.

8.7 WRITING TEXT TO THE LCD

Once again, writing text to the LCD is something we will almost certainly want to do over and over; so let us make it a subroutine.

```
WRITE_TEXT:
        SETB RS
        MOV DATA, A
        SETB EN
        CLR EN
        LCALL WAIT_LCD
        RET
```

The **WRITE_TEXT** routine that we just wrote will send the character in the accumulator to the LCD which will, in turn, display it. Thus to display text on the LCD all we need to do is load the accumulator with the byte to display and make a call to this routine.

8.8 A "HELLO WORLD" PROGRAM

Now that we have all the component subroutines written, writing the classic "Hello World" program--which displays the text "Hello World" on the LCD is a relatively easy matter. Consider:

```
LCALL INIT_LCD
LCALL CLEAR_LCD
MOV A,#'H'
LCALL WRITE_TEXT
MOV A,#'E'
LCALL WRITE_TEXT
MOV A,#'L'
LCALL WRITE_TEXT
MOV A,#'L'
LCALL WRITE_TEXT
MOV A,#'O'
LCALL WRITE_TEXT
MOV A,#' '
LCALL WRITE_TEXT
MOV A,#'W'
```

```
LCALL WRITE_TEXT
MOV A,#'O'
LCALL WRITE_TEXT
MOV A,#'R'
LCALL WRITE_TEXT
MOV A,#'L'
LCALL WRITE_TEXT
MOV A,#'D'
LCALL WRITE_TEXT
```

The above "Hello World" program should, when executed, initialize the LCD, clear the LCD screen, and display "Hello World" in the upper left-hand corner of the display.

8.9 CURSOR POSITIONING

The above "Hello World" program is simplistic in the sense that it prints its text in the upper left-hand corner of the screen. However, what if we wanted to display the word "Hello" in the upper left-hand corner but wanted to display the word "World" on the second line at the tenth character? This sounds simple--and actually, it **is** simple. However, it requires a little more understanding of the design of the LCD.

The 44780 contains a certain amount of memory which is assigned to the display. All the text we write to the 44780 is stored in this memory, and the 44780 subsequently reads this memory to display the text on the LCD itself. This memory can be represented with the "memory map" of Figure 8.1:

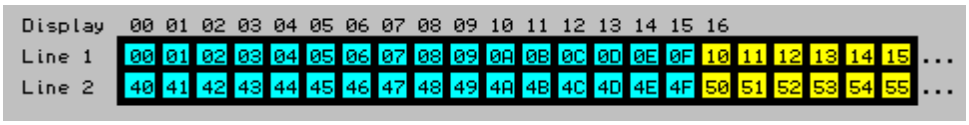


Figure 8.1: LCD Memory map

In the memory map of Figure 8.1, the area shaded in blue is the visible display. It measures 16 characters per line by 2 lines. The numbers in each box is the memory address that corresponds to that screen position. Thus, the first character in the upper left-hand corner is at address 00h.

The following character position (character #2 on the first line) is address 01h, etc. This continues until we reach the 16th character of the first line which is at address 0Fh.

However, the first character of line 2, as shown in the memory map, is at address 40h. This means if we write a character to the last position of the first line and then write a second character, the second character will **not** appear on the second line. That is because the second character will effectively be written to address 10h--but the second line begins at address 40h.

Thus we need to send a command to the LCD that tells it to position the cursor on the second line. The "Set Cursor Position" instruction is 80h. To this we must add the address of the location where we wish to position the cursor. In our example, we said we wanted to display "World" on the second line on the tenth character position.

Referring again to the memory map, we see that the tenth character position of the second line is address 4Ah. Thus, before writing the word "World" to the LCD, we must send a "Set Cursor Position" instruction--the value of this command will be 80h (the instruction code to position the cursor) plus the address 4Ah. 80h + 4Ah = CAh. Thus sending the command CAh to the LCD will position the cursor on the second line at the tenth character position:

```
CLR RS
MOV DATA,#0CAh
SETB EN
CLR EN
LCALL WAIT_LCD
```

The above code will position the cursor on line 2, character 10. To display "Hello" in the upper left-hand corner with the word "World" on the second line at character position 10 just requires us to insert the above code into our existing "Hello World" program. This results in the following:

```
LCALL INIT_LCD
LCALL CLEAR_LCD
MOV A,#'H'
LCALL WRITE_TEXT
MOV A,#'E'
LCALL WRITE_TEXT
MOV A,#'L'
LCALL WRITE_TEXT
MOV A,#'L'
```

```
LCALL WRITE_TEXT
MOV A,#'O'
LCALL WRITE_TEXT
CLR RS
MOV DATA,#0CAh
SETB EN
CLR EN
LCALL WAIT_LCD
MOV A,#'W'
LCALL WRITE_TEXT
MOV A,#'O'
LCALL WRITE_TEXT
MOV A,#'R'
LCALL WRITE_TEXT
MOV A,#'L'
LCALL WRITE_TEXT
MOV A,#'D'
LCALL WRITE_TEXT
```

9.7 Keypad Interface

K E Y P A D 4 x 4

Keypads are often used as a primary input device for embedded microcontrollers. The keypads actually consist of a number of switches, connected in a row/column arrangement as shown in Figure 9.10.

In order for the microcontroller to scan the keypad, it outputs a nibble to force one (only one) of the columns low and then reads the rows to see if any buttons in that column have been pressed. The rows are pulled up by the internal weak pull-ups in the 8051 ports. Consequently, as long as no buttons are pressed, the microcontroller sees a logic high on each of the pins attached to the keypad rows. The nibble driven onto the columns always contains only a single 0. The only way the microcontroller can find a 0 on any row pin is for the keypad button to be pressed that connects the column set to 0 to a row. The controller knows which column is at a 0-level and which row reads 0, allowing it to determine which key is pressed. For the keypad, the pins from left to right are: R1, R2, R3, R4, C1, C2, C3, C4.

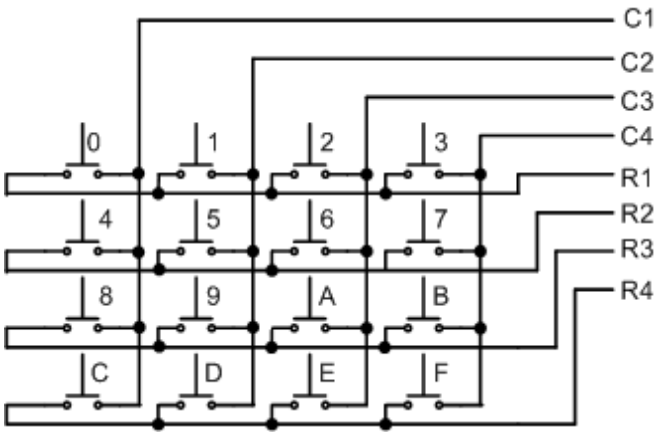


Figure 9.10: Keypad 4 X 4 Connection

9.7.1 The Algorithm

Matrix-type keypads consist of a rectangular array of momentary push button. Each row and each column of push buttons is connected to a common rail. Suppose a four by four array of push button are used. A four by four array is often used to input hexadecimal numbers. There are four column rails and four row rails. Each pushbutton has two terminals, one connected to its column rail and the other to its row rail. The row and column rails are connected to the microcontroller ports. The columns are driven low by output port. The rows are then read into the input ports. If no key is pressed, the rows read 1. When a row is detected to be 0, it indicates that a key in that row is pressed. the task now is to detect which key of the row is actually pressed. The microcontroller loops through each column, driving only one column low at a time as it inspects the row. The microcontroller needs to poll the rows to see if a key is pressed. Only when the column in which the pressed key resides is driven low is the row rail grounded, and thus the voltage is low. The rows and columns are interchangeable, that is, the rows may be driven low as the columns are read by the input ports.

9.7.2 Display Keypad Data to LED

In this lesson we will like to design how to scan keypad 4 x 4, and then display it to LED.

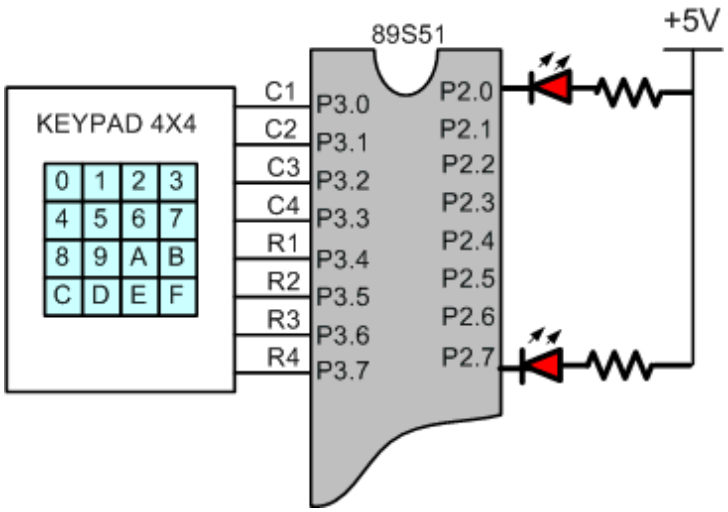


Figure 9.11: Keypad connection and display to LED

P3.0 through P3.7 connected to keypad 4 x 4 and driven LEDs are connected to P2.0 through P2.7,Figure 9.11.

The assembly program to scan your keypad data.

```
;IN THIS LESSON WE'LL SCAN KEYPAD AND GET DATA OUT TO LED
;AND CONVERT IT INTO BINARY DATA
ROW1 BIT P2.4
ROW2 BIT P2.5
ROW3 BIT P2.6
ROW4 BIT P2.7
COL1 BIT P2.0
COL2 BIT P2.1
COL3 BIT P2.2
COL4 BIT P2.3
;
KEYDATA   EQU 70H
KEYBOUNC  EQU 71H
KEYPORT   EQU P2
      ORG 0H
START: CALL KEYPAD4X4    ;CALLING SUBROUTINE KEYPAD4X4
      MOV A,KEYDATA     ;A = KEYDATA
      CJNE A,#0FFH,SEND ;
      SJMP START        ;LOOPING FOREVER
SEND:  CPL A            ;A = NOT A
```

```
      MOV P0,A         ;P0 = A
      SJMP START       ;LOOPING FOREVER PART 2
;
DELAY: MOV R0,#0
DELAY1:MOV R2,#50
      DJNZ R2,$
      DJNZ R0,DELAY1
      RET
;
;===================================
; SUBROUTINE SCAN KEYPAD 4X4
;===================================
KEYPAD4X4:
      MOV KEYBOUNC,#50    ;KEYBOUNC = 50
      MOV KEYPORT,#0FFH   ;KEYPORT=P2= FF
      CLR COL1            ;COL1= P3.0 = 0
DETECT:JB ROW1,KEY1       ;JUMP TO KEY1 IF ROW1=1
      DJNZ KEYBOUNC,DETECT
      MOV KEYDATA,#00H    ;KEYDATA =00H
      RET
;
KEY1: JB ROW2,KEY2        ;JUMP TO KEY2 IF ROW2=1
     DJNZ KEYBOUNC,KEY1
     MOV KEYDATA,#04H     ;KEYDATA = 04H
     RET
;
KEY2: JB ROW3,KEY3        ; IDEM
     DJNZ KEYBOUNC,KEY2
     MOV KEYDATA,#08H
     RET
;
KEY3: JB ROW4,KEY4        ; IDEM
     DJNZ KEYBOUNC,KEY3
     MOV KEYDATA,#0CH
```

```
    RET
;
KEY4: SETB COL1
    CLR COL2
    JB ROW1,KEY5
    DJNZ KEYBOUNC,KEY4
    MOV KEYDATA,#01H
    RET
;
KEY5: JB ROW2,KEY6
    DJNZ KEYBOUNC,KEY5
    MOV KEYDATA,#05H
    RET
;
KEY6: JB ROW3,KEY7
    DJNZ KEYBOUNC,KEY6
    MOV KEYDATA,#09H
    RET
;
KEY7: JB ROW4,KEY8
    DJNZ KEYBOUNC,KEY7
    MOV KEYDATA,#0DH
    RET
;
KEY8: SETB COL2
    CLR COL3
    JB ROW1,KEY9
    DJNZ KEYBOUNC,KEY8
    MOV KEYDATA,#02H
    RET
;
KEY9: JB ROW2,KEYA
    DJNZ KEYBOUNC,KEY9
    MOV KEYDATA,#06H
```

```
    RET
;
KEYA: JB ROW3,KEYB
    DJNZ KEYBOUNC,KEYA
    MOV KEYDATA,#0AH
    RET
;
KEYB: JB ROW4,KEYC
    DJNZ KEYBOUNC,KEYB
    MOV KEYDATA,#0EH
    RET
;
KEYC: SETB COL3
    CLR COL4
    JB ROW1,KEYD
    DJNZ KEYBOUNC,KEYC
    MOV KEYDATA,#03H
    RET
;
KEYD: JB ROW2,KEYE
    DJNZ KEYBOUNC,KEYD
    MOV KEYDATA,#07H
    RET
;
KEYE: JB ROW3,KEYF
    DJNZ KEYBOUNC,KEYE
    MOV KEYDATA,#0BH
    RET
;
KEYF: JB ROW4,NOKEY
    DJNZ KEYBOUNC,KEYF
    MOV KEYDATA,#0FH
    RET
NOKEY:MOV KEYDATA,#0FFH
```

```
    RET
;================================
;THE END OF KEYPAD 4X4 SUBROUTINE
;================================
END
```

9.7.3 Display Keypad 4 x 4 with LCD Character 2 x16

Program to scan keypad and read data out to LED. The data will be read out with LCD Character 2 x16.

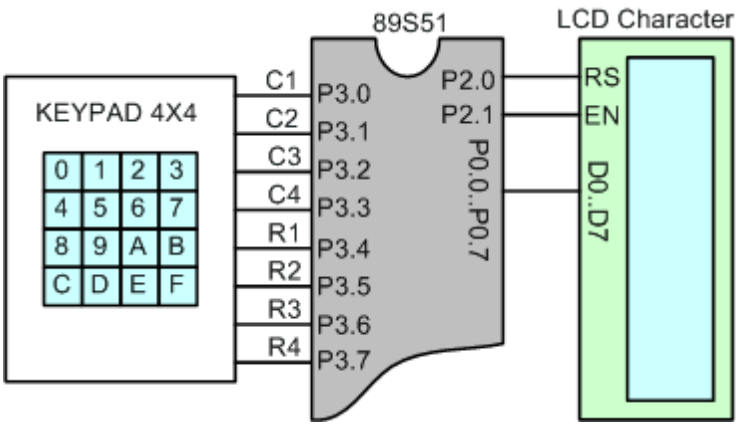


Figure 9.12: Keypad Connection to Microcontroller

P3.0 through P3.7 connected to keypad 4 x 4 and LCD Character 2x16 to read keypad data is connected to P0.0 through P0.7., and P2.0 and P2.1 are connected to RS and EN each, Figure 9.12.

The assembly program to scan your keypad data.

```
;THE FOLLOWING EXPERIMENT IS USED TO SCAN
;KEYPAD 4X4 AND RESULT OF SCAN WILL BE RELEASED
;TO LCD CHARACTER
ROW1 BIT P2.4
ROW2 BIT P2.5
ROW3 BIT P2.6
ROW4 BIT P2.7
COL1 BIT P2.0
COL2 BIT P2.1
```

```
COL3 BIT P2.2
COL4 BIT P2.3
;
KEYDATA   EQU 70H
KEYBOUNC  EQU 71H
KEYPORT   EQU P2
      ORG 0H
START: CALL KEYPAD4X4    ;CALLING SUBROUTINE KEYPAD4X4
      MOV A,KEYDATA     ;A = KEYDATA
      CJNE A,#0FFH,WRLCD;
      SJMP START        ;LOOPING FOREVER PART 1
;
WRLCD: MOV R1,#80H       ;PICK DDRAM 1ST ROW AND 1ST COL
      CALL WRITE_INST
      MOV R1,A
      CALL WRITE_DATA   ;WRITE DATA
      SJMP START        ;LOOPING FOREVER PART 2
;
INIT_LCD:
   MOV R1,#00000001B ;DISPLAY CLEAR
   ACALL WRITE_INST  ;
   MOV R1,#00111000B ;FUNCTION SET,
                     ;DATA 8 BIT,2 LINE FONT 5X7
   ACALL WRITE_INST  ;
   MOV R1,#00001100B ;DISPLAY ON,
                     ;CURSOR OFF,CURSOR BLINK OFF
   ACALL WRITE_INST
   MOV R1,#00000110B ;ENTRY MODE, SET INCREMENT
   ACALL WRITE_INST
   RET
;
WRITE_INST:
   CLR P2.0  ; RS = P2.0 = 0, WRITE MODE INSTRUCTION
   MOV P0,R1 ; D7 S/D D0 = P0 = R1
```

```
   SETB P2.1 ; EN = 1 = P2.1
   CALL DELAY; CALL DELAY TIME
   CLR P2.1  ; EN = 0 = P2.1
   RET
;
WRITE_DATA:
   SETB P2.0 ; RS = P2.0 = 1, WRITE MODE DATA
   MOV P0,R1 ; D7 S/D D0 = P0 = R1
   SETB P2.1 ; EN = 1 = P2.1
   CALL DELAY; CALL DELAY TIME
   CLR P2.1  ; EN = 0 = P2.1
   RET
;
DELAY: MOV R0,#0
DELAY1:MOV R2,#50
      DJNZ R2,$
      DJNZ R0,DELAY1
      RET
;
;===================================
; SUBROUTINE SCAN KEYPAD 4X4
;===================================
KEYPAD4X4:
      MOV KEYBOUNC,#50    ;KEYBOUNC = 50
      MOV KEYPORT,#0FFH   ;KEYPORT=P2= FF
      CLR COL1            ;COL1= P3.0 = 0
DETECT:JB ROW1,KEY1       ;JUMP TO KEY1 IF ROW1=1
      DJNZ KEYBOUNC,DETECT
      MOV KEYDATA,#00H    ;KEYDATA =00H
      RET
;
KEY1: JB ROW2,KEY2        ;JUMP TO KEY2 IF ROW2=1
     DJNZ KEYBOUNC,KEY1
     MOV KEYDATA,#04H     ;KEYDATA = 04H
```

```
    RET
;
KEY2: JB ROW3,KEY3      ; IDEM
    DJNZ KEYBOUNC,KEY2
    MOV KEYDATA,#08H
    RET
;
KEY3: JB ROW4,KEY4      ; IDEM
    DJNZ KEYBOUNC,KEY3
    MOV KEYDATA,#0CH
    RET
;
KEY4: SETB COL1
    CLR COL2
    JB ROW1,KEY5
    DJNZ KEYBOUNC,KEY4
    MOV KEYDATA,#01H
    RET
;
KEY5: JB ROW2,KEY6
    DJNZ KEYBOUNC,KEY5
    MOV KEYDATA,#05H
    RET
;
KEY6: JB ROW3,KEY7
    DJNZ KEYBOUNC,KEY6
    MOV KEYDATA,#09H
    RET
;
KEY7: JB ROW4,KEY8
    DJNZ KEYBOUNC,KEY7
    MOV KEYDATA,#0DH
    RET
;
```

```
KEY8: SETB COL2
      CLR COL3
      JB ROW1,KEY9
      DJNZ KEYBOUNC,KEY8
      MOV KEYDATA,#02H
      RET
;
KEY9: JB ROW2,KEYA
      DJNZ KEYBOUNC,KEY9
      MOV KEYDATA,#06H
      RET
;
KEYA: JB ROW3,KEYB
      DJNZ KEYBOUNC,KEYA
      MOV KEYDATA,#0AH
      RET
;
KEYB: JB ROW4,KEYC
      DJNZ KEYBOUNC,KEYB
      MOV KEYDATA,#0EH
      RET
;
KEYC: SETB COL3
      CLR COL4
      JB ROW1,KEYD
      DJNZ KEYBOUNC,KEYC
      MOV KEYDATA,#03H
      RET
;
KEYD: JB ROW2,KEYE
      DJNZ KEYBOUNC,KEYD
      MOV KEYDATA,#07H
      RET
;
```

```
KEYE: JB ROW3,KEYF
    DJNZ KEYBOUNC,KEYE
    MOV KEYDATA,#0BH
    RET
;
KEYF: JB ROW4,NOKEY
    DJNZ KEYBOUNC,KEYF
    MOV KEYDATA,#0FH
    RET
NOKEY:MOV KEYDATA,#0FFH
    RET
;================================
;THE END OF KEYPAD 4X4 SUBROUTINE
;================================
END
```

9.7.4 Display Keypad 4 x 4 with 8x7 Seven Segment

This is a more complex experiment to read out the keypad 4 x 4 data with 8x7 Seven Segment.

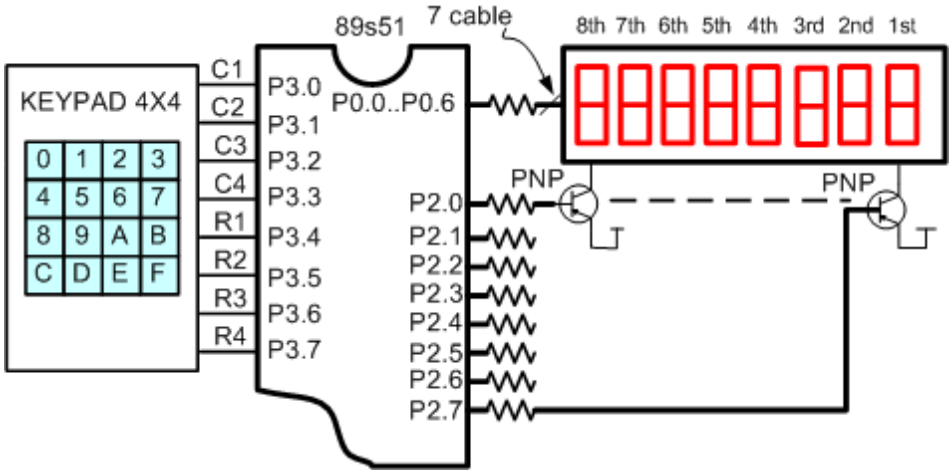


Figure 9.13: Keypad Connection to Microcontroller with 8x7 segment

P3.0 through P3.7 is connected to keypad 4 x 4 and 8 x 7 Segment, to read keypad data, connected to P0.0 through P0.7. while P2.0 and P2.1 are connected to RS and EN respectively, Figure 9.13. The assembly program to scan your keypad data.

```
;THE FOLLOWING EXPERIMENT IS USED TO SCAN
;KEYPAD 4X4 AND RESULT OF SCAN WILL BE RELEASED
;TO DISPLAY 7 SEGMEN
ROW1 BIT P2.4
ROW2 BIT P2.5
ROW3 BIT P2.6
ROW4 BIT P2.7
COL1 BIT P2.0
COL2 BIT P2.1
COL3 BIT P2.2
COL4 BIT P2.3
;
KEYDATA   EQU 70H
KEYBOUNC  EQU 71H
KEYPORT   EQU P2
     ORG 0H
START: CALL KEYPAD4X4    ;CALLING SUBRUTINE KEYPAD4X4
     MOV A,KEYDATA     ;A = KEYDATA
     CJNE A,#0FFH,WR7SEG;
     SJMP START        ;LOOPING FOREVER PART 1
;
WR7SEG:
;================================================
;I LEFT THE ASSEMBLY INSTRUCTION FOR YOU TO LEARN
;================================================
;
DELAY: MOV R0,#0
DELAY1:MOV R2,#50
     DJNZ R2,$
     DJNZ R0,DELAY1
     RET
;
;====================================
; SUBROUTINE SCAN KEYPAD 4X4
```

```
;===================================
KEYPAD4X4:
      MOV KEYBOUNC,#50    ;KEYBOUNC = 50
      MOV KEYPORT,#0FFH   ;KEYPORT=P2= FF
      CLR COL1            ;COL1= P3.0 = 0
DETECT:JB ROW1,KEY1       ;JUMP TO KEY1 IF ROW=1
      DJNZ KEYBOUNC,DETECT
      MOV KEYDATA,#00H    ;KEYDATA =00H
      RET
;
KEY1: JB ROW2,KEY2        ;JUMP TO KEY2 IF ROW2=1
     DJNZ KEYBOUNC,KEY1
     MOV KEYDATA,#04H     ;KEYDATA = 04H
     RET
;
KEY2: JB ROW3,KEY3        ; IDEM
     DJNZ KEYBOUNC,KEY2
     MOV KEYDATA,#08H
     RET
;
KEY3: JB ROW4,KEY4        ; IDEM
     DJNZ KEYBOUNC,KEY3
     MOV KEYDATA,#0CH
     RET
;
KEY4: SETB COL1
     CLR COL2
     JB ROW1,KEY5
     DJNZ KEYBOUNC,KEY4
     MOV KEYDATA,#01H
     RET
;
KEY5: JB ROW2,KEY6
     DJNZ KEYBOUNC,KEY5
```

```
    MOV KEYDATA,#05H
    RET
;
KEY6: JB ROW3,KEY7
    DJNZ KEYBOUNC,KEY6
    MOV KEYDATA,#09H
    RET
;
KEY7: JB ROW4,KEY8
    DJNZ KEYBOUNC,KEY7
    MOV KEYDATA,#0DH
    RET
;
KEY8: SETB COL2
    CLR COL3
    JB ROW1,KEY9
    DJNZ KEYBOUNC,KEY8
    MOV KEYDATA,#02H
    RET
;
KEY9: JB ROW2,KEYA
    DJNZ KEYBOUNC,KEY9
    MOV KEYDATA,#06H
    RET
;
KEYA: JB ROW3,KEYB
    DJNZ KEYBOUNC,KEYA
    MOV KEYDATA,#0AH
    RET
;
KEYB: JB ROW4,KEYC
    DJNZ KEYBOUNC,KEYB
    MOV KEYDATA,#0EH
    RET
```

```
;
KEYC: SETB COL3
      CLR COL4
      JB ROW1,KEYD
      DJNZ KEYBOUNC,KEYC
      MOV KEYDATA,#03H
      RET
;
KEYD: JB ROW2,KEYE
      DJNZ KEYBOUNC,KEYD
      MOV KEYDATA,#07H
      RET
;
KEYE: JB ROW3,KEYF
      DJNZ KEYBOUNC,KEYE
      MOV KEYDATA,#0BH
      RET
;
KEYF: JB ROW4,NOKEY
      DJNZ KEYBOUNC,KEYF
      MOV KEYDATA,#0FH
      RET
NOKEY:MOV KEYDATA,#0FFH
      RET
;================================
;THE END OF KEYPAD 4X4 SUBROUTINE
;================================
END
```

9.8 Build Digital to Analog Converter Using 0808

The task is to build a simple D/A convertor and evaluate its performance. You will be using a DAC0808 (DAC0808 datasheet), Figure 9.16. Your design should take 8 bits from the

microcontroller (use the 8 8bits on Port 0 of the AT89s51 as the input to the D/A convertor). The D/A output should range from 0 to 5 volts. The lower 8 bits from the AT89s51 should go into the 8 bits from the DAC0808.

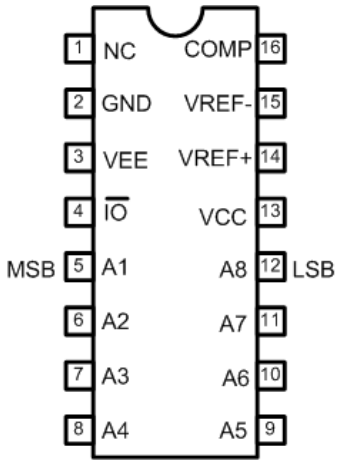


Figure 9.16: Pin Configuration DAC0808

9.9.1 Things to consider with the this design:

The pins are labeled A1 through A8, but note that A1 is the Most Significant Bit, and A8 is the Least Significant Bit (the opposite of the normal convention). Ground the two least significant bits. The D/A convertor has an output current, instead of an output voltage. The output pin should stay at about 0 volts. The op-amp on the "Typical Application" on the datasheet converts the current to a voltage, Figure 9.17.
The output current from pin 4 ranges between 0 (when the inputs are all 0) to Imax*255/256 when all the inputs are 1. The current, Imax, is determined by the current into pin 14 (which is at 0 volts).

Note: Since we are using 8 bits, the maximum value is Imax*255/256. You will need to modify the circuit given in the datasheet to get a full scale range of 0 to 5 volts. Again, our output will be just under 5 volts. The output of the D/A convertor takes some time to settle. You may need to take this in consideration when planning the timing of the A/D conversion. Check the DAC0808 datasheet for specs. The code below shows an easy way to send 8 bits to the output of the microcontroller. You should probably test your code without the D/A convertor separately to ensure that the microcontroller is behaving as you expect.

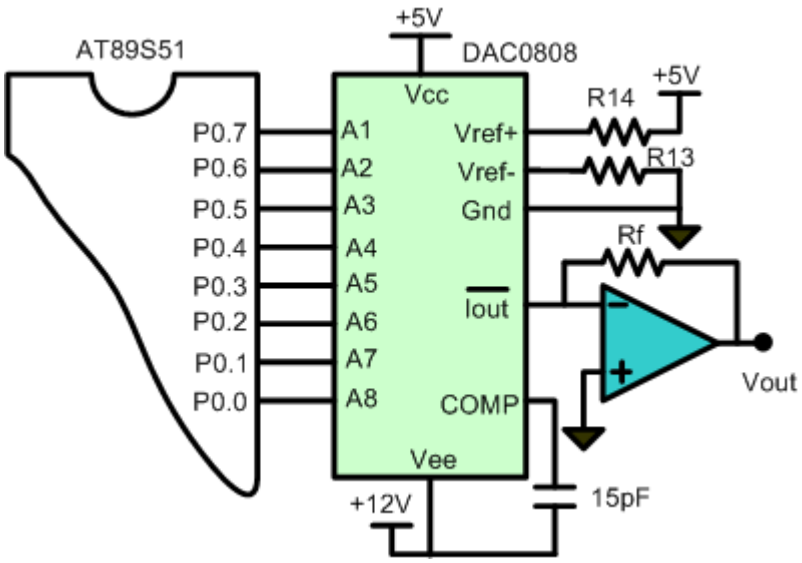


Figure 9.17: Typical Application DAC0808

$$I_o = K\left(\frac{A1}{2}+\frac{A2}{4}+\frac{A3}{8}+\frac{A4}{16}+\frac{A5}{32}+\frac{A6}{64}+\frac{A7}{128}+\frac{A8}{256}\right)$$

$$where, K \cong \frac{V_{REF}}{R14}$$

$$Vo = R_f . I_o$$

$$so, Vo = Rf . K\left(\frac{A1}{2}+\frac{A2}{4}+\frac{A3}{8}+\frac{A4}{16}+\frac{A5}{32}+\frac{A6}{64}+\frac{A7}{128}+\frac{A8}{256}\right)$$

Where, Rf = Feedback Resistor of Current to Voltage Converter circuit

9.9.1 Simple Experiment to Generate a Voltage from DAC

In this lesson we will like to design how to generate a voltage 2 volt from DAC0808. Something you must do is to calculate the constant K.

P0.0 trough P0.7 is connected to A8 - A1.

or example, we assumes that the Vref = 5 volt, R14 = 5 k, and Rf = 5k

K = Vref/ R14;

= 5/5k

= 1 mA

If, we put logic high into A1 through A8 then we have:

Io = 1 mA x 0.99

= 1 mA

So,Vo = Io x Rf

= 1 mA x 5k

= 5 volt

Voltage Resolution = 5/255 = 0.02 volt

If you like to send a 2 volt out from your DAC, than you

must write down this decimal

= 2/0.02

= 100 decimal

Now type the assembly program to make a voltage from your DAC.

```
      ORG 0H
START: MOV A,#100
      MOV P0,A
      SJMP START
```

9.9.2 Generating a Stair Step Voltage from DAC

In this example we will look at how to generate stair step voltage from 0 volt through 5 volt, with resolution 1 volt each step. Again you must calculate the constant K.

P0.0 trough P0.7 is connected to A8 - A1.

or example, we assumes that the Vref = 5 volt, R14 = 5 k, and Rf = 5k

K = Vref/ R14;

= 5/5k

= 1 mA

If, we put logic high into A1 through A8 then we have:

Io = 1 mA x 0.99

= 1 mA

So,Vo = Io x Rf

= 1 mA x 5k

= 5 volt

Voltage Resolution = 5/255 = 0.02 volt

* Voltage 0 volt

= 0/ 0.02

= 0 decimal

* Voltage 1 volt

= 1/0.02

= 50 decimal

* Voltage 2 volt

= 2/0.02

= 100 decimal

* Voltage 3 volt

= 3/0.02

= 150 decimal

* Voltage 4 volt

= 4/0.02

= 200 decimal

* Voltage 5 volt

= 5/0.02

= 255 decimal

Now, you must type the assembly program to make a voltage from your DAC.

```
      ORG 0H
START: MOV A,#0
     MOV P0,A
     CALL DELAY
;
     MOV A,#50
     MOV P0,A
     CALL DELAY
;
     MOV A,#100
     MOV P0,A
     CALL DELAY
;
     MOV A,#150
     MOV P0,A
     CALL DELAY
;
     MOV A,#200
     MOV P0,A
     CALL DELAY
;
     MOV A,#255
     MOV P0,A
     CALL DELAY
;
     SJMP START
;===========================================
;SUBROUTINE DELAY CREATED TO RISE DELAY TIME
;===========================================
```

```
DELAY: MOV R1,#255
DEL1:  MOV R2,#255
DEL2:  DJNZ R2,DEL2
       DJNZ R1,DEL1
       RET
END
```

9.10 Simulating to Display Temperature by Using Look Up Table

Look up table is just couple data that are saved in ROM and can be manipulated. In this case we are going to display temperature with three digit and one digit after point.

Temp = DataADC * 100/ 255 ^{o}C.

Note : DataADC is data that came from ADC, and have range 0-255. In this simple task, we assume that we are using temperature range from 0 - 100 ^{o}C, so you just divide 100 by 255 as you look in Table 9.3.

Table 9.3: Temperature Data

Decimal	Temp.(oC)	Hundreds	Tens	Ones	Fraction
0	0	0	0	0	0
1	0.4	0	0	0	4
2	0.8	0	0	0	8
3	1.2	0	0	1	2
4	1.6	0	0	1	6
5	2.0	0	0	2	0
6	2.4	0	0	2	4
7	2.7	0	0	2	7
8	3.1	0	0	3	1
9	3.5	0	0	3	5
10	3.9	0	0	3	9
:	:	:	:	:	:
:	:	:	:	:	:
:	:	:	:	:	:
243	95.3	0	9	5	3
244	95.7	0	9	5	7
245	96.1	0	9	6	1
246	96.5	0	9	6	5
247	96.9	0	9	6	9
248	97.3	0	9	7	3
249	97.6	0	9	7	6
250	98.0	0	9	8	0
251	98.4	0	9	8	4
252	98.8	0	9	8	8
253	99.2	0	9	9	2
254	99.6	0	9	9	6
255	100.0	1	0	0	0

After you have made the table by using Microsoft Excel, you should write down the data in the look up table:

fraction :
db 0,4,8,2,6,0,4,7,1,5,9,3,7,1,5,9,3,7,1,5,8,2,6,0,4
db 8,2,6,0,4,8,2,5,9,3,7,1,5,9,3,7,1,5,9,3,6,0,4,8,2,6
db 0,4,8,2,6,0,4,7,1,5,9,3,7,1,5,9,3,7,1,5,8,2,6,0
db 4,8,2,6,0,4,8,2,5,9,3,7,1,5,9,3,7,1,5,9,3,6,0,4,8,2
db 6,0,4,8,2,6,0,4,7,1,5,9,3,7,1,5,9,3,7,1,5,8,2,6
db 0,4,8,2,6,0,4,8,2,5,9,3,7,1,5,9,3,7,1,5,9,3,6,0,4,8
db 2,6,0,4,8,2,6,0,4,7,1,5,9,3,7,1,5,9,3,7,1,5,8,2
db 6,0,4,8,2,6,0,4,8,2,5,9,3,7,1,5,9,3,7,1,5,9,3,6,0,4
db 8,2,6,0,4,8,2,6,0,4,7,1,5,9,3,7,1,5,9,3,7,1,5,8,2,6
db 0,4,8,2,6,0,4,8,2,5,9,3,7,1,5,9,3,7,1,5,9,3,6,0,4,8,2,6,0
;
Ones :
db 0,0,0,1,1,2,2,2,3,3,3,4,4,5,5,5,6,6,7,7,7,8,8,9,9
db 9,0,0,1,1,1,2,2,2,3,3,4,4,4,5,5,6,6,6,7,7,8,8,8,9,9
db 0,0,0,1,1,2,2,2,3,3,3,4,4,5,5,5,6,6,7,7,7,8,8,9
db 9,9,0,0,1,1,1,2,2,2,3,3,4,4,4,5,5,6,6,6,7,7,8,8,8,9
db 9,0,0,0,1,1,2,2,2,3,3,3,4,4,5,5,5,6,6,7,7,7,8,8,9
db 9,9,0,0,1,1,1,2,2,2,3,3,4,4,4,5,5,6,6,6,7,7,8,8,8
db 9,9,0,0,0,1,1,2,2,2,3,3,3,4,4,5,5,5,6,6,7,7,7,8,8

```
db 9,9,9,0,0,1,1,1,2,2,2,3,3,4,4,4,5,5,6,6,6,7,7,8,8
db 8,9,9,0,0,0,1,1,2,2,2,3,3,3,4,4,5,5,5,6,6,7,7,7,8,8
db 9,9,9,0,0,1,1,1,2,2,2,3,3,4,4,4,5,5,6,6,6,7,7,8,8,8,9,9,0
;
Tens :
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
db 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2
db 2,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3
db 3,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4
db 4,4,4,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5
db 5,5,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
db 6,6,6,6,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7
db 7,7,7,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8
db 8,8,8,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,0
;
Hundreds :
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1
```

our plan is display the data to look like for example:

8th	**7th**	**6th**	**5th**	**4th**	**3rd**	**2nd**	**1st**
	Hundr.	Tens	Ones		Fraction		
	1	**0**	**0**	**,**	**0**	**º**	**C**

9.10.1 Generating a Sine Wave Voltage by Using Look Up Table

In this lesson we will like to design how to generate sine wave voltage from 0 Volt through 5 volt, with resolution 0.1 volt each step. Again first calculate the constant K. For this lesson something that gives you a clue is given in Figure 9.18.

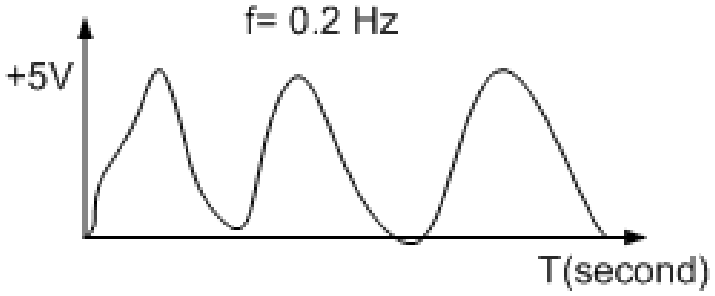


Figure 9.18: Sine wave voltage 0 - 5 volt

APPENDIX I: PAST EXAMINATIONS QUESTIONS

DEPARTMENT OF ELECTRICAL/ELECTRONIC ENGINEERING
(COMPUTER ENGINEERING)
UNIVERSITY OF BENIN, BENIN CITY
1st SEMESTER EXAMINATIONS 2010/2011 SESSION
02nd June, 2011

CPE575: MICROPROGRAMMING **Time: 3 hours**

INSTRUCTION: ATTEMPT ANY FIVE (5) QUESTIONS ONLY

1a. State the rules that govern how interrupt priorities work. (4 Marks)

b. Two sets of thirty-two (32) numbers are stored in consecutive locations starting at addresses 30H and 60H respectively. Write a program that compares corresponding numbers and sounds a 1 second alarm on P2.0 if the numbers are equal. *(10 Marks)*

2a. With the aid of suitable sketch(es) explain the serial port buffer register. *(6 Marks)*

b. Access to the circuitry around the CPU of a microcomputer is provided by the buses.

i. Explain the concept of "bus" as used with microcomputers

ii. List and explain these buses clearly highlighting, where applicable, their uses in microcomputer definitions. *(8 Marks)*

3a. Consider the code program Qn3a

i. Attach addressing modes to each instruction

ii. Describe how instructions for the addressing modes listed for Qn3ai are generally coded. *(7 Marks)*

b. An interrupt-driven-system gives an illusion of doing many things simultaneously. Explain *(7 Marks)*

4ai. Add comments to program Qn4a

ii. Explain how the program will work when executed *(10 Marks)*

b. Describe the structure of the first 32 bytes of the 8051 internal RAM. *(4 Marks)*

5a. State and explain the clocking sources for the 8051 internal timers. *(7 Marks)*

b. Explain:

i. The concept of polling in microcontrollers

ii. Polling sequence *(7 Marks)*

6a. Write an instruction sequence to initialize the 8051 serial port to operate as in the 8-bit UART mode at 19200 baud using Timer 1 as the baud rate clock. *(5 Marks)*

b. Briefly but satisfactorily explain "stack" handling in the 8051 microcontroller. *(5 Marks)*

c. Outline the sequence of events that occur when an interrupt is triggered *(4 Marks)*

7a. Write a program that generates pulses on p2.6 with 60% duty cycle *(10 Marks)*

b. Write equivalent bit instruction(s) for the instruction: MOV 2BH,#OFFH. *(4 Marks)*

PROGRAM QN3a

```
MOV R0,#60H
MOV @R0, A
INC R0
.
.
```

PROGRAM QN4a

```
0000            5               ORG O
0000  020030    6               LJMP MAIN
0023            7               ORG  0023H
0023  020042    8               LJMP SPISR
0030            9               ORG 0030H
0030  758920    10    MAIN:     MOV TMOD, #02H
```

```
0033  758DE6   11
0036  D28E     12                  SETB TR0
0038  759842   13                  MOV SCON, #42H
003B  7420     14                  MOV A, #30H
003D  75A890   15                  MOV IE, 90H
0040  80FE     16                  SJMP $
               17        ,
0042  B47F02   18        SPISR:    CJNE A, #7FH, SKIP
0045  7420     19                  MOV A, #30H
0047  F599     20        SKIP:     MOV SBUF, A
0049  04       21                  INC A
004A  C299     22                  CLR TI
004C  32       23                  RETI
               24                  END
```

GoodLuck!!!

DEPARTMENT OF ELECTRICAL/ELECTRONIC ENGINEERING (COMPUTER ENGINEERING)

UNIVERSITY OF BENIN, BENIN CITY

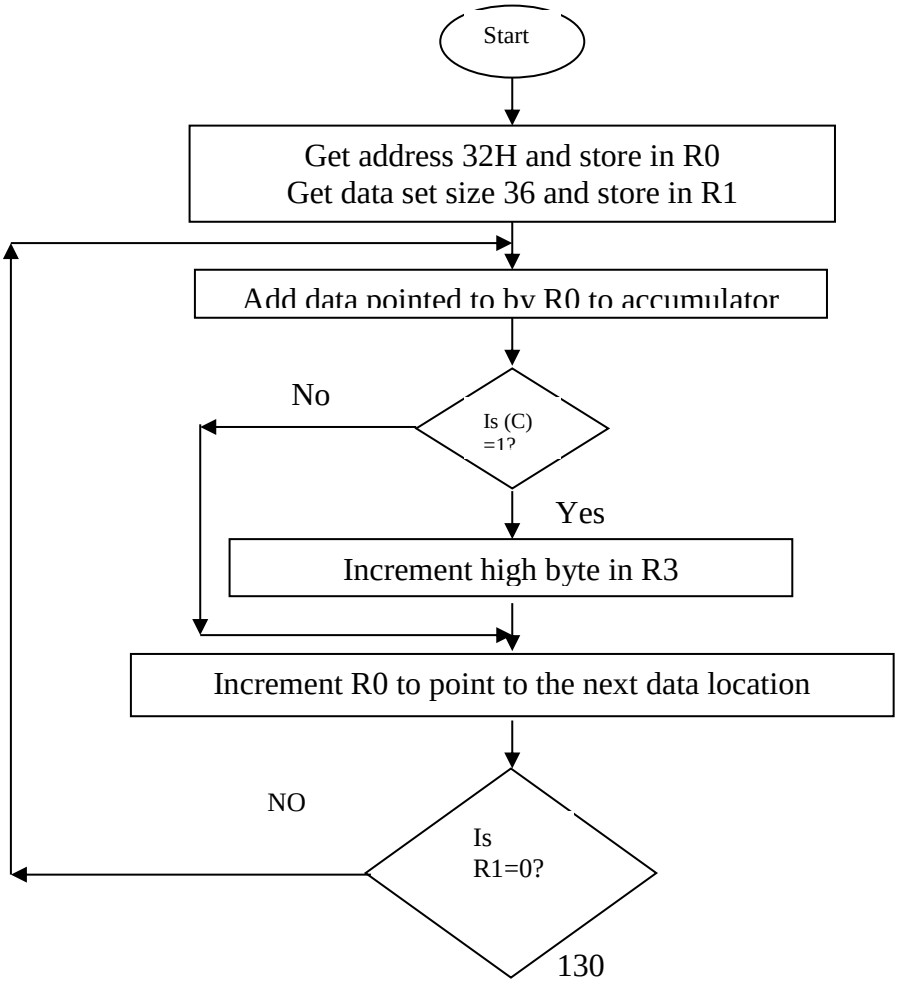
1st SEMESTER EXAMINATIONS 2009/2010 SESSION

3rd August, 2010

CPE575: MICROPROGRAMMING **Time: 3hours**

INSTRUCTION: ATTEMPT ANY <u>FIVE 5</u> QUESTIONS ONLY

1a. Consider the flowchart Figure Qn1a:



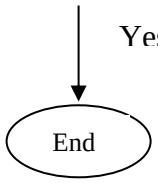


Figure Qn1a.

i. Write an assembly language program to implement this flowchart on the 8051 microcontroller

ii. What does this program do? *(8 Marks)*

b List and briefly explain the uses of the 8051 timers. *(6 Marks)*

2a. Ten (10) numbers are stored in the 8051 internal RAM starting at address 4FH. Assuming that their sum will not overflow the accumulator, write a program using indirect addressing to find the average of the numbers. (*8 Marks)*

bi. Sketch TMOD register showing Timer 0 set to Mode 3 for event counting and Timer 1 set to Mode 2 for interval timing. *(2 Marks)*

ii. Briefly but satisfactorily, explain the 8051 Timer Mode 3. *(4 Marks)*

3a. Briefly explain and clearly distinguishing between the 8051's serial port operating Modes 2 and 3. *(6 Marks)*

b. Write a program for 8051 to relocate a block of data stored in sixteen (16) internal RAM locations starting at address 32H to another block in RAM starting at address 60H. *(8 Marks)*

4a. Ten (10) numbers are stored in the 8051's internal RAM consecutively starting at RAM location 64H. Write an assembly language program that sequentially examines these numbers and triggers an alarm connected to P1.7 for 0.5s whenever any of the numbers has even parity. *(10 Marks)*

b. Without any sketch, briefly give the hardware overview of the 8051 microcontroller.*(4 Marks)*

5a. Write a program that continually transmits characters from a transmit buffer (internal RAM 40H to 4FH). If the incoming characters are detected on the serial port, store them in the receive buffer starting at internal RAM location 60H. Assume that the 8051 serial port has already been initialized in Mode 1. *(8 Marks)* [**Hint**: This is full duplex serial port operation]

b. Describe the 8051 coding for:

i. Register addressing

ii. Register indirect addressing *(6 Marks)*

6a. Write a program that sends the texts 'abcdefgh' in ASCII from down the serial port line at a baud rate of 9600 with a system clock frequency of 12MHz. Use serial port Mode 1 with even parity and assume that the text characters will first be stored in internal RAM locations starting at address 40H. *(10 Marks)*

b. Describe the functionality of the 8051's general purpose registers. *(4 Marks)*

7a. Write a program that generates 100Hz on P1.0 of the 8051. *(9 Marks)*

b. Describe the 8051 bit memory address space. *(5 Marks)*

GOODLUCK!!!!!

DEPARTMENT OF ELECTRICAL/ELECTRONIC ENGINEERING(COMPUTER ENGINEERING)

UNIVERSITY OF BENIN, BENIN CITY

1st SEMESTER EXAMINATIONS 2008/2009 SESSION

2TH June, 2009

CPE575: MICROPROGRAMMING **Time: 3hours**

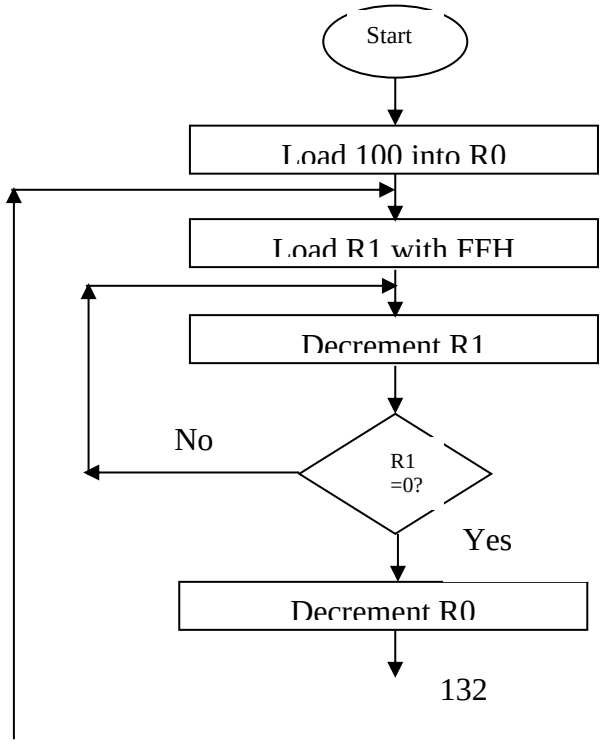
INSTRUCTION: ATTEMPT ANY <u>FIVE 5</u> QUESTIONS ONLY

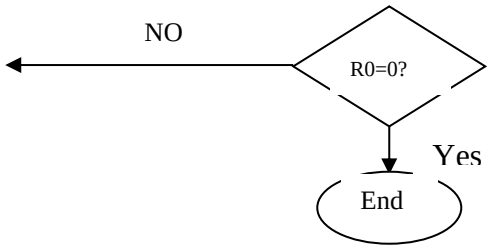
1a. A security lamp connected to P1.7 of the Intel 8051 micro controller and a debounced switch is connect to P1.6. Write a program that reads the logic level provided by the switch and causes the lamp to flash five (5) times with each flash lasting for ten (10) seconds.

bi. Describe the program status word of the 8051

ii. Clearly distinguish between the PSW.7 and PSW.2

2a. Consider the flowchart below:





iii. Write an assembly language program to implement this flowchart on the AT8951 microcontroller .

iv. What does this program do?

b Consider the instruction : MOV TMOD, #17H.

i. Describe the effect of the instruction .

ii. Explain the functionality of the effect of the instruction.

3a. A computer monitors an industrial process and is used to find the average of a relatively large number of temperature and pressure reading . The temperature readings are stored as one block of data in external rate memory starting at address 2000H while the pressure readings are also shred as a block of data in external data memory starting at address 3000H. Assuming that a hundred readings each are taken for both temperature and pressure ,write a program that calculate both averages. (Hint: Use one subroutine for both averages and assume that the readings are values above a minimum below which no readings fall below so that the sum does not overflow the accumulator)

b. Consider the following instructions:

i. MOV,@Ri

ii. MOVC A,@A + PC

v. XCH A, Rn

vi. MOVX @ A + DPTR, A

I. Attach addressing mode to each instruction

II. Explain what each instruction will do when executed

4a. Sixteen signed binary numbers are stored in the 8051's internal RAM consecutively starting at RAM location 30H. Write an assembly language program that sequentially examines these numbers and buzzes for 5s whenever any of the numbers is a negative number.

c. What do you understand by relative address

d. Using memory sketch, explain what happens when the following instructions are executed:

i. SJMP 08H

ii. SJMP 82H

iii. SJMP E6H

iv. SJMP FFH

Assume that the PC contained 2000H before each instruction was executed.

5a. Fully describe the bit address space of the 8051

c. Distinguish interval timing and event counter

d. Describe indirect addressing encoding for the 8051

6ai. Manually multiply 3620H by 2dH clearly outlining each step of the multiplication

ii. Assemble these steps in a table to show the composite steps of the multiplication process

iii. Outline the steps for the process

iv. Hence, write a program to implement the multiplication

bi. What are the key distinguishing features between microprocessors and microcontrollers?

ii. What design objective(s) necessitated these differences?

7a. Sketch the 8051 chip(detailed pin assignment not necessary) and explain its interface features.

b. Assume that data values are already loaded into the consecutive locations 18H through 1DH.

i. Write a program using direct addressing to sum the numbers

ii. Write a program using indirect addressing to sum the numbers.

GOODLUCK!!!!!

DEPARTMENT OF ELECTRICAL/ELECTRONIC ENGINEERING
(COMPUTER ENGINEERING)
UNIVERSITY OF BENIN, BENIN CITY
1st SEMESTER EXAMINATIONS 2010/2011 SESSION
11TH MAY, 2011

CPE575: MICROPROGRAMMING **Time: 2 hours**

INSTRUCTION: ATTEMPT ALL QUESTIONS

1. Assume that data values are already loaded into the consecutive locations 30H through 3FH of internal RAM.

i. Write a program using direct addressing to sum the numbers

ii. Write a program using indirect addressing to sum the numbers.

iii. Make a relevant comparative comment on the programs you have written

[Hint: Assume possibility of accumulator overflow]

2. Sixteen signed binary numbers are stored in the 8051's internal RAM consecutively starting at RAM location 40H. Write an assembly language program that sequentially examines these numbers and sounds an alert alarm on p1.1 for 1s whenever any of the numbers is a negative number.

3. Write a program that sends the texts 'abcde' in ASCII from down the serial port line at a baud rate of 12000 with a system clock frequency of 12MHz. Use serial port Mode 1 with even parity and assume that the text characters will first be stored in internal RAM locations starting at address 40H.

4a. Outline the sequence of events that occur when an interrupt is triggered

b. State the rules that govern how interrupt priorities work.

5. Distinguish between microprocessors and microcontrollers

GoodLuck!!!

DEPARTMENT OF ELECTRICAL/ELECTRONIC ENGINEERING
(COMPUTER ENGINEERING)
UNIVERSITY OF BENIN, BENIN CITY
1st SEMESTER EXAMINATIONS 2010/2011 SESSION
27TH MAY, 2011

CPE575: MICROPROGRAMMING **Time: 1 1/2 hours**

INSTRUCTION: ATTEMPT ALL QUESTIONS

1. Sixteen (16) numbers are stored in the 8051 internal RAM starting at address 50H. Assuming that their sum will not overflow the accumulator, write a program using indirect addressing to find the average of the numbers.
2. Sketch TMOD register showing Timer 0 set to Mode 3 for event counting and Timer 1 set to Mode 2 for interval timing.
3. Briefly but satisfactorily, explain the 8051 Timer Mode 3.
4. Write a program that sends the texts ‘abcde’ in ASCII from down the serial port line at a baud rate of 96000 with a system clock frequency of 12MHz. Use serial port Mode 1 with even parity and assume that the text characters will first be stored in internal RAM locations starting at address 40H.

5a. Outline the sequence of events that occur when an interrupt is triggered

b. State the rules that govern how interrupt priorities work.

6. Distinguish between microprocessors and microcontrollers

Appendix II:
==========

8051 Instructions in numerical Order

Abbreviations: direct = 8-bit DATA address in internal memory
const8 = 8-bit constant in CODE memory
const16 = 16-bit constant in CODE memory
addr16 = 16-bit long CODE address
addr11 = 11-bit absolute CODE address
rel = signed 8-bit relative CODE address
bit = 8-bit BIT address in internal memory

Opcode	Mnemonic	Operands	Bytes	Flags		Cycles
00	NOP		1			1
01	AJMP	addr11	2			2
02	LJMP	addr16	3			2
03	RR	A	1			1
04	INC	A	1		P	1
05	INC	direct	2			1
06	INC	@R0	1			1
07	INC	@R1	1			1
08	INC	R0	1			1
09	INC	R1	1			1
0A	INC	R2	1			1
0B	INC	R3	1			1
0C	INC	R4	1			1
0D	INC	R5	1			1
0E	INC	R6	1			1
0F	INC	R7	1			1
10	JBC	bit, rel	3			2
11	ACALL	addr11	2			2
12	LCALL	addr16	3			2
13	RRC	A	1	CY	P	1
14	DEC	A	1		P	1
15	DEC	direct	2			1
16	DEC	@R0	1			1
17	DEC	@R1	1			1
18	DEC	R0	1			1
19	DEC	R1	1			1
1A	DEC	R2	1			1
1B	DEC	R3	1			1

1C	DEC	R4	1		1
1D	DEC	R5	1		1
1E	DEC	R6	1		1
1F	DEC	R7	1		1
20	JB	bit, rel	3		2
21	AJMP	addr11	2		2
22	RET		1		2
23	RL	A	1		1
24	ADD	A, #const8	2	CY AC OV P	1
25	ADD	A, direct	2	CY AC OV P	1
26	ADD	A, @R0	1	CY AC OV P	1
27	ADD	A, @R1	1	CY AC OV P	1

Opcode	Mnemonic	Operands	Bytes	Flags	Cycles
28	ADD	A, R0	1	CY AC OV P	1
29	ADD	A, R1	1	CY AC OV P	1
2A	ADD	A, R2	1	CY AC OV P	1
2B	ADD	A, R3	1	CY AC OV P	1
2C	ADD	A, R4	1	CY AC OV P	1
2D	ADD	A, R5	1	CY AC OV P	1
2E	ADD	A, R6	1	CY AC OV P	1
2F	ADD	A, R7	1	CY AC OV P	1
30	JNB	bit, rel	3		2
31	ACALL	addr11	2		2
32	RETI		1		2
33	RLC	A	1	CY P	1
34	ADDC	A, #const8	2	CY AC OV P	1
35	ADDC	A, direct	2	CY AC OV P	1
36	ADDC	A, @R0	1	CY AC OV P	1
37	ADDC	A, @R1	1	CY AC OV P	1
38	ADDC	A, R0	1	CY AC OV P	1
39	ADDC	A, R1	1	CY AC OV P	1
3A	ADDC	A, R2	1	CY AC OV P	1
3B	ADDC	A, R3	1	CY AC OV P	1
3C	ADDC	A, R4	1	CY AC OV P	1
3D	ADDC	A, R5	1	CY AC OV P	1
3E	ADDC	A, R6	1	CY AC OV P	1
3F	ADDC	A, R7	1	CY AC OV P	1
40	JC	rel	2		2
41	AJMP	addr11	2		2
42	ORL	direct, A	2		1
43	ORL	direct, #const8	3		2
44	ORL	A, #const8	2	P	1
45	ORL	A, direct	2	P	1
46	ORL	A, @R0	1	P	1
47	ORL	A, @R1	1	P	1
48	ORL	A, R0	1	P	1
49	ORL	A, R1	1	P	1
4A	ORL	A, R2	1	P	1
4B	ORL	A, R3	1	P	1
4C	ORL	A, R4	1	P	1
4D	ORL	A, R5	1	P	1
4E	ORL	A, R6	1	P	1
4F	ORL	A, R7	1	P	1
50	JNC	rel	2		2
51	ACALL	addr11	2		2
52	ANL	direct, A	2		1
53	ANL	direct, #const8	3		2
54	ANL	A, #const8	2	P	1
55	ANL	A, direct	2	P	1
56	ANL	A, @R0	1	P	1
57	ANL	A, @R1	1	P	1
58	ANL	A, R0	1	P	1
59	ANL	A, R1	1	P	1

```
5A      ANL     A, R2                   1               P       1
5B      ANL     A, R3                   1               P       1
5C      ANL     A, R4                   1               P       1
5D      ANL     A, R5                   1               P       1
5E      ANL     A, R6                   1               P       1
5F      ANL     A, R7                   1               P       1
```

Opcode	Mnemonic	Operands	Bytes	Flags	Cycles
60	JZ	rel	2		2
61	AJMP	addr11	2		2
62	XRL	direct, A	2		1
63	XRL	direct, #const8	3		2
64	XRL	A, #const8	2	P	1
65	XRL	A, direct	2	P	1
66	XRL	A, @R0	1	P	1
67	XRL	A, @R1	1	P	1
68	XRL	A, R0	1	P	1
69	XRL	A, R1	1	P	1
6A	XRL	A, R2	1	P	1
6B	XRL	A, R3	1	P	1
6C	XRL	A, R4	1	P	1
6D	XRL	A, R5	1	P	1
6E	XRL	A, R6	1	P	1
6F	XRL	A, R7	1	P	1
70	JNZ	rel	2		2
71	ACALL	addr11	2		2
72	ORL	C, bit	2	CY	2
73	JMP	@A+DPTR	1		2
74	MOV	A, #const8	2	P	1
75	MOV	direct, #const8	3		2
76	MOV	@R0, #const8	2		1
77	MOV	@R1, #const8	2		1
78	MOV	R0, #const8	2		1
79	MOV	R1, #const8	2		1
7A	MOV	R2, #const8	2		1
7B	MOV	R3, #const8	2		1
7C	MOV	R4, #const8	2		1
7D	MOV	R5, #const8	2		1
7E	MOV	R6, #const8	2		1
7F	MOV	R7, #const8	2		1
80	SJMP	rel	2		2
81	AJMP	addr11	2		2
82	ANL	C, bit	2	CY	2
83	MOVC	A, @A+PC	1	P	2
84	DIV	AB	1	CY OV P	4
85	MOV	direct, direct	3		2
86	MOV	direct, @R0	2		2
87	MOV	direct, @R1	2		2
88	MOV	direct, R0	2		2
89	MOV	direct, R1	2		2
8A	MOV	direct, R2	2		2
8B	MOV	direct, R3	2		2
8C	MOV	direct, R4	2		2
8D	MOV	direct, R5	2		2
8E	MOV	direct, R6	2		2
8F	MOV	direct, R7	2		2
90	MOV	DPTR, #const16	3		2
91	ACALL	addr11	2		2

92	MOV	bit, C	2						2
93	MOVC	A, @A+DPTR	1					P	2
94	SUBB	A, #const8	2	CY	AC	OV		P	1
95	SUBB	A, direct	2	CY	AC	OV		P	1
96	SUBB	A, @R0	1	CY	AC	OV		P	1
97	SUBB	A, @R1	1	CY	AC	OV		P	1

Opcode	Mnemonic	Operands	Bytes	Flags	Cycles
98	SUBB	A, R0	1	CY AC OV P	1
99	SUBB	A, R1	1	CY AC OV P	1
9A	SUBB	A, R2	1	CY AC OV P	1
9B	SUBB	A, R3	1	CY AC OV P	1
9C	SUBB	A, R4	1	CY AC OV P	1
9D	SUBB	A, R5	1	CY AC OV P	1
9E	SUBB	A, R6	1	CY AC OV P	1
9F	SUBB	A, R7	1	CY AC OV P	1
A0	ORL	C, /bit	2	CY	2
A1	AJMP	addr11	2		2
A2	MOV	C, bit	2	CY	1
A3	INC	DPTR	1		2
A4	MUL	AB	1	CY OV P	4
A5	illegal	opcode			
A6	MOV	@R0, direct	2		2
A7	MOV	@R1, direct	2		2
A8	MOV	R0, direct	2		2
A9	MOV	R1, direct	2		2
AA	MOV	R2, direct	2		2
AB	MOV	R3, direct	2		2
AC	MOV	R4, direct	2		2
AD	MOV	R5, direct	2		2
AE	MOV	R6, direct	2		2
AF	MOV	R7, direct	2		2
B0	ANL	C, /bit	2	CY	2
B1	ACALL	addr11	2		2
B2	CPL	bit	2		1
B3	CPL	C	1	CY	1
B4	CJNE	A, #const8, rel	3	CY	2
B5	CJNE	A, direct, rel	3	CY	2
B6	CJNE	@R0, #const8, rel	3	CY	2
B7	CJNE	@R1, #const8, rel	3	CY	2
B8	CJNE	R0, #const8, rel	3	CY	2
B9	CJNE	R1, #const8, rel	3	CY	2
BA	CJNE	R2, #const8, rel	3	CY	2
BB	CJNE	R3, #const8, rel	3	CY	2
BC	CJNE	R4, #const8, rel	3	CY	2
BD	CJNE	R5, #const8, rel	3	CY	2
BE	CJNE	R6, #const8, rel	3	CY	2
BF	CJNE	R7, #const8, rel	3	CY	2
C0	PUSH	direct	2		2
C1	AJMP	addr11	2		2
C2	CLR	bit	2		1
C3	CLR	C	1	CY	1
C4	SWAP	A	1		1
C5	XCH	A, direct	2	P	1
C6	XCH	A, @R0	1	P	1
C7	XCH	A, @R1	1	P	1
C8	XCH	A, R0	1	P	1
C9	XCH	A, R1	1	P	1

CA	XCH	A, R2	1	P	1
CB	XCH	A, R3	1	P	1
CC	XCH	A, R4	1	P	1
CD	XCH	A, R5	1	P	1
CE	XCH	A, R6	1	P	1
CF	XCH	A, R7	1	P	1

Opcode	Mnemonic	Operands	Bytes	Flags	Cycles
D0	POP	direct	2		2
D1	ACALL	addr11	2		2
D2	SETB	bit	2		1
D3	SETB	C	1	CY	1
D4	DA	A	1	CY P	1
D5	DJNZ	direct, rel	3		2
D6	XCHD	A, @R0	1	P	1
D7	XCHD	A, @R1	1	P	1
D8	DJNZ	R0, rel	2		2
D9	DJNZ	R1, rel	2		2
DA	DJNZ	R2, rel	2		2
DB	DJNZ	R3, rel	2		2
DC	DJNZ	R4, rel	2		2
DD	DJNZ	R5, rel	2		2
DE	DJNZ	R6, rel	2		2
DF	DJNZ	R7, rel	2		2
E0	MOVX	A, @DPTR	1	P	2
E1	AJMP	addr11	2		2
E2	MOVX	A, @R0	1	P	2
E3	MOVX	A, @R1	1	P	2
E4	CLR	A	1	P	1
E5	MOV	A, direct	2	P	1
E6	MOV	A, @R0	1	P	1
E7	MOV	A, @R1	1	P	1
E8	MOV	A, R0	1	P	1
E9	MOV	A, R1	1	P	1
EA	MOV	A, R2	1	P	1
EB	MOV	A, R3	1	P	1
EC	MOV	A, R4	1	P	1
ED	MOV	A, R5	1	P	1
EE	MOV	A, R6	1	P	1
EF	MOV	A, R7	1	P	1
F0	MOVX	@DPTR, A	1		2
F1	ACALL	addr11	2		2
F2	MOVX	@R0, A	1		2
F3	MOVX	@R1, A	1		2
F4	CPL	A	1	P	1
F5	MOV	direct, A	2		1
F6	MOV	@R0, A	1		1
F7	MOV	@R1, A	1		1
F8	MOV	R0, A	1		1
F9	MOV	R1, A	1		1
FA	MOV	R2, A	1		1
FB	MOV	R3, A	1		1
FC	MOV	R4, A	1		1
FD	MOV	R5, A	1		1
FE	MOV	R6, A	1		1
FF	MOV	R7, A	1		1

Appendix J:

8051 Instructions in lexical Order

Abbreviations:
- direct = 8-bit DATA address in internal memory
- const8 = 8-bit constant in CODE memory
- const16 = 16-bit constant in CODE memory
- addr16 = 16-bit long CODE address
- addr11 = 11-bit absolute CODE address
- rel = signed 8-bit relative CODE address
- bit = 8-bit BIT address in internal memory

- i = register numbers 0 or 1
- n = register numbers 0 thru 7
- a = 32 * m
- m = the 3 most significant bits of an absolute address

Opcode	Mnemonic	Operands	Bytes	Flags	Cycles
11+a	ACALL	addr11	2		2
24	ADD	A, #const8	2	CY AC OV P	1
26+i	ADD	A, @Ri	1	CY AC OV P	1
25	ADD	A, direct	2	CY AC OV P	1
28+n	ADD	A, Rn	1	CY AC OV P	1
34	ADDC	A, #const8	2	CY AC OV P	1
36+i	ADDC	A, @Ri	1	CY AC OV P	1
35	ADDC	A, direct	2	CY AC OV P	1
38+n	ADDC	A, Rn	1	CY AC OV P	1
01+a	AJMP	addr11	2		2
54	ANL	A, #const8	2	P	1
56+i	ANL	A, @Ri	1	P	1
55	ANL	A, direct	2	P	1
58+n	ANL	A, Rn	1	P	1
B0	ANL	C, /bit	2	CY	2
82	ANL	C, bit	2	CY	2
53	ANL	direct, #const8	3		2
52	ANL	direct, A	2		1
B6+i	CJNE	@Ri, #const8, rel	3	CY	2
B4	CJNE	A, #const8, rel	3	CY	2
B5	CJNE	A, direct, rel	3	CY	2
B8+n	CJNE	Rn, #const8, rel	3	CY	2
E4	CLR	A	1	P	1
C2	CLR	bit	2		1
C3	CLR	C	1	CY	1

F4	CPL	A	1			P	1
B2	CPL	bit	2				1
B3	CPL	C	1	CY			1
D4	DA	A	1	CY		P	1
16+i	DEC	@Ri	1				1
14	DEC	A	1			P	1
15	DEC	direct	2				1
18+n	DEC	Rn	1				1
84	DIV	AB	1	CY	OV	P	4
D5	DJNZ	direct, rel	3				2

Opcode	Mnemonic	Operands	Bytes	Flags	Cycles
D8+n	DJNZ	Rn, rel	2		2
06+i	INC	@Ri	1		1
04	INC	A	1	P	1
05	INC	direct	2		1
A3	INC	DPTR	1		2
08+n	INC	Rn	1		1
20	JB	bit, rel	3		2
10	JBC	bit, rel	3		2
40	JC	rel	2		2
73	JMP	@A+DPTR	1		2
30	JNB	bit, rel	3		2
50	JNC	rel	2		2
70	JNZ	rel	2		2
60	JZ	rel	2		2
12	LCALL	addr16	3		2
02	LJMP	addr16	3		2
76+i	MOV	@Ri, #const8	2		1
F6+i	MOV	@Ri, A	1		1
A6+i	MOV	@Ri, direct	2		2
74	MOV	A, #const8	2	P	1
E6+i	MOV	A, @Ri	1	P	1
E5	MOV	A, direct	2	P	1
E8+n	MOV	A, Rn	1	P	1
92	MOV	bit, C	2		2
A2	MOV	C, bit	2	CY	1
75	MOV	direct, #const8	3		2
86+i	MOV	direct, @Ri	2		2
F5	MOV	direct, A	2		1
85	MOV	direct, direct	3		2
88+n	MOV	direct, Rn	2		2
90	MOV	DPTR, #const16	3		2
78+n	MOV	Rn, #const8	2		1
F8+n	MOV	Rn, A	1		1
A8+n	MOV	Rn, direct	2		2
93	MOVC	A, @A+DPTR	1	P	2
83	MOVC	A, @A+PC	1	P	2
F0	MOVX	@DPTR, A	1		2
F2+i	MOVX	@Ri, A	1		2
E0	MOVX	A, @DPTR	1	P	2
E2+i	MOVX	A, @Ri	1	P	2
A4	MUL	AB	1	CY OV P	4
00	NOP		1		1
44	ORL	A, #const8	2	P	1
46+i	ORL	A, @Ri	1	P	1
45	ORL	A, direct	2	P	1
48+n	ORL	A, Rn	1	P	1
A0	ORL	C, /bit	2	CY	2
72	ORL	C, bit	2	CY	2
43	ORL	direct, #const8	3		2
42	ORL	direct, A	2		1

D0	POP	direct	2			2
C0	PUSH	direct	2			2
22	RET		1			2
32	RETI		1			2
23	RL	A	1			1
33	RLC	A	1	CY	P	1

Opcode	Mnemonic	Operands	Bytes	Flags	Cycles
03	RR	A	1		1
13	RRC	A	1	CY P	1
D2	SETB	bit	2		1
D3	SETB	C	1	CY	1
80	SJMP	rel	2		2
94	SUBB	A, #const8	2	CY AC OV P	1
96+i	SUBB	A, @Ri	1	CY AC OV P	1
95	SUBB	A, direct	2	CY AC OV P	1
98+n	SUBB	A, Rn	1	CY AC OV P	1
C4	SWAP	A	1		1
C6+i	XCH	A, @Ri	1	P	1
C5	XCH	A, direct	2	P	1
C8+n	XCH	A, Rn	1	P	1
D6+i	XCHD	A, @Ri	1	P	1
64	XRL	A, #const8	2	P	1
66+i	XRL	A, @Ri	1	P	1
65	XRL	A, direct	2	P	1
68+n	XRL	A, Rn	1	P	1
63	XRL	direct, #const8	3		2