ELE 414 Microprocessors and Digital Logic

Number systems, Codes and coding, Minimization techniques applied to design of logic systems, Component specifications, Discussion of microprocessors, Memory and I/O logic elements, Microcomputer structure and operation, I/O modes and interfacing, Machine language and assembly programming, Design and application of digital systems for data collection and control of pneumatic, hydraulic, and machine systems.

Instructor: Dr. Salah Gad Foda

Prerequisite(s): ELE 413

Lecture Hours: 3 **Exercise/Lab Hours:** 2

Textbook: M.A. Mazidi, J.G. Mazidi, and R.D. McKinlay, *The 8051 Microcontroller and Embedded Systems* , Prentice Hall.

Topics Covered:

1. Introduction to computing and microprocessors
2. The 8051 assembly language programming
3. Control instructions and I/O port programming
4. Addressing modes
5. Arithmetic and logical instructions
6. 8051 timer programming
7. 8051 serial port programming
8. Interrupts in 8051
9. 8051 interfacing
10. Applications

Evaluation:

Midterm tests* and projects	40%
Assignments and class participation	20%
Final Examination	40%
Total	100%

* On 6th and 11th weeks of the term.

Chapter I

Introduction to computing and microprocessors

Decimal and Binary Numbers

It is possible that our counting is based on decimal digits due to the fact that we have ten fingers, but, in general, we can express any number in terms of a general base d. In this case, our numbers will be written as

$$\cdots + a_3 \times d^3 + a_2 \times d^2 + a_1 \times d^1 + a_0 \times d^0$$

where $a_{0,}$ $a_{1,}$ etc., are a number between 0 and d-1. A short form of writing our number would be $a_3\ a_2\ a_1\ a_0$ where the weight of each digit is implied by its relative position. For example, the number $(23)_8 = 2\times8^1 + 3\times8^0$ which is 19 in decimal. Counting to the base 8 is called *octal*. The same number to the base 16 (*Hexadecimal* counting) would be $(23)_{16} = 2\times16^1 + 3\times16^0$ which is 35 in decimal.

However, in digital systems we have two possible states, a *HIGH* and a *LOW*. Since our numbers can be either *HIGH* or *LOW* or 1 and 0, it makes sense for digital systems to count using a base-two system.

Example: Convert the binary number 10011 to decimal form.

Solution:

Binary weight	2^4	2^3	2^2	2^1	2^0
Weight value	16	8	4	2	1
Binary number	1	0	0	1	1

So that the equivalent decimal number is 19.

Octal and Hexadecimal Numbers

In octal numbering system, the base is 8 so our code symbols are limited to 8 (0-7) as for the hexadecimal system, we need 16 symbols to properly present numbers to the base 16. The adopted convention is to use the familiar ten decimal symbols plus A, B, C, D, E, and F to represent the decimal numbers 10, 11, 12, 13, 14, and 15, respectively.

Table: Binary/Octal and Hex conversion

Binary	*Octal*	*Binary*	*Hexadecimal*
000	0	0000	0
001	1	0001	1
010	2	0010	2
011	3	0011	3
100	4	0100	4
101	5	0101	5
110	6	0110	6
111	7	0111	7
		1000	8
		1001	9
		1010	A
		1011	B
		1100	C
		1101	D
		1110	E
		1111	F

From the above table, it can be seen that the conversion between binary and octal or hexadecimal formats is quite straight forward as the following example shows.

Example: Convert the binary number 110001101 to its octal and hexadecimal formats.

Solution:

First, divide the given binary number into groups of 3 bits $\underline{110}\ \underline{001}\ \underline{101}$ and the equivalent octal number would be $(615)_8$.

Similarly, to get the equivalent hexadecimal number, divide the number into groups of 4 bits $\underline{0001}\ \underline{1000}\ \underline{1101}$ so that equivalent hexadecimal number would be $(18D)_{16}$.

Since hexadecimal presentation of a decimal number is more compact than its binary presentation while it is a simple matter to expand it into its binary format, hexadecimal presentation is widely used in computer programming. Hex numbers also bear more resemblance with machine register (two hex digits form a byte). An int$_e$l convention often used in Hex representation is to end a Hex number with h.

This is not a general convention. Motorola, for example, prefaces a Hex number with a $ sign while in the MS Macro Assembler Hex numbers are the default. The Debug command *H* adds and subtracts two hex numbers, for example *H* 23FD 1000 produces 33FD 13FD.

Binary or Hex fractions can also be represented by placing binary/Hex digits to the right of a binary/Hex point. The weights in this case are

$$\cdots d^2\, d^1\, d^0 \bullet d^{-1}\, d^{-2}\ \ d^{-3}\ \cdots$$

Example: Convert the binary number 0•10011 to decimal form.
Solution:

Binary weight	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
Weight value	0.5	0.25	0.125	0.0625	0.03125
Binary number	1	0	0	1	1

So that the equivalent decimal fraction is 0.59375 which can be obtained by simply dividing 19 by 2^5.

Decimal numbers can also be converted to binary numbers as shown in these two examples:

Example: Convert the decimal number 25 to binary form.

Solution:

25/2	12	Remainder = 1	⇐ Least significant digit
12/2	6	(	
6/2	3	(	
3/2	1	1	
1/2	0	1	⇐ Most significant digit

And the answer is $(11001)_2$.

Example: Convert the decimal fraction 0.78125 to binary form.

Solution:

$0.78125\times2 = 1.5625$	Integer = 1	$\Leftarrow$ Most significant digit
$0.5625\times2 = 1.125$	1	
$0.125\times2 = 0.25$	0	
$0.25\times2 = 0.5$	0	
$0.5\times2 = 1.0$	1	$\Leftarrow$ Least significant digit

And the answer is $(0.11001)_2$.

Signed and Unsigned Numbers

To identify positive and negative numbers, a sign bit is often added to the binary number as the most significant bit. The convention is to use 0 for positive numbers and 1 for negative numbers. For example, ±19 will be written as S10011 where S is zero for +19 and one for –19. ***However, using this notation, we need two different hardware types for addition and subtraction***.

With a fixed number of bits we can only represent a certain number of objects. For example, with eight bits we can only represent 256 different objects. Negative values are objects in their own right, just like positive numbers. Therefore, we have to use some of the 256 different values to represent negative numbers. In other words, we got to use up some of the positive numbers to represent negative numbers. To make things fair, we assign half of the possible combinations to the negative values and half to the positive values. So we can represent the negative values –128 ··· –1 and the positive values 0 ··· 127 with a single eight-bit byte. With a 16-bit word we can represent values in the range –32,768 ··· +32,767. In general, with n bits we can represent the signed values in the range -2^{n-1} to $+2^{n-1} - 1$.

There are many ways to represent negative values, but most microprocessors use the two’s complement notation. In the two’s complement representation, the number with negative sign is first one’s complemented and then a one is added. In this way, the high order bit of a number is a sign bit. If it is zero, the number is positive; otherwise, the number is negative. For examples, 8000h is negative and 7FFFh is positive.

Example: Convert the binary number 1001 1110 1000 into two’s complement.

Solution: First, perform the one’s complement 0110 0001 0111 and add 1 to get 0110 0001 1000.

An alternative way to obtain the two's complement of a binary number is to leave unultered the binary bits which are 0 from the right up to and including the first 1 encountered and then complement every other bit. To demonstrate the method, use the binary number 1001 1110 1000:

$$\underbrace{0110\ 0001}_{\text{1's complement}} \quad \underbrace{1000}_{\text{unaltered bits}}$$

Which is the same as before.

The above method is used to go back and forth between positive numbers and their negative counterparts (two's complement). Note that using the above methods, the two's complement of 8000h is also 8000h which means that – (–32,768) is –32,768. Of course, this is not true but this means that the value +32,768 cannot be represented with a 16-bit signed number. Therefore, we cannot negate the smallest negative value.

With the two's complement representation, most operations are as easy as the binary system. For example, suppose you were to perform the addition 13+(–13). Consider what happens when we add these two values in the two's complement system:

```
00001101

11110011
```

The result is zero as expected except that a carry into the ninth bit. As it turns out, if we ignore the carry out, adding two signed values always produces the correct result when using the two's complement numbering system. This means we can use the same hardware for signed and unsigned addition and subtraction. As mentioned before, this would not be the case with some other numbering systems.

In many occasions, you need to convert an eight bit two's complement value to 16 bits. This problem, and its converse (converting a 16 bit value to eight bits) can be accomplished via sign extension and contraction operations. Consider the value "–64". The eight bit two's complement value for this number is C0h. The 16-bit equivalent of this number is FFC0h. Now consider the value "+64". Hence, to sign extend a value from some number of bits to a greater number of bits, just copy the sign bit into all the additional bits in the new format.

Bit Fields and Packed Data

Although the 80x86 operates most efficiently on byte, word, and double word data types, occasionally you may need to work with a data type that uses some number of bits other than eight, 16, or 32. For example, consider a date of the form "14/9/2011". It takes three numeric values to represent this date: a day, month, and year value. Days range between 1 ⋯ 30. So it takes five bits (maximum of 32 different values) to represent the day entry. Months, on the other hand, take on the values between 1 ⋯ 12. So, it requires four bits (maximum of sixteen different values) to represent the month. The year value requires eleven bits (which can be used to represent up to 2048 different values). Five plus four plus eleven is 20 bits, or more than two memory bytes.

In other words, we can pack our date data into three bytes rather than the four bytes that would be required if we used a separate byte for each of the month, day, and two bytes for year values. This saves one byte of memory for each date stored, which could be a substantial saving if you need to store a lot of dates. The bits could be arranged as shown in figure below.

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D	D	D	D	D	M	M	M	M	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Although packed values are efficient in terms of memory usage, they are computationally inefficient. The reason is that it takes extra instructions to unpack the data packed into the various bit fields. These extra instructions take additional time to execute (and additional bytes to hold the instructions); hence, you must carefully consider whether packed data fields will save you anything. Examples of practical packed data types abound. For example, you could pack two BCD digits into a single byte.

ASCII Code

Alphanumeric codes are used to represent numbers, alphabetic letters and control characters. The most common alphanumeric code is the ASCII (*A*merican *S*tandard *C*ode for *I*nformation *I*nterchange) code. It uses 7 binary digits and a parity bit.

The ASCII character set (excluding the extended characters defined by IBM) is divided into four groups of 32 characters. The first 32 characters, ASCII codes 0 through 1Fh (31), form a special set of non-printing characters called the control characters. They are called control characters because they perform various printer/display control operations rather than displaying symbols. Examples include carriage return, which positions the cursor to the left side of the current line of characters 8, line feed (which moves the cursor down one line on the output device), and back space (which moves the cursor back one position to the left). Unfortunately, different control characters perform different operations on different output devices. There is very little standardization among output devices. To find out exactly how a control character affects a particular device, you will need to consult its manual.

The second group of 32 ASCII character codes comprises various punctuation symbols, special characters, and the numeric digits. The most notable characters in this group include the space character (ASCII code 20h) and the numeric digits (ASCII codes 30h ⋯ 39h). Note that the numeric digits differ from their numeric values only in the high order nibble. By subtracting 30h from the ASCII code for any particular digit you can obtain the numeric equivalent of that digit.

The third group of 32 ASCII characters is reserved for the upper case alphabetic characters. The ASCII codes for the characters “A” ⋯ ”Z” lie in the range 41h ⋯ 5Ah (65 ⋯ 90). Since there are only 26 different alphabetic characters, the remaining six codes hold various special symbols.

The final group of 32 ASCII character codes are reserved for the lower case alphabetic symbols, five additional special symbols, and another control character (delete). Note that the lower case character symbols use the ASCII codes 61h ⋯ 7Ah. If you convert the codes for the upper and lower case characters to binary, you will notice that the upper case symbols differ from their lower case equivalents in exactly one bit position.

ASCII	Hex	ASCII	Hex	ASCII	Hex	ASCII	Hex	ASCII	Hex	ASCII	Hex
NUL	00	SOH	01	STX	02	ETX	03	EOT	04	ENQ	05
ACK	06	BEL	07	BS	08	HT	09	LF	0A	VT	0B
FF	0C	CR	0D	SO	0E	SI	0F	DLE	10	DC1	11
DC2	12	DC3	13	DC4	14	NAK	15	SYN	16	ETB	17
CAN	18	EM	19	SUB	1A	ESC	1B	FS	1C	GS	1D
RS	1E	US	1F	SP	20	!	21	”	22	#	23
$	24	%	25	&	26	‘	27	(	28	)	29
*	2A	+	2B	,	2C	-	2D	•	2E	/	2F
0	30	1	31	2	32	3	33	4	34	5	35
6	36	7	37	8	38	9	39	:	3A	;	3B
<	3C	=	3D	>	3E	?	3F	@	40	A	41
B	42	C	43	D	44	E	45	F	46	G	47
H	48	I	49	J	4A	K	4B	L	4C	M	4D
N	4E	O	4F	P	50	Q	51	R	52	S	53
T	54	U	55	V	56	W	57	X	58	Y	59
Z	5A	[	5B	\	5C	]	5D	↑	5E	─	5F
‘	60	a	61	b	62	c	63	d	64	e	65
f	66	g	67	h	68	i	69	j	6A	k	6B
l	6C	m	6D	n	6E	o	6F	p	70	q	71
r	72	s	73	t	74	u	75	v	76	w	77
x	78	y	79	z	7A	{	7B	|	7C	}	7D
~	7E	DEL	7F								

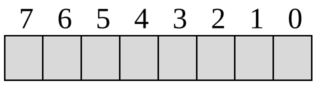
Control Characters:

ACK	Acknowledge	BEL	Ring bell	BS	Backspace
CAN	Cancel	CR	Carriage Return	DC_{1-4}	Direct control
DEL	Delete idle	DLE	Data link escape	EM	End of medium
ENQ	Enquiry	EOT	End of transmission	ESC	Escape
EOB	End of transmission Block	ETX	End text	FF	Form feed
FS	Form separator	GS	Group separator	HT	Horizontal tab
LF	Line feed	NAK	Negative Acknowledge	NUL	Null
RS	Record separator	SI	Shift in	SO	Shift out
SOH	Start of heading	SP	Space	STX	Start text
SUB	Substitute	SYN	Synchronous idle	US	Unit Separator
VT	Vertical tab				

Table: Hexadecimal representation of the ASCII Code

Bits and Bytes

A single binary digit is called a bit. Four bits grouped together are called a nibble. The nibble is not a particularly interesting data structure except for BCD and hexadecimal numbers. A collection of eight bits forms a byte. It is the smallest addressable data item on a μprocessor. Main memory and I/O addresses on the μprocessor are all byte addresses. To access anything smaller requires that you read the byte containing the data and mask out the unwanted bits. The bits in a byte are normally numbered from zero to seven using the convention shown below:



Two bytes may be grouped together to form a word. With a word, you can represent 2^{16} (or 64K) different values. The three major uses for words are integer values, offsets, and segment values. A double word is a pair of words (32 bits). A double word may be used to represent segmented addresses, a 32-bit integer value (which allows unsigned numbers in the range 0 – 4,294,967,295 or signed numbers in the range –2,147,483,648 to +2,147,483,647), or a 32-bit floating point number.

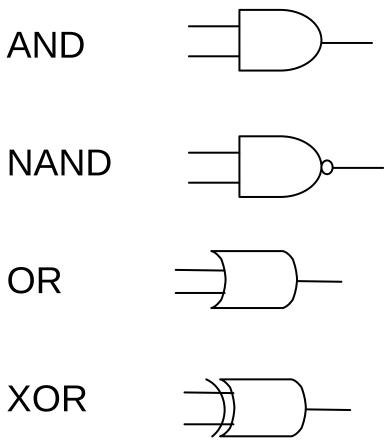
Boolean Logic and Digital Gates

Logical expressions are either simple or compound expressions. Simple logical expressions are:

(a) Logical variables (True or False)
(b) Relational expression of the form

Table: Basic logical operators

A	B	!A	A and B	A or B	A xor B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0



Computer Architecture

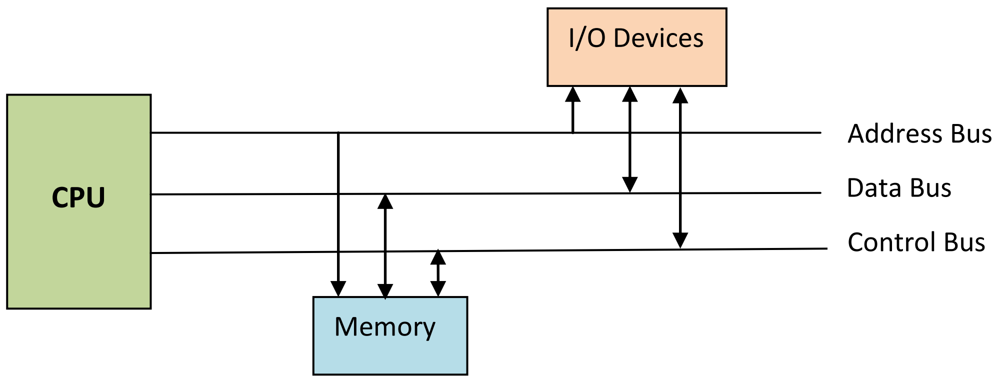


Figure: Von Neumann architecture

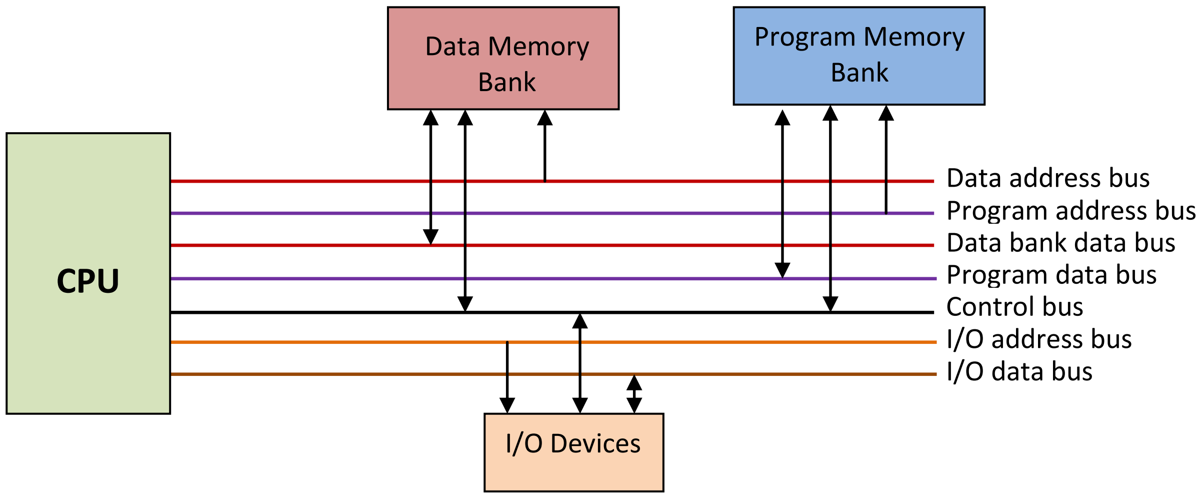


Figure: Harvard architecture

Harvard and Von Neumann Architectures

Harvard	Von Neumann
➢ μprocessor is connected to two different memory banks via two sets of buses to provide the processor with two distinct data paths, one for instruction and one for data ➢ CPU can read instructions and data from the respective memory banks at the same clock cycle which increases the throughput of the machine ➢ Such system is complex in hardware and commonly used in DSPs	➢ μprocessor is connected to a single sequential memory and uses a single data path (bus) for both instructions and data ➢ CPU can either fetch an instruction or read/write data from memory during a certain clock cycle ➢ Processor speeds are much faster than memory access times, and an extremely fast cache memory is used ➢ Modern processors use a Harvard Architecture to read from two instruction and data caches, and use a Von-Neumann Architecture to access external memory

Chapter 2

Microcontrollers and Embedded Systems

What is an imbedded system?

An embedded system is a device or devices used to control, monitor or assist the operation of equipment, machinery or plant. The word “Embedded” reflects the fact that they are an integral part of the system. In many cases, their “embeddedness” may be such that their presence is far from obvious to the casual observer.

Institute of Electrical Engineers (IEE)

Mechatronics is the synergistic combination of precision mechanical engineering, electronic control and systems thinking in the design of products and processes

μprocessors vs. μcontrollers

It is a single chip with ALU, memory and I/O ports on board.

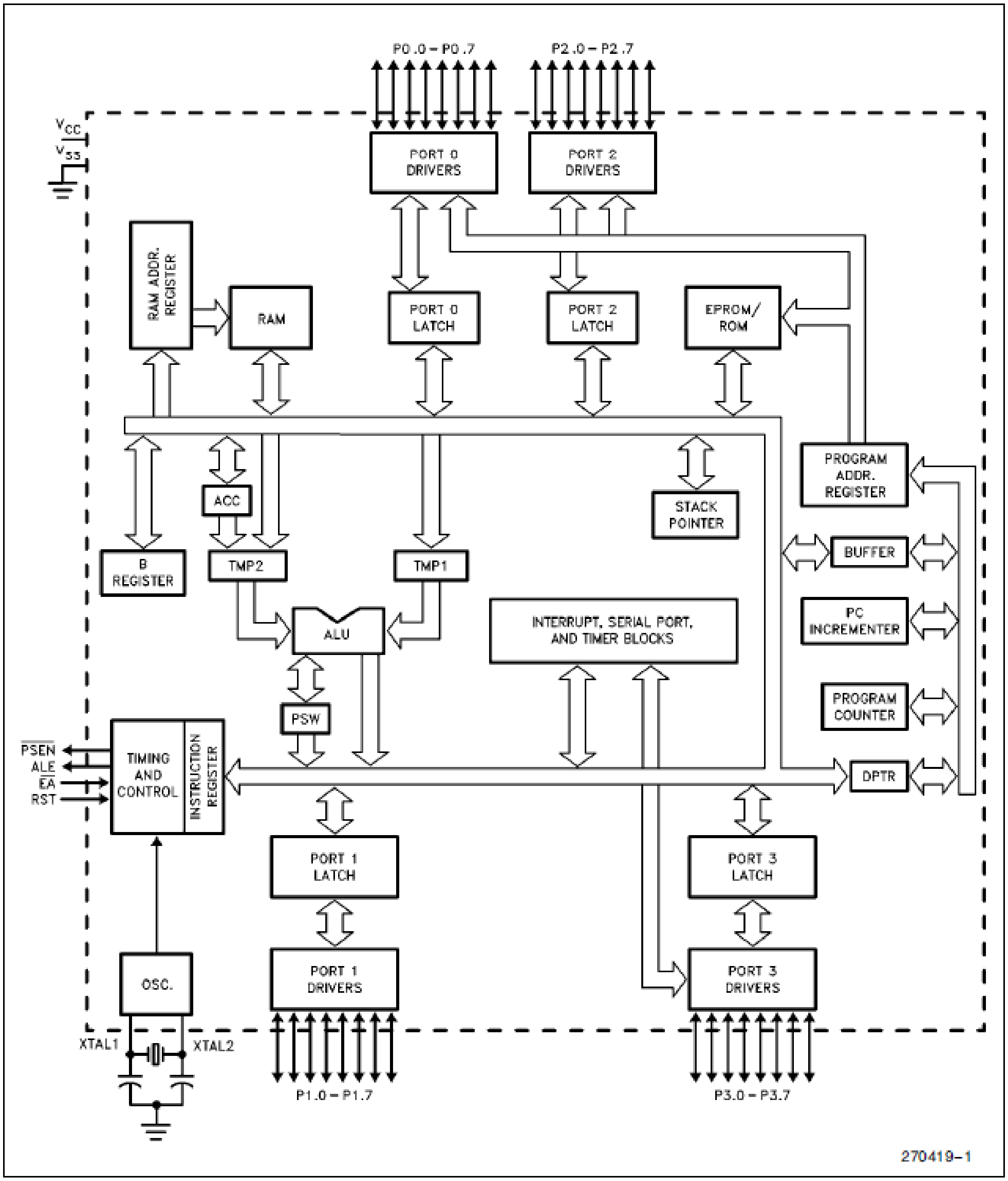
Advantages:

- Low cost (order of a few dollars)
- Low power consumption
- Low speed, on the order of 10 KHz – 33 MHz
- Small architecture (8-bit ALU, no cache, no floating-point processor, etc.)
- Small memory size, but usually enough for a typical application
- Limited I/O, enough for intended applications
- Small size

The 8051 μcontrollers Family

	8051	**8052**	**8031**
ROM (on-chip program space)	4k	8k	0k
RAM	128	256	128
Timers	2	3	2
I/O pins	32	32	32
Serial port	1	1	1
Interrupt sources	6	8	6

The block diagram for the Intel 80C51BH family of high speed 8-bit µcontrollers



Tutorial (I)

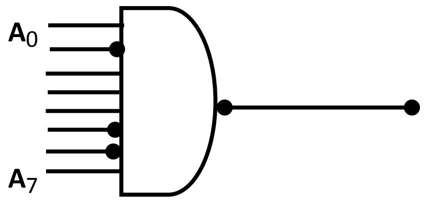
1. Convert the following binary numbers to decimal:
   (a) 0110, (b) 1011, (c) 1111 0000, (d) 1010 1010

2. Convert these binary numbers to octal and hexadecimal:
(a) 1110 (b) 11011 (c) 110110101 (d) 1010111101110010

3. Convert the following decimal numbers to binary, and then to octal and hexadecimal:
(a) 12 (b) 15 (c) 27 (d) 96

4. Perform the following binary additions:
(a) 1 + 1, (b) 1010 + 1111, (c) 11 0111 + 1 1000

5. Using each finger as a binary digit, what is the highest binary number you could count using one hand? How about two hands? Why is this more efficient than the usual way of counting on your fingers?
6. A program variable is to be used to store a unique number identifying any day in the year. How many bits will be required to store it?
7. Multiply 0111 by 0011 in binary.
8. Add –35 to 85 using the two's complement
9. What is the address decoded by the NAND shown:



Answers:

1. (a) 6, (b) 11, (c) 240, (d) 170
2. (a) 1110 = 16 (Octal), E (Hex).
(b) 1 1011 = 33 (Octal), 1B (Hex)
(c) 1 1011 0101 = 665 (Octal), 1B5 (Hex)
(d) 1010111101110010 = 127562 (Octal), AF72 (Hex)
3. (a) 12 = 1100 = 14 (Octal) = C (Hex)
(b) 15 = 1111 = 17 (Octal) = F (Hex)
(c) 27 = 11011 = 33 (Octal) 1B (Hex)
(d) 96 = 1100000 = 140 (Octal) = 60 (Hex)
4. (a) 1 + 1 = 10
(b) 1010 + 1111 = 1 1001
(c) 11 0111 + 1 1000 = 100 1111
5. One hand: 31 and Two hands: 1023.
6. If the days are numbered 0 to 365, then 9 bits will be insufficient.
8. 0101 0101+1101 1101 = 0011 0010
9. 9DH =10011101

Chapter 3

The 8051 Assembly Language Programming

Programming model of the 8051

We mean by the programming model the internal registers and hardware relevant to the microcontroller programmer. Registers hold data to be processed or address of data to be fetched.

Instructions format for the 8051 can be demonstrated by the data movement instruction **`mov`** which copies data from a source into a destination of matching size as follows:

```
   mov dest,source
```

For example,

```
   mov A,R0  ;copies the contents of the R0 register into the accumulator A register .
   mov A,#23H ;loads the accumulator A register with value 23H.
```

Almost all the 8051 registers are single byte wide. The following are the most widely used registers in assembly programming:

A Register (Accumulator)

The A register is the most commonly used register in the 8051 μcontroller. It is a general-purpose register used for storing intermediate results obtained during an ALU operation. Prior to executing an instruction upon any number or operand it is necessary to store it in the accumulator first. All results obtained from arithmetical operations performed by the ALU are stored in the accumulator. Data to be moved from one register to another must go through the accumulator.

B Register

Multiplication and division can be performed only upon numbers stored in the A and B registers. All other instructions in the program can use this register as a spare accumulator.

PC Program counter register

The PC register points to the next executable byte to be executed. Each code byte fetched increments the PC register to point to the next executable instruction.

The PC in the 8051 is two-bytes wide which means that the 8051 can access addresses from 0000h to FFFFh or 64k bytes of code. When the 8051 is powered on, the PC points at byte in location 0000h.

Program status word (PSW) register

PSW register is one of the most important SFRs. It contains several status bits that reflect the current state of the CPU.

0	0	0	0	0	0	0	0	Reset value
CY	AC	F0	RS1	RS0	OV		P	Bit name
PSW_7	PSW_6	PSW_5	PSW_4	PSW_3	PSW_2	PSW_1	PSW_0	

P- Parity bit

Set if a number stored in the accumulator is odd. It is mainly used during data transmit and receive via serial communication.

PSW1

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

OV - Overflow

Set to one if the result of an arithmetical operation is larger than 255 and cannot be stored in a single byte register.

```
mov A,#60H        0110 0000
add A,#46H        0100 0110
                 -----------
                  1010 0110
```

It is clear that the single byte accumulator could not hold the correct answer since the addition of two positive numbers produced a negative result (A6h = –5AH). Hence, the overflow flag will be set OV=1.

RS0, RS1 Register bank select bits

These two bits are used to select one of four register banks of RAM; registers R0-R7 stored in one of four banks of RAM.

RS1 RS0	Register bank	Space in RAM
00	0	00H-07H
01	1	08H-0FH
10	2	10H-17H
11	3	18H-1FH

F0 - Flag 0

This is a general-purpose bit available for user

AC - Auxiliary Carry Flag
It is used for BCD operations only.

```
mov A,#38H          0011 1000
add A,#2FH          0010 1111
                    ---------
                    0110 0111
```

AC is set since there is a carry resulted from first nibble to second nibble.

CY - Carry Flag
Carry flag bit used for all arithmetical operations and shift instructions.

```
mov A,#9CH          1001 1100
add A,#74H          0111 0100
                    ---------
                   10001 0000
```

CY=1 and AC =1

P0, P1, P2, P3 - I/O Registers
The 8051 has 4 ports with a total of 32 I/O pins available for connection to peripherals. Each bit within these ports affects the state and performance of appropriate pin of the microcontroller. Thus, bit logic state is reflected on appropriate pin as a voltage (0 or 5 V) and vice versa, voltage on a pin reflects the state of appropriate port bit.

1	1	1	1	1	1	1	1	Reset value
P0.7	P0.6	P0.5	P0.4	P0.3	P0.2	P0.1	P0.0	Bit name

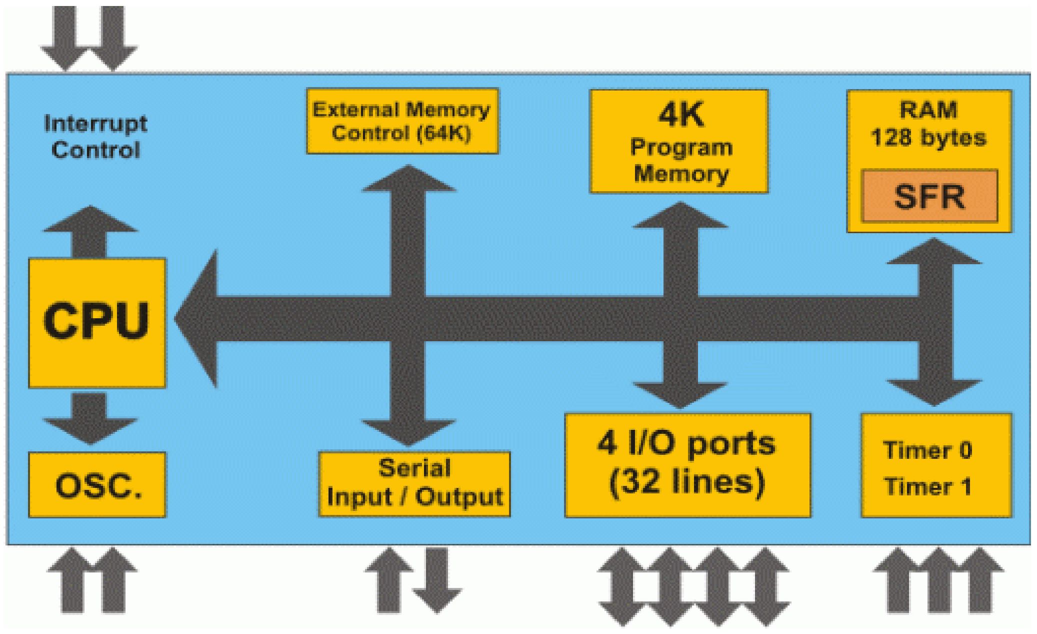
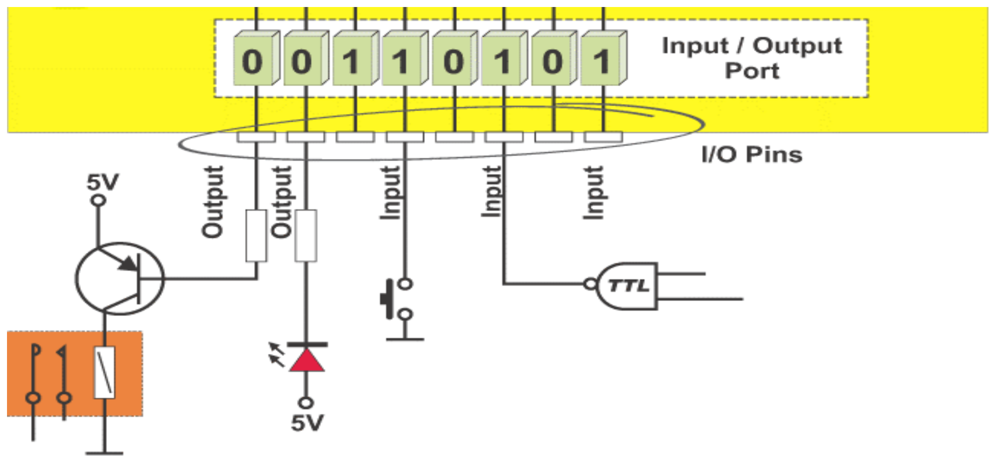


Figure: The block diagram for the Intel µcontroller

If a bit is cleared (i.e. set to 0), the appropriate pin will be configured as an output, while if it is set (1), the appropriate pin will be configured as an input. Reset and power-on set all port bits which means that all appropriate pins will be configured as inputs.



SP Stack pointer register

A value stored in the SP points to the first free stack address and permits stack availability. Stack pushes increment the value in the Stack Pointer by 1. Likewise, stack pops decrement its value by 1.

Upon any reset and power-on, the value 7 is stored in the Stack Pointer, which means that the space of RAM reserved for the stack starts at this location. If another value is written to this register, the entire Stack is moved to the new memory location.

The 8051 Register Banks

The 8051 has 128 bytes of RAM that are allocated as registers and stack. They are divided into the following three groups:

(a) First 32 bytes are reserved for register banks and stack. The PSW bits RS0 and RS1 are used as register bank select bits.

(b) The next 16 bytes are bit-addressable read/write memory.

(c) The remaining 80 bytes are used as scratch pad RAM to store data and parameters.

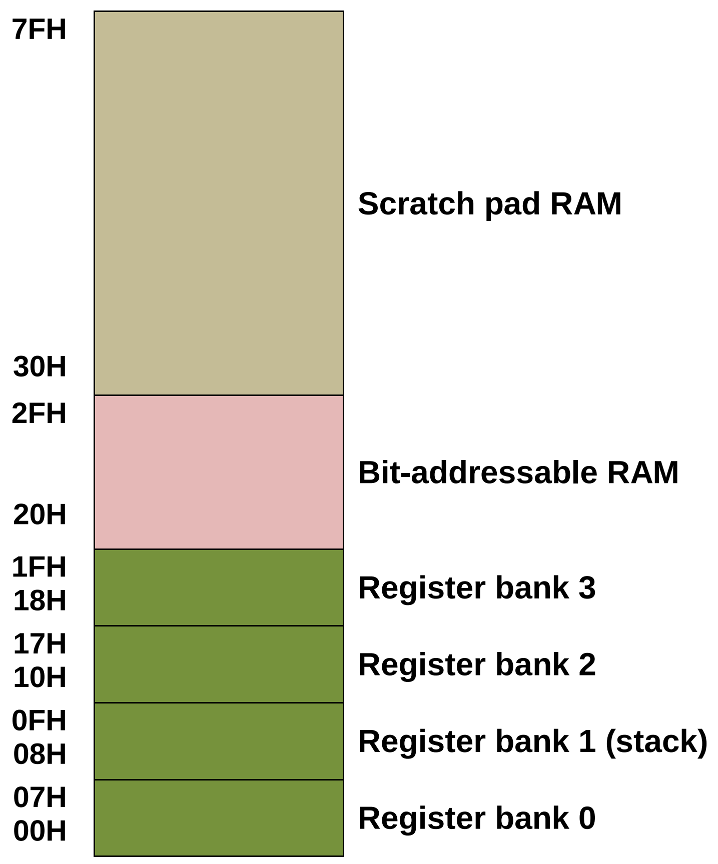


Figure: RAM allocation in the 8051.

As shown in Figure, the first 32 bytes are allocated for 4 register banks each has 8 registers denoted R0-R7. For example, the instruction

```
mov R1,#9CH
```

is very much the same as

```
mov 01,#9CH
```

Register bank 0 is the default unless the PSW bits RS1 RS0 are set to select another register bank.

RS1 RS0	Register bank	Space in RAM
00	0	00H-07H
01	1	08H-0FH
10	2	10H-17H
11	3	18H-1FH

This can be done using the set bit instruction as shown in the following example:

```
setb PSW.4      ;choose register bank 2
mov R0,#9CH     ;set location 10H in RAM to be 9CH
mov R5,#26H     ;set location 15H in RAM to be 26H
```

Stack Operations

The stack is a very important mechanism in microcontroller programming. In the 8051, the stack starts at register bank 1 of the RAM. Hence, the SP register is set initially at 7 so that the first byte to push data into is 8. Each time a byte is pushed into the stack, the SP is incremented by 1 till 1F is reached. Also, popping a byte decrements SP by one till SP points at 7 again.

It must be remembered that push and pop instructions support only direct addressing, i.e. direct RAM address as the following example shows:

```
mov R0,#9CH     ;load R0 9CH
mov R5,#26H     ;load R5 26H
push 0          ;set location 8H in RAM to be 9CH, SP=8
push 5          ;set location 9H in RAM to be 26H, SP=9
pop 3           ;pop stack into R3, i.e. R3 is set to 26H, SP=8
pop 2           ;pop stack into R2, i.e. R2 is set to 9CH, SP=7
```

HW#1: Problems 6, 8, 10, 37, 45, 48 (pp. 64-66).

Introduction to assembler

Assembly language codes are fast and compact which makes them easy to write and debug. Hence, if you have a limited storage and need to develop an application fast, the assembler language is certainly the right choice.

The following diagram illustrates the steps for producing an executable file:

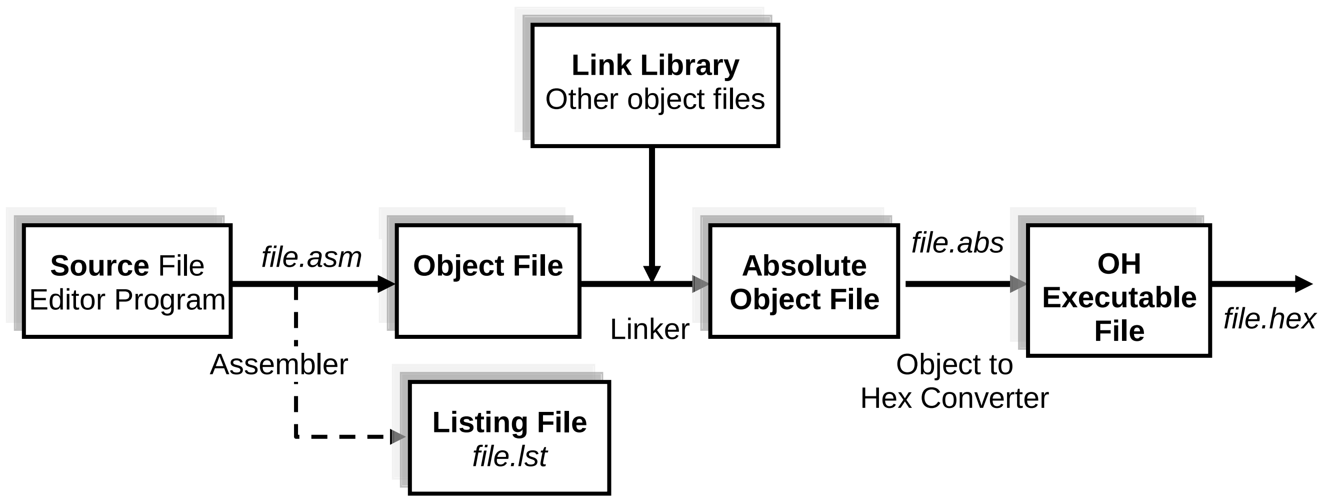


Figure: Steps to assemble a file into an executable program

Assembler Directives

ORG
Used to indicate the beginning address of your code.

DATA
Used to define a name for memory locations.

```
SP      DATA   0x81  ;special function registers
MY_VAL  DATA   0x44   ;RAM location
```

EQU
Used to create symbols that can be used to represent registers, numbers, and addresses

```
LIMIT   EQU   2000
SERIAL  EQU   SBUF
COUNT   EQU   R5
MY_VAL  EQU   0x44
```

DB

```
    CHAR        DB 'A'                  ; ASCII character
    SIGNED      DB +127                 ; Largest signed value
    SIGNED2     DB –128                 ; Smallest signed value
    UNSIGNED    DB 255                  ; Largest unsigned value
    BINARY      DB 01011100b            ; Binary value
    HEX         DB 42h                  ; Hexadecimal value
    STRING      DB "Salam"              ; String value
```

END

Used to indicate the end of the assembly file

List file of a simple 8051 assembly program:

```
   ROM
  Address  M/C              Assembly Code
  -------------------------------------------------------------------
1  0000                     ORG  0H          ;start code at location 0
2  0000    7D26             mov R5,#26H      ;load 26H into R5
3  0002    7F3C             mov R7,#3CH      ;load 3CH into R7
4  0004    7400             mov A,#0         ;load 0H into acc
5  0006    2D               add A,R5         ;acc = acc+R5
6  0007    2F               add A,R7         ;acc = R5+R7
7  0008    2415             add A,#15H       ;acc = acc+15H=R5+R7+15H
8  000A    80FE     here:   sjmp here        ;stay here
9  000C                     END
```

Rules for label:

(1) Alphanumeric in lower and upper case

(2) It can have some or all the special characters _, ?, $, @, . with some reservations in some assemblers

(3) The first character is Alphabetic

(4) Special or reserved words are not allowed

(5) In some assemblers, the label name can start with the underline character _

Tutorial (II)

1. Determine CY and AC flags for the following code:

```
mov A,#0FFH       1111 1111
add A,#01H        0000 0001
                 ----------
                 10000 0000
```

ACC= 00H, CY=1 and AC =1

```
mov A,#0C2H       1100 0010
add A,#3DH        0011 1101
                  ---------
                  1111 1111
```

ACC=FFH, CY=0 and AC =0

2. Determine the register bank used after the following code:

```
setb PSW.3
setb PSW.3
```

RS1 RS0=11 → Register bank 3

3. Determine the contents of memory locations 200H-205H after the following code:

```
        ORG 200H
   Data:  DB  "ABC123"
```

200H-205H has 41H 42H 43H 31H 32H 33H

Chapter 4

The 8051 Addressing Modes

General

The 8051 CPU can access data in various ways. These various ways to access data are called ***addressing modes***. These addressing modes are predetermined by the CPU designer. The 8051 µcontroller has the following addressing modes:

i. Immediate
ii. Register
iii. Memory
iv. Register indirect
v. Indexed

The immediate addressing mode

In immediate addressing, the source operand is a constant. It must be preceded by the pound sign (#). Note the following examples:

```
mov A,#23H          ;load 23H into acc
mov R4,#23          ;load 23 decimal into register R4
mov B,#40H          ;load 40H into B
mov P1,#40H         ;send 40H to port P1
mov DPTR,#4423H     ;DPTR=4423H
```

Where the data pointer (DPTR) register is a two byte register. It can also be accessed as a two independent single byte register DPH and DPL as shown below.

DPTR	DPH	DPL

Therefore, the instruction is equivalent to the following:

```
mov DPH,#44H
mov DPL,#23H
```

```
        ORG 200H
Mydata  DB  "Salam"
        mov DPTR,#Mydata  ;DPTR=200H
```

The register addressing mode

Register addressing mode involves the use of registers to hold the data to be manipulated.

```
mov A, R0       ; copy the contents of R0 into acc
mov R2, A       ; copy the contents of acc into R2
add A, R5       ; add the contents of R5 to contents of acc
add A, R7       ; add the contents of R7 to contents of acc
mov R6, A       ; save the acc in R6
mov R6, R2      ; not allowed
mov DPTR, A     ; error due to size mismatch
```

Remarks:

(a) Data movement between Rn registers is not allowed.

(b) It must be reminded that in all opcodes source and destination must match in size.

The memory addressing mode

Data in memory (RAM) can be accessed using direct addressing or register indirect addressing modes. The RAM in the 8051 has 128 bytes allocated as follows:

a) Bytes in addresses 00-1FH are allocated to register banks and stack
b) Bytes in addresses 20-2FH are allocated for bit addressable space to save single-bit data
c) Bytes in addresses 30-7FH are available to save data bytes

The 8051 memory map is shown in Figure below.

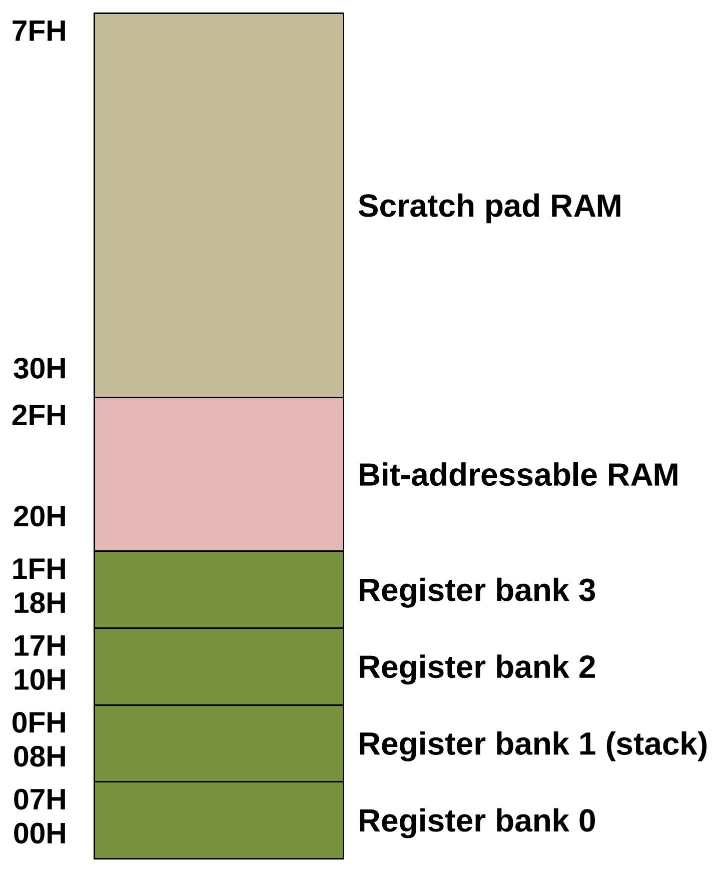


Figure: RAM allocation in the 8051.

Examples:

```
mov  R0,40H        ; copy the contents of RAM location 40H into R0
mov  55H,A         ; copy the contents of acc into RAM location 55H
mov  A,4           ; copy R4 into acc
mov  B,0           ; copy R0 into B register
```

<u>Note the absence of the pound sign #</u>

So far, we know the RAM locations assigned to Rn registers but what about the rest of special function registers (SFR) addresses?

a) Since RAM space from 00H to 7FH is already allocated, SFRs have addresses above 80H.
b) Not all bytes in 80-FFH are used by SFRs and may not be used by 8051 programmer.

Table: SFR addresses

Symbol	Name	Address
Acc*	Accumulator	0E0H
B*	B register	0F0H
PSW*	Program status word	0D0H
SP	Stack pointer	81H
DPTR	Data pointer	
DPL	Data pointer low byte	82H
DPH	Data pointer high byte	83H
P0*	Port 0	80H
P1*	Port 1	90H
P2*	Port 2	0A0H
P3*	Port 3	0B0H
IP*	Interrupt priority control	0B8H
IE*	Interrupt enable control	0A8H
TMOD	Time/counter mode control	89H
TCON*	Time/counter control	88H
T2CON*	Time/counter 2 control	0C8H
T2MOD	Time/counter 2 mode control	0C9H
TH0	Time/counter 0 high byte	8CH
TL0	Time/counter 0 low byte	8AH
TH1	Time/counter 1 high byte	8DH
TL1	Time/counter 1 low byte	8BH
TH2	Time/counter 2 high byte	0CDH
TL2	Time/counter 2 low byte	0CCH
RCAP2H	T/C 2 capture register high byte	0CBH
RCAP2L	T/C 2 capture register low byte	0CAH
SCON*	Serial control	98H
SBUF	Serial data buffer	99H
PCON	Power control	87H

* Bit-addressable

Example:

Write assembly code to send 55H to port P1 and P2 using their names and addresses.

```
mov  A,55H          ; acc= 55H
mov  P1,A           ; copy the contents of acc into port 1
mov  P2,A           ;copy acc into P2
```

or equivalently

```
mov  A,55H          ; acc= 55H
mov  90H,A          ; copy the contents of acc into port 1
mov  0A0H,A         ;copy acc into P2
```

Register indirect addressing mode

In this addressing mode, registers R0 and R1 are used as pointers to data. This means that the data address is held in one of these registers. They must be preceded by the address pointer sign @.

```
mov A,@R0  ; move contents of RAM whose address is held in R0 into acc
mov @R1,B  ; move contents of B into RAM location pointed at by contents of R1
```

Example:

Write an assembly code to copy the value 55H into RAM locations 40H-43H.

```
mov A,#55H    ; A=55H
mov R0,#40H   ; R0=40H to point to RAM location 40H
mov @R0,A     ; RAM location 40H=55H
inc R0        ; increment R0 to point to next byte
mov @R0,A     ; RAM location 41H=55H
inc R0        ; increment R0 to point to next byte
mov @R0,A     ; RAM location 42H=55H
inc R0        ; increment R0 to point to next byte
mov @R0,A     ; RAM location 43H=55H
```

A more elegant to do the above code is to use a loop as follows:

```
        mov A,#55H        ; A=55H
        mov R0,#40H       ; R0=40H to point to RAM location 40H
        mov R2,#04H       ; set counter R2=4H
again:  mov @R0,A         ; copy acc to RAM location
        inc R0            ; increment R0 to point to next byte
        djnz R2, again    ; decrement R2 and jump in not zero to again
```

The listing file for the above code is shown below:

```
8051 Assembler        Version 1.00   03/18/112 01:06:51    Page 1
C:\LaTeX\COURSES\Microcontrollers\8051Assembler\p1.a51

    1 0000                org 0h
    2 0000 7455           mov  A,#55H      ; A=55H
    3 0002 7840           mov  R0,#40H     ; R0=40H to point to RAM location 40H
    4 0004 7A04           mov  R2,#04H     ; set counter R2=4H
    5 0006 F6      again: mov  @R0,A       ; copy acc to RAM location
    6 0007 08             inc  R0          ; increment R0 to point to next byte
    7 0008 DAFC           djnz  R2, again  ; decrement R2 and jump in not zero to again
    8 000A                end
```

```
Defined Symbols:
Defined Labels:
  again                    000006  6
```

Now, since R0 and R1 are one byte wide, data outside FFH space cannot be accessed. In this case, the DPTR register may be used to access data stored in externally connected RAM or on-chip ROM.

Example: (Look-up tables)
Write an assembly code to get the value of an integer between 0 and 9 from port P1 and its square to P2. Assume the look-up table address starts at 300H.

```
        ORG 0
        mov  DPTR,#300H      ; look-up table pointer
        mov  A,#0FFH         ; A=FF
        mov  P1,A            ; configure P1 as input port

again:  mov  A,P1            ; get x from port P1
        movc A,@A+DPTR       ; move code byte pointed at by acc+DPTR (look-up table)
        mov  P2, A           ; send square to P2
        sjmp again           ; short jump again

        ORG 300H
xsqr_table:
        DB 0,1,4,9,16,25,36,49,64,81
        END
```

Chapter 5

Arithmetic, Logic and Program Control Instructions

Programming Instructions

The 8051 µcontroller program comprises a set of instructions written by the programmer. There are four classes of instructions:

1. Arithmetic operations
2. Logic operations
3. Data transfer operations
4. Branch operations

Arithmetic operations

Arithmetic instructions operate on whole numbers only and support addition, subtraction, multiplication and division.

Addition

`add` A,#66H ; add the hex number 66 to the accumulator A

Remember that this is an example of immediate addressing. The # sign is important, if it were omitted the operation would have a different meaning.

`add` A,66H ; add to accumulator A the contents of RAM address 0066H

This is an example of direct addressing.

`inc` 66H ; increment (add 1) the contents of address 0066H

Is there a difference between: (a) `add` A,#1H (b) `inc` A ?

Subtraction

`subb` A, #66H ; subtract 66H from the contents of A

The extra B in the instruction implies Borrow. If the contents of A are less than the number being subtracted then bit 7 of the program status word (PSW).

`dec` A ; decrement A by 1and put result into A

Multiplication

The 8051 supports mutliplications of unsigned bytes.

`mul` AB ; multiply the contents of A and B, put the answer in AB

A is the accumulator and B is another 8-bit SFR provided for use with the instructions multiply and divide. The 2 byte product of the multiplication process would be stored in the concatenated AB register.

Example

If A = 135 decimal, B = 36 decimal. What would be the value in each register after executing the instruction **`mul`** AB?

Solution

A × B = 4860 = 12FCH so that 2H would be placed in A and FCH in B

Division

`div` AB ; divide A by B, put quotient in A and remainder in B

Example

Let A =135, B = 36. What would be the value in each register after execution of the instruction **`div`** AB?

Solution

Decimal values are assumed if the value quoted is not followed by an H

A/B = 3; remainder 27 = 1BH: Hence 03H in A, 1BH in B

Logic operations

The set of logic functions include:

1. **`anl`** AND Logic
2. **`orl`** OR Logic
3. **`xrl`** exclusive OR Logic
4. **`cpl`** Complement (i.e. switch to the opposite logic level)
5. **`rl`** Rotate Left (i.e. shift byte left)
6. **`rr`** Rotate Right (i.e. shift byte right)
7. **`setb`** Set bit to logic 1
8. **`clr`** Clear bit to logic 0

AND operation

The **`anl`** instruction is useful in forcing a particular bit in a register to logic 0 without altering other bits. The technique is called masking.
Suppose register 1 (R1) contains EDH (1110 1101B), i.e. bit 1 and bit 4 are at logic 0, the rest at logic 1.

`anl` R1, #7FH ; 7FH = 0111 1111B, forces bit 7 to zero

ORL operation

Another aspect of masking is to use the ORL instruction to force a particular bit to logic 1, without altering other bits. For example, the power control (PCON) SFR in the 8051 family, is not bit addressable and yet has a couple of bits that can send the microcontroller into idle mode or power down mode, useful when the power source is a battery. The contents of the PCON SFR are:

PCON

SMOD1	SMOD2		POF	GPF1	GPF2	PD	IDL

- SMOD1 and 2 are used when setting the baud rate of the serial onboard peripheral.
- POF, GPF1, and GPF2 are indictor flag bits.
- IDL is the idle bit; when set to 1 the microcontroller core goes to sleep and becomes inactive. The on-chip RAM and SFRs retain their values.
- PD is the Power Down bit, which also retains the on-chip RAM and SFR values but saves the most power by stopping the oscillator clock.

`orl` PCON,#02H ; enables Power Down
`orl` PCON,#01H ; enables Idle mode

Either mode can be terminated by an external interrupt signal.

CPL complement operation

The instructions described so far have operated on bytes (8 bits) but some instructions operate on bits and CPL is an example.
`cpl` P1.7 ; complement bit 7 on Port 1
Port 1 is one of the microcontroller's ports with 8 pins.

Example

If the contents of port 0 (P0) = 125, what would be the port contents after execution of the following instruction?

`cpl` P0

Solution

P0 = 125 = 01111101 B = 7DH

`cpl` P0 will complement P0 to 82H =130

RL, rotate left one bit, RR, rotate right one bit operations

Suppose the accumulator A contents are 01H, then

`rl` A ; contents of A become 0000 0010B or 02H

`rl` A ; 0000 0100B or 04H

`rl` A ; 0000 1000B or 08H

`rl` three times has the effect of multiplying A by 2^3 i.e. by 8.

On the other hand, suppose the accumulator A contents are 80H, or 128 decimal, then:

`rr` A ; contents of A become 0100 0000B which is 64 decimal

`rr` A ; A becomes 0010 0000B=32 decimal

`rr` A ; A becomes 0001 0000B=16 decimal

`rr` three times has the same effect as dividing A by 2^3 i.e. 8.

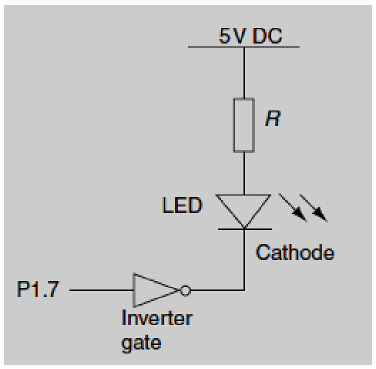
SETB set bit, CLR clear bit operations

This instruction operates on a bit, setting it to logic 1.

`setb` P1.7 ; set bit 7 on Port 1 to logic 1

Example

Consider Figure below where pin 7 of port 1 is connected as shown.



`setb` P1.7 puts logic 1 (e.g. 5 V) onto the inverter input and therefore its output, the LED cathode, is at 0V causing current to flow through the LED.
The LED has a particular forward voltage V_f. Typically $V_f = 2.2V$ and forward current $I_f = 8mA$ so that:

$$R = (5–V_f)/ I_f = (5-2.2)/8mA = 350\Omega$$

`clr` P1.7 ; clears bit 7 on port 1 to zero

`clr` P1.7 puts logic 0 on the inverter gate input and therefore its output, the LED cathode, becomes logic 1 which is 5 V. This gives a voltage difference of 0V and the LED turns off.

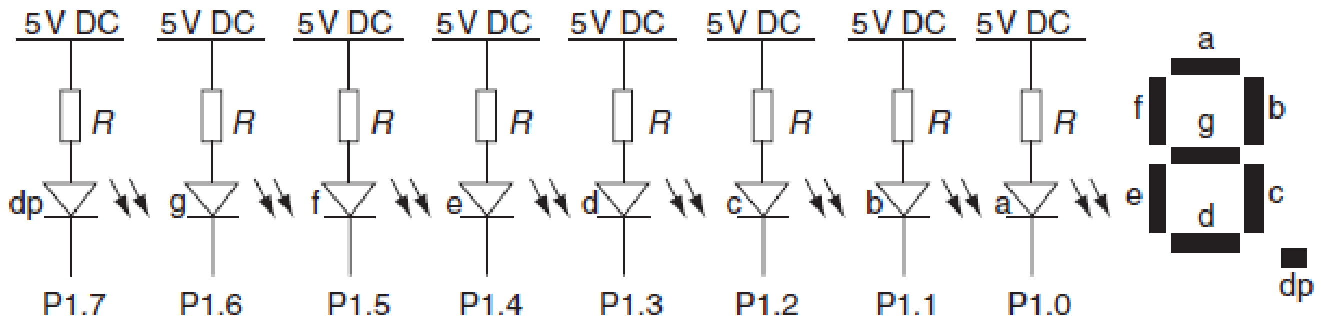
The inverter gate in the above circuit provides a good current buffer protecting the microcontroller port pin from unnecessary current loading. In the above circuit the current flow is between the inverter gate and the 5V DC supply.

Data transfer operations

This is mainly concerned with transfer of data bytes.

MOV operation example

Consider driving a seven-segment display (decimal point dp included) where each LED is driven by the sink method as shown in Figure below:



where each LED illuminates a segment. The seven-segment display is shown to the right with its standard segment identification letters.

Now, let us write two program lines, one to display 3, the second to display 4 turning the decimal point off in both cases.

	P1.7	P1.6	P1.5	P1.4	P1.3	P1.2	P1.1	P1.0	
	dp	g	f	e	d	c	b	a	
3	1	0	1	1	0	0	0	0	B0H
4	1	0	0	1	1	0	0	1	99H

```
mov P1,#0B0H      ; display 3
mov P1,#99H       ; display 4
```

Note: **mov** P1,#B0H would give a syntax error. In common with a number of cross assemblers the software would see B0H as a label because it starts with a hex symbol, therefore. a zero must be placed in front of the hex symbol .

HW#2: Problems 1, 3, 12, 15, 24 (pp. 174-176).

Application Projects

Project 1: Speed control of a small DC motor

The requirement is to use the 8051 microcontroller to drive a DC motor in both forward and reverse directions of shaft rotation and to implement a two-speed (fast and slow) arrangement. Switches are to be used to produce the two speeds and effect a reversal of shaft rotation.

Project 2: Speed control of a stepper motor

The requirement are similar to Project 1 but for driving a stepper motor in both forward and reverse directions of shaft rotation and to implement a two-speed (fast and slow) arrangement. Switches are to be used to produce the two speeds and effect a reversal of shaft rotation.

Project 3: Function generator

The requirement is to design a function generator, using the 8051 microcontroller, with the minimal amount of external components, to generate sine, square and sawtooth waveforms. The output of the circuit is not designed to source an output current to the circuits under test and a buffer circuit is required to enhance the current sourcing capability and also provide a low output resistance for the function generator.

Project 4: InfraRed Remote Switch

The requirement is to switch on/off the Home Appliances by using a standard Remote control. The system is used to switch on/off several electrical devices. All the above processes are controlled by the 8051Microcontroller. The Microcontroller receives the Infrared Signal from the receiver and it decodes and switch on/off the appropriate Device.

Project 5: Traffic Light Controller

The requirement is to use the 8051 microcontroller to design a four way traffic signal with Green, Yellow, and Red LEDs. The traffic signal should be timed to reflect road traffic condition.

Project 6: Two Line Intercom

The requirement is to use the 8051 microcontroller to design an electronic private exchange. It has two telephones, which have the intercom facility, and they can be connected to the telephone line. The DTMF (Dual Tone Multiple Frequency) signals are decoded by a DTMF decoder and the switching functions are done via relays.

Project 7: Temperature controlled Fan

The requirement is to use the 8051 microcontroller to control Fan speed according to the temperature and it also indicates the temperature. The system will get the temperature from sensor and it will control the speed according the values stored by designer in the code.