

TOMSK POLYTECHNIC UNIVERSITY

S.N. Torgaev

**MICROPROCESSOR CONTROL SYSTEMS
PART 1. THE INTEL 8080 MICROPROCESSOR
AND 8051 MICROCONTROLLER**

*It is recommended for publishing as a study aid
by the Editorial Board of Tomsk Polytechnic University*

Tomsk Polytechnic University Publishing House
2013

UDC 681.322
BBC 32.973.26-04
T60

Torgaev S.N.

T60 Microprocessor control systems. Part1. The Intel 8080 microprocessor and 8051 microcontroller / S.N. Torgaev; Tomsk Polytechnic University. – Tomsk: TPU Publishing House, 2013. – 126 p.

The basic principles of operation of Intel 8080 microprocessor, its internal structure and assembly programming are discussed. The differences between microprocessor and microcontrollers are shown and basic blocks of microcontrollers and their initialization are described.

UDC 681.322
BBC 32.973.26-04

Reviewers

associate professor of the Tomsk Polytechnic University
F.A. Gubarev

senior staff scientist of the Institute of Atmosphere Optics SB RAS
D.V. Shiyarov

© STE HPT TPU, 2013

© Torgaev S.N., 2013

© Design. Tomsk Polytechnic University
Publishing House, 2013

INTRODUCTION

Nowadays it is difficult to find an area in our lives where microprocessors and microcontrollers are not used. An understanding of physical and logical principles in the simplest microprocessors and microcontrollers is essential for the development of various control devices based on them.

This manual focuses on the principles of operation of microprocessors and basic blocks (timers, I/O ports, memory and others) of modern microcontrollers, Intel 8080 microprocessor and 8051 microcontroller being used as examples. A detailed description of the microprocessor operation during the program execution and the purpose of building blocks within its structure is given.

PART 1. POSITIONAL NOTATIONS

In positional notation the numerical value of a digit depends on its position in the sequence of numbers.

In general, the sequence of numbers is as follows [1]:

$$x = a_{n-1}a_{n-2}\dots a_1a_0,$$

where $0 \leq a_k \leq b-1$, and b is a base.

At present the positional notations with bases $b = 2, 10, 16$ are the most common.

If $b > 10$ for denoting a number of the corresponding value, the notation conforming to numbers 10, 11, etc. is used. For example, in hexadecimal system such symbols are represented by letters:

$$A \rightarrow 10$$

$$B \rightarrow 11$$

$$C \rightarrow 12$$

$$D \rightarrow 13$$

$$E \rightarrow 14$$

$$F \rightarrow 15.$$

Example 1.1:

535_{10} – decimal system

$0001\ 0111_2 \rightarrow 0001\ 0111b \rightarrow bx0001\ 0111$ – binary system

537_8 – octal system

$8CA_{16} \rightarrow 8CAh \rightarrow 0x8CA$ – hexadecimal system

A positive integer x in positional base- b system is represented as a finite linear combination of powers of b .

$$x = \sum_{k=0}^{n-1} a_k b^k,$$

where $0 \leq a_k \leq b-1$.

Example 1.2:

$$539_{10} = 5 \cdot 10^2 + 3 \cdot 10^1 + 9 \cdot 10^0 = 500 + 30 + 9 = 539$$

$$0001\ 0111_2 = 0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + \\ + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 16 + 4 + 2 + 1 = 23$$

$$537_8 = 5 \cdot 8^2 + 3 \cdot 8^1 + 7 \cdot 8^0 = 320 + 24 + 7 = 351$$

$$8CA_{16} = 8 \cdot 16^2 + 12 \cdot 16^1 + 10 \cdot 16^0 = 2048 + 192 + 10 = 2250$$

Microprocessor technology operates on binary numbers. Regardless of the numbers in the user program, the microprocessor always converts them into a sequence of binary digits 0 and 1.

The following abbreviations are used to indicate the formats of binary numbers [1].

- **Bit** – a binary digit, which has two values (0, 1).
Two bits can represent only four numbers (00, 01, 10, 11).
 n bits can represent 2^n numbers.
- **Tetrad** – a combination of four bits, it describes the 16 combinations of numbers. This combination coincides with the number of digits in the base-16 system. Thus any combination of the tetrad can be represented by 4 bits in a binary system or by one number or letter in the base-16 system:

<i>Base-10</i>	<i>Base-2</i>	<i>Base-16</i>
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

- **Byte** – 8 bits. Byte allows us to describe 256 different numbers. The bits in a byte are numbered from right to left starting from zero. The smallest number in base-16 system is written as 00_{16} , and the largest one is written as FF_{16} .

Example 1.3:

$$0001\ 0111_2 = 17_{16}$$

$$0001\ 1100_2 = 1Ch$$

$$1010\ 1111_2 = 0AFh$$

- **Word** – a combination of 16 bits or 2 bytes. The word contains $2^{16} = 65536$ combinations.

For a brief description of large powers of 2 (which is used to characterize the memory size) special symbols are used. The size of 2^{10} is represented by the letter **K**, which reads "**KB**" (kilobytes). The size of 2^{20} is represented by the letter **M**, which reads "**MB**" (megabyte), the number 2^{30} is represented by **G**, which reads "**GB**" (gigabyte).

1.1. CONVERSION FROM BASE-10 TO OTHER POSITIONAL NOTATIONS

To convert a decimal number x to the number base- b system it is necessary to consistently divide the original number x and derived quotients by b until the quotient is zero. The desired representation of a sequence of modulo from operations of division, where the first digit of the modulo is the least significant bit of the desired number.

Example 1.4:

Converting the number 23 to the base-2 system:

$$23/2 = 11(\text{excess } 1)$$

$$11/2 = 5(\text{excess } 1)$$

$$5/2 = 2(\text{excess } 1)$$

$$2/2 = 1(\text{excess } 0)$$

$$1/2 = 0(\text{excess } 1)$$

$$23_{10} = 0001\ 0111_2$$

Converting the number 2250 to the base-16 system yields:

$$2250/16 = 140(\text{excess}10)$$

$$140/16 = 8(\text{excess}12)$$

$$8/16 = 0(\text{excess}8)$$

$$2250_{10} = 8CA_{16}$$

1.2. SIGNED BINARY NUMBERS

For representing the sign, high-order bit (or MSB) is used. To represent the positive number the significant bit is set equal to zero; for negative numbers, this bit is set to 1 [1].

For writing signed numbers it is necessary to set its format, i.e. the number of bits reserved for writing a signed number. Typically, this format consists of eight bits. In the true form the MSB is used as a sign bit, and the remaining low-order bits (or LSB) are used to indicate the module of number.

Example 1.5:

$$0001\ 0111_2 = 23_{10}$$

$$1001\ 0111_2 = 175_{10} \text{ - unsigned}$$

$$1001\ 0111_2 = -23_{10} \text{ - signed}$$

The maximum positive number in the true form in a base-2 system is written as $01111111 = 127_{10}$.

The minimum negative number in true form in a base-2 system is written as $11111111 = -127_{10}$.

The true form was used at the early stages of development of microprocessors. Now only a complement-on-two is used to write the signed numbers. The positive numbers in complement-on-two encoding coincide with the true form and for encoding negative numbers a special rule is introduced. For obtaining the complement-on-two of negative numbers in the format of n bits it is necessary to write n -bit module of number, then we invert all the bits and add one to this number.

Example 1.6:

$$0001\ 0111_2 = 23_{10}$$

$$\text{Inversion} - 1110\ 1000_2$$

$$\text{Addition of one} - 1110\ 1001_2 = -23_{10}$$

$$1110\ 1001_2 = 233_{10} \text{ - unsigned}$$

$$1110\ 1001_2 = -23_{10} \text{ - signed}$$

$$\text{And } 11111111_2 = -1_{10}.$$

1.3. FIXED-POINT NUMBERS

Writing any fractional part of x in base- b positional system with a fixed point is based on the idea of representing the numbers in the polynomial form [1].

$$x = a_{-1}b^{-1} + a_{-2}b^{-2} + a_{-3}b^{-3} + \dots + a_{-m}b^{-m}.$$

When converting a fractional part of numbers from the base-10 system to the base- b system the following algorithm is typically used:

1. The fractional part is multiplied by b , and the integer part is discarded;
2. The newly obtained fractional part is multiplied by b , etc.;
3. This procedure continues until the fractional part becomes zero;
4. The integer parts are written after the decimal point in the order they have been received (left to right). So if the integer part is 0, then we write "0" and if the integer part is 1, then we write "1".
5. The result can be a terminating fraction or periodical fraction in the base- b system. Therefore, when the fraction is periodic, it is necessary to break off the multiplication at any step and be content with an approximate notation of the original number in the base- b system.

Example 1.7: Write the number 521.4375 in base-2 system.

Converting the integer part of number:

$$521/2=260(\text{excess } 1)$$

$$260/2=130(\text{excess } 0)$$

$$130/2=65(\text{excess } 0)$$

$$65/2=32(\text{excess } 1)$$

$$32/2=16(\text{excess } 0)$$

$$16/2=8(\text{excess } 0)$$

$$8/2=4(\text{excess } 0)$$

$$4/2=2(\text{excess } 0)$$

$$2/2=1(\text{excess } 0)$$

$$1/2=0(\text{excess } 1)$$

$$521_{10}=10\ 0000\ 1001_2$$

Converting the fractional part of number:

$$0.4375 \cdot 2 = 0.875 \text{ write } 0$$

$$0.875 \cdot 2 = 1.75 \text{ write } 1$$

$$0.75 \cdot 2 = 1.5 \text{ write } 1$$

$$0.5 \cdot 2 = 1 \text{ write } 1$$

$$.4375_{10} = .0111_2$$

Thus, a complete notation of the number 521.4375 in base-2 system would be $0010\ 0000\ 1001.0111_2 = 209.7_{16}$. Then we check:

$$0010\ 0000\ 1001.0111_2$$

$$x = 1 \cdot 2^9 + 1 \cdot 2^3 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} = .$$

$$= 512 + 8 + 1 + 0.25 + 0.125 + 0.0625 = 521.4375$$

Example 1.8: Write in base-2 system the number 1.94.

Converting the integer part of number:

$$1/2 = 0(\text{excess } 1)$$

$$1_{10} = 0001_2$$

Converting the fractional part of number:

$$0.94 \times 2 = 1.88 \text{ write } 1$$

$$0.88 \times 2 = 1.76 \text{ write } 1$$

$$0.76 \times 2 = 1.52 \text{ write } 1$$

$$0.52 \times 2 = 1.04 \text{ write } 1$$

$$0.04 \times 2 = 0.08 \text{ write } 0$$

$$0.08 \times 2 = 0.16 \text{ write } 0$$

etc.

$$.94_{10} = .111100_2$$

Thus, a complete notation of the number 1.94 in base-2 system is $0001.1111_2 = 1.F_{16}$. We check:

$$0001.1111_2$$

$$x = 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} = .$$

$$= 1 + 0.5 + 0.25 + 0.125 + 0.0625 = 1.9375$$

1.4. FLOATING POINT NUMBERS

It is often necessary to process very large numbers (e.g. the distance between stars) or very small numbers (for example, the size of atoms or electrons). In such calculations we would have to use a great number of bits. For such calculations fixed-point numbers are ineffective [1].

In decimal arithmetic for writing such numbers the algebraic form is used. This number is written as the *mantissa* multiplied by 10 in the power that shows the number exponent.

Example 1.9:

$$0.2 \cdot 10^{25} \text{ or } 0.16 \cdot 10^{-30}.$$

This form is also used for the notation of binary numbers. This notation is called a floating point number. In this form mantissa cannot be more than one, and zero cannot be written after the decimal point in the mantissa.

There is the standard IEEE 754 for the representation of single-precision (float) and double-precision (double) numbers. To write the floating-point single precision number a 32-bit word is required. To write the double precision number a 64-bit word is required. The number is commonly saved in several neighboring cells of memory. The formats of single precision floating-point and double-precision floating-point numbers are shown in Fig. 1.

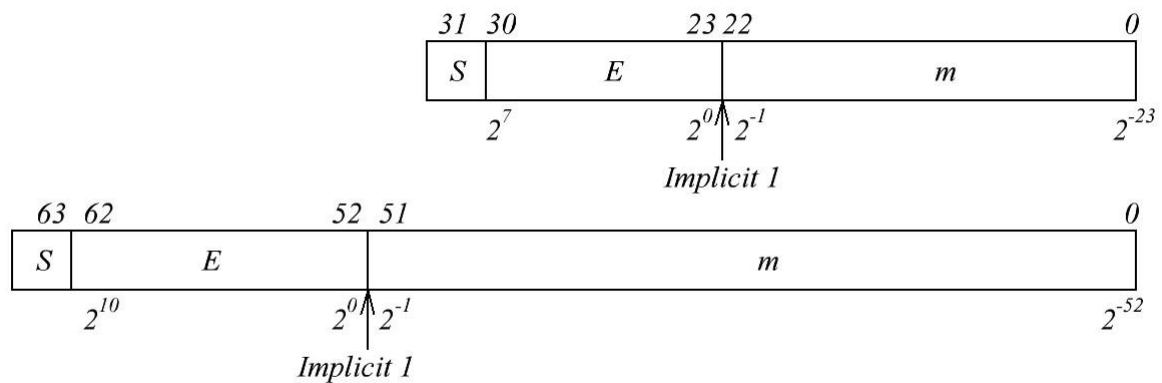


Fig. 1.1. Formats of floating-point numbers

The letter S is a sign of numbers, 0 is a positive number, 1 is a negative number. *E* is a biased exponent of number.

The biased exponent is required not to introduce another sign in the number. The biased exponent is always a positive number. There are eight bits for single-precision and 11 bits for double-precision for the biased exponent. For single-precision, the biased exponent equals 127, and for double precision it is 1023. In decimal mantissa the digits 1-9 can be used and in binary mantissa - only 1. Therefore, there is not a single bit after the binary point in the floating-point number. The unit is meant as a binary comma. In addition, it is assumed in the format of floating point numbers that the mantissa is always more than 1. So, the mantissa is in the range from 1 to 2.

The number in its binary floating-point code can be calculated from:

$$x = (-1)^S \cdot (1.m) \cdot 2^E.$$

Properties of the floating-point numbers:

1. Normalized form of any non-zero number can be represented in a single form. The disadvantage of such a notation is the fact that it is impossible to represent the number 0.
2. Since the MSB of the binary number written in the normalized form is always equal to 1, it can be turned down. This rule is used in the IEEE 754.
3. In contrast to integers, floating-point numbers (double, for example) have a quasi-uniform distribution.
4. Because of Property 3, floating-point numbers have a constant relative error (in contrast to integers, which have a constant absolute error).
5. Not all real numbers can be represented as a floating-point number.
6. In the double format the numbers from $\times 10^{-308}$ to $.7 \times 10^{-307}$ can be represented.

Here is an example of calculations of numbers in a binary code.

Example 1.10: Determine the floating point number consisting of four bytes:

11000001 01001000 00000000 00000000

- Sign bit equal to 1 shows that the number is negative;
- The biased exponent 10000010 in decimal form corresponds to the number 130. Subtracting 127 from 130, we get the number 3.
- Mantissa: 1.100 1000 0000 0000 0000 0000
- So the decimal number is: $1100.1_2 = 12.5_{10}$.

Check:

$$x = (-1)^1 \cdot (1.100\ 1000) \cdot 2^{130-127} = -1.5625 \cdot 2^3 = -12.5.$$

PART 2. MICROPROCESSORS AND MICROPROCESSOR SYSTEMS

2.1. BASIC CONCEPTS

Microprocessor – a software-controlled device that processes digital information and controls it.

Microprocessor-based hardware – a set of compatible chips, such as a microprocessor chip RAM, non-volatile memory, input-output devices, etc.

Microprocessor system – a computer or control system based on microprocessor – based hardware.

Microcomputer with little computational resources and simplified command system, performing logic control of various devices, is called **microcontroller**.

In microprocessor systems, the processing of all information and control of computation process is performed by a microprocessor. It is the microprocessor that performs all arithmetic and logic operations (addition, multiplication, shift, comparison, etc.), the temporary storage of program code and data, the data transfer between peripheral devices, etc. All the additional functions such as the storage of programs and data, communication between devices of microprocessor system, communication with a personal computer and others are performed by peripheral devices.

The microprocessor performs all operations sequentially. At present there are a large number of processors capable of performing multiple operations in parallel. The disadvantage of serial operations is that the run-time of the program, strongly depending on the complexity and speed of the microprocessor system, will be significant.

The control program describes all operations that will be performed by the microprocessor. The program consists of the commands, the digital codes (operational codes), that ‘report’ to the microprocessor which operation should be performed. A set of commands that a microprocessor can perform forms its command system.

In 1945 John von Neumann offered the basic program control principles:

1. The computer must operate completely electronically;
2. Application of the binary system, where bits – or binary digits – can only have the value “0” or “1”. He also proposed the bit as a measuring unit for computer memory;
3. The use of an arithmetical unit;
4. The use of a central processing unit;
5. Storage and control of a program and data storage.

2.2. CLASSIFICATION OF MICROPROCESSORS

In Fig. 2.1 the classification of microprocessors is presented:

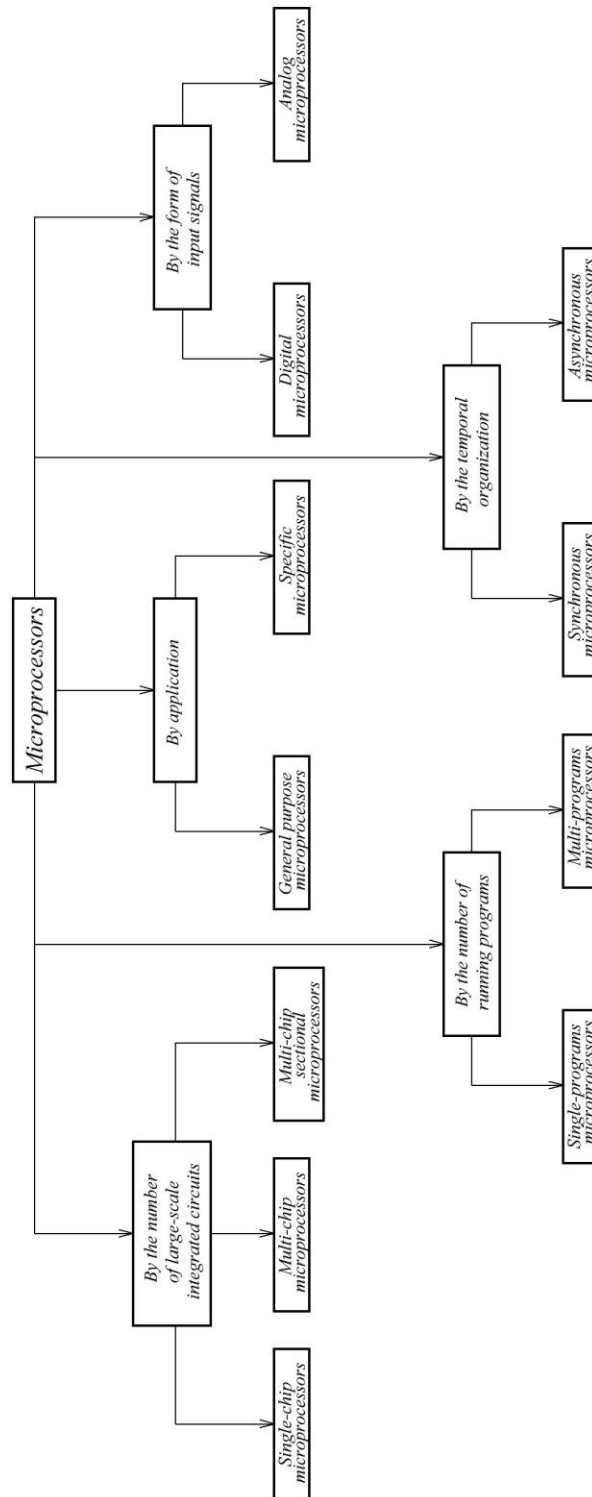


Fig 2.1. Classification of microprocessors

1. By the number of large-scale integrated circuits (LSIs) the microprocessors are divided into [2]:

- single-chip microprocessors;
- multichip microprocessors;
- multichip sectional microprocessors.

Single-chip microprocessors consist of one LSI or VLSI (very large-scale integrated circuit). The possibility of single-chip microprocessors is limited by the characteristics of the crystal and the case.

The multi-chip microprocessor is divided into the functional blocks. These functional blocks are implemented in the form of individual LSI or VLSI. All blocks of the microprocessor perform specific functions and can work independently.

Typically, a multichip processor is divided into the following functional blocks: an operating processor, a control processor and an interface processor.

The operating processor performs data processing. And the control processor is used for fetching, decoding, address computation and also for determining the sequence of commands. The interface processor is used to connect the memory with various peripheral and other devices and to provide direct memory access mode (DMA).

2. By their application the processors are divided into (Fig. 2.2) [2]:

- General purpose microprocessors;
- Specific microprocessors.

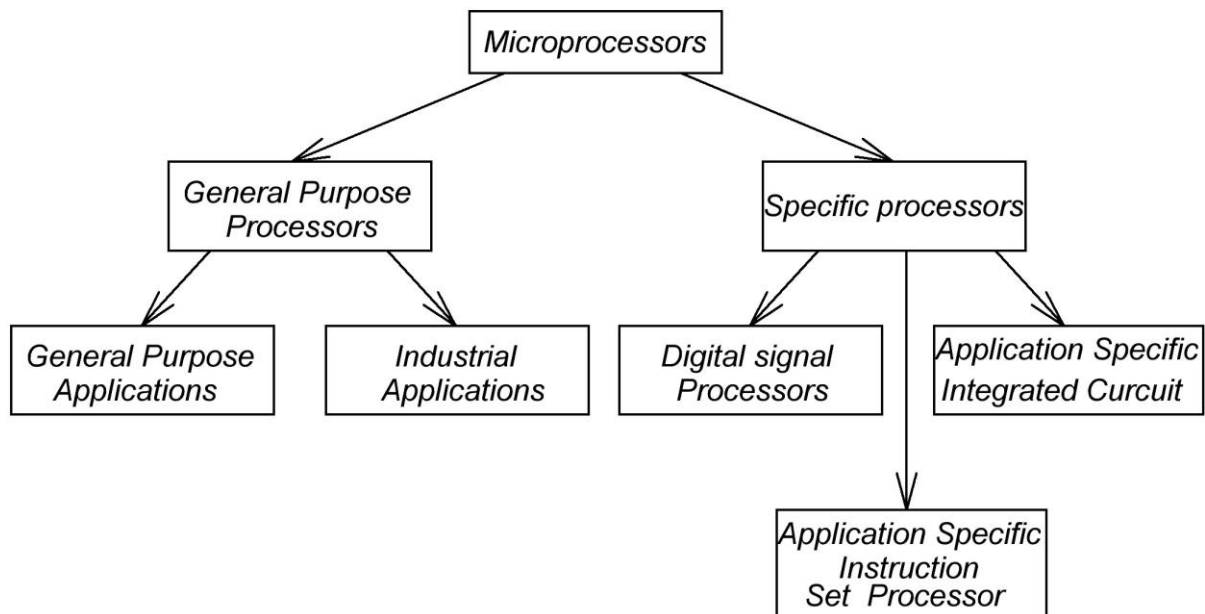


Fig. 2.2. Classification of microprocessors by application

General purpose microprocessors are designed for a wide range of tasks. The specific microprocessors targeted for specific functions can significantly increase the productivity. The field of application of specific microprocessors is wide. Typically, they are used for the control of complex technical devices and processes in manufacturing, transportation, communications, military industry, medicine, etc. A particular class of specific microprocessors are digital signal processors that have a high performance in analog signals processing.

Digital signal processors have the following characteristics:

- Multiply-accumulate units;
 - Multiple access memory architecture;
 - Specialized addressing modes: circular addressing, bit-reverse addressing;
 - Residual control / predicated execution;
 - Hardware loops / zero-overhead loops;
 - Restricted interconnectivity between registers and functional units;
 - Encoding restrictions.
3. By the form of processed input signals two types are distinguished [2]:
- digital microprocessors;
 - analog microprocessors.

In analog microprocessors, input signals are previously digitized. After that the processing is carried out in digital form and the result is converted into the analog form at the output.

4. By the temporal organization the microprocessors are classified into [2]:
- synchronous microprocessors;
 - asynchronous microprocessors.
5. By the number of running programs the following types are distinguished [2]:
- single-program microprocessors;
 - multi-program microprocessors.

Single-program microprocessors can perform only one program, and the implementation of the following programs is possible only after previous one. In the multi-program microprocessor, several programs can be performed at the same time, which allows to control a large number of sources and receivers of information.

2.3. THE CHARACTERISTICS OF MICROPROCESSORS

The main characteristics of microprocessor are:

1. ***The power of microprocessor.*** The power of microprocessor is its capability to process data. As a rule, this parameter is estimated by three main characteristics:
 - ***Data word size*** (the number of bits that can be processed at the same time). Data word size can be 4, 8, 16, 32 and 64 bits. This parameter is one of the main characteristics of microprocessors;
 - ***The number of addressable memory words or address bus size.*** It characterizes the capability of a microprocessor in accessing its memory. For 16-bit address bus the number of addressable memory words will be $2^{16} = 65536 = 65K$ and for 20-bit address bus - $2^{20} = 1048576 = 1M$. Each word in the memory has a number of its location that is called address. To get the words from the memory, computer refers to the corresponding address. Memory addresses start at zero and are represented in binary form. The different types of computers have different maximum memory addresses and it specifies the computing power of the microprocessor;
 - ***Speed.*** Most often, this parameter is stated as the time required for the microprocessor to execute one operation, in terms of the clock rate, or the access time;
2. ***The type of microelectronic technology*** used in the microprocessor manufacturing: p-channel MOS, n-channel MOS, CMOS, silicon-on-sapphire, bipolar TTL, Schottky TTL, I²L, ECL;
3. ***The number of crystals forming the microprocessor, the number of elements (transistors) on the crystal, the crystal dimensions;***
4. ***Package dimensions and number of pins;***
5. ***Form of control:*** in-system or micro-programming;
6. ***Efficiency of the command system, which is specified by*** the number of commands, possible addressing modes, the availability of commands working with stack memory and commands working with bits, decimal numbers, floating point numbers, etc.;
7. ***The number of interrupt levels;***
8. ***The possibility of working in direct memory access (DMA) mode;***
9. ***The number of power supply voltages;***
10. ***Power consumption and signal parameters;***
11. ***Service conditions*** (operating temperature range, relative humidity, etc.);

12. *Price of microprocessor.*

2.4. THE MICROPROCESSOR ARCHITECTURE

The microprocessor architecture is a logical organization of the microprocessor, which specifies its possibility of using it for hardware and software realization of the functions needed to build microprocessor systems.

The term “*microprocessor architecture*” characterizes:

- the set of components the microprocessor consists of, and the connections between them;
- the presentation medium and data formats;
- addressing to all (software available) structure elements that are available to the user (addressing to the registers, RAM cells and permanent memory, peripherals);
- command system of microprocessor;
- characteristics of control word and signals generated by the microprocessor and received by it;
- the response to external signals (interrupt processing circuitry, etc.).

There are two types of microprocessors: **RISC (Reduced Instruction Set Computing)** and **CISC (Complex Instruction Set Computing)**.

1. Complex Instruction Set Computer (CISC):

- large number of complex addressing modes;
- many versions of instructions for different operands;
- different execution times for instructions;
- few processor registers;
- microprogrammed control logic.

2. Reduced Instruction Set Computer (RISC):

- one instruction per clock cycle;
- memory accesses by dedicated load/store instructions;
- few addressing modes;
- hard-wired control logic.

Modern microprocessors have RISC-core and CISC-cover.

The processors with the **MISC (Minimum Instruction Set Computing)** type architecture with a minimal set of commands and a very high speed are currently being developed.

Figure 2.3 shows a simplified architecture for a standard von Neumann processor. When an instruction is processed in such a processor, units of the processor not involved at each instruction phase wait idly until control is passed on to them. Increase in processor speed is achieved by making the in-

dividual units operate faster, but there is a limit on how fast they can be made to operate [3].

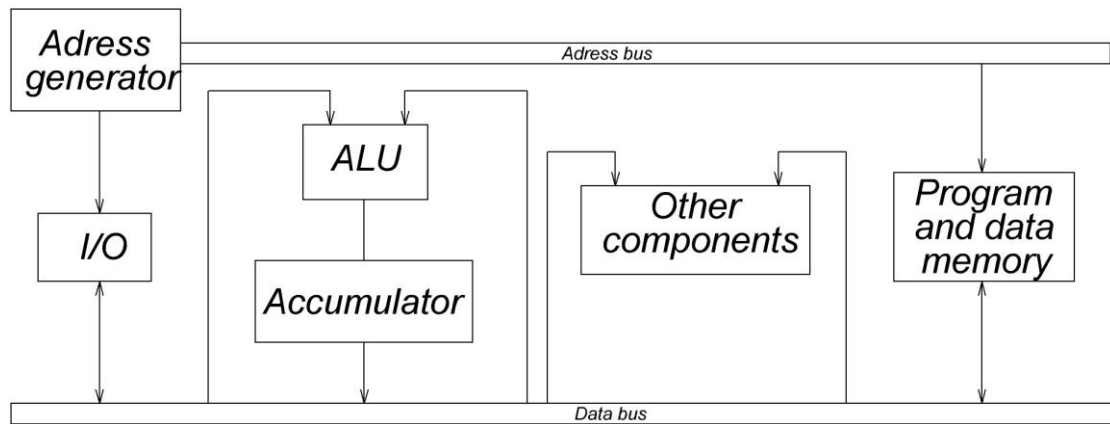


Fig. 2.3. A simplified architecture for standard microprocessors

The principal feature of the Harvard architecture is that the program and data memories lie in two separate spaces, permitting a full overlap of instruction fetch and execution [3].

Typically, each instruction involves three steps:

- instruction fetch;
- instruction decode;
- instruction execute.

In a standard processor, without Harvard architecture, the program instructions (that is, the program code) and the data (operands) are held in one memory space; see Fig. 2.4. Thus the fetching of the next instruction while the current one is executing is not allowed, because the fetch and execution phases each require memory access [3].

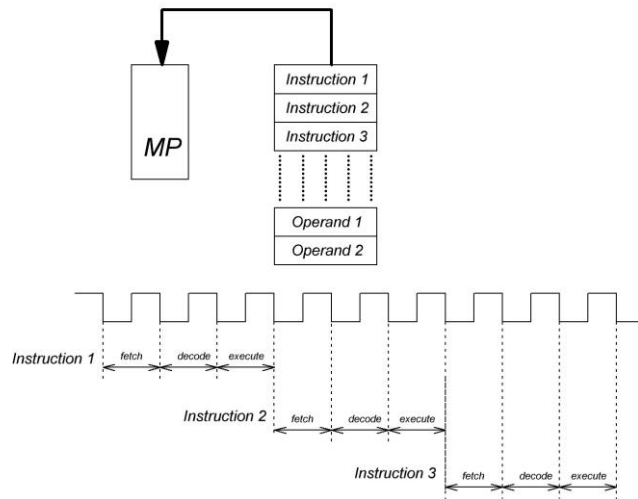


Fig. 2.4. Instruction fetch, decode and execute in a non-Harvard architecture with single memory space.

In a Harvard architecture (Fig. 2.5), since the program instructions and data lie in separate memory spaces, the fetching of the next instruction can overlap the execution of the current instruction; see Fig. 2.6. Normally, the program memory holds the program code, while the data memory stores variables [3].

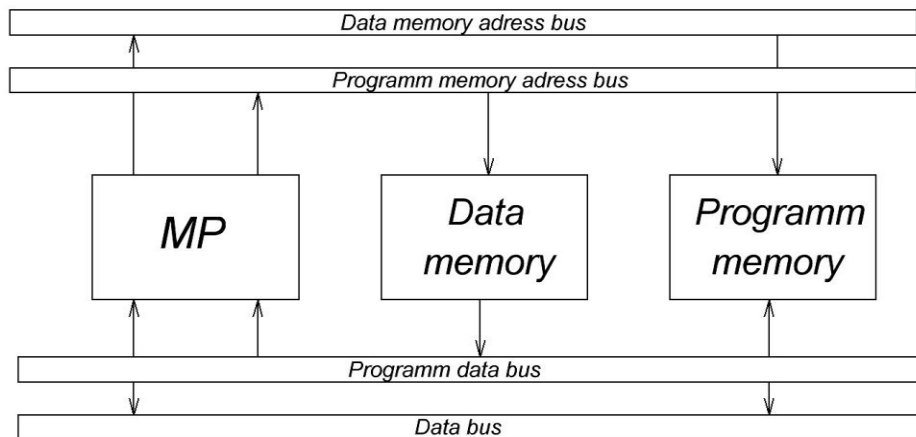


Fig. 2.5. Standard Harvard architecture

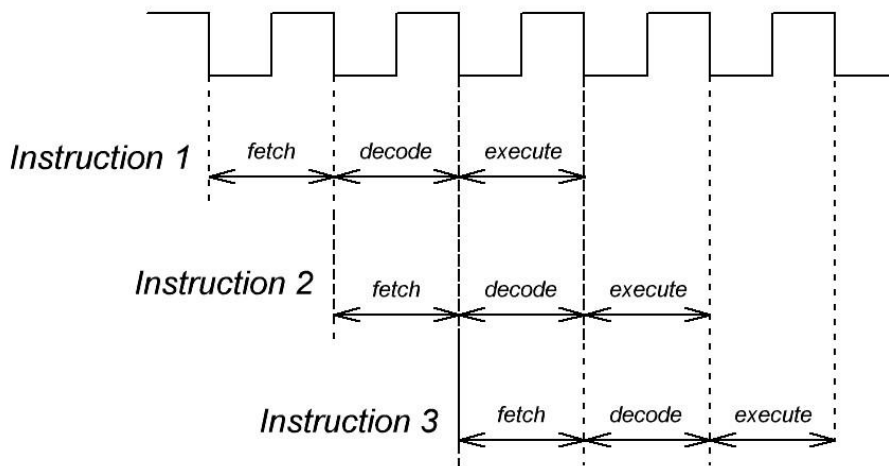


Fig. 2.6 Instruction fetch, decode and execute in Harvard architecture

2.5. BUS ORGANIZED STRUCTURE

To obtain maximum versatility and easy communication protocols, the bus organized structure of connections between devices within the system is used in microprocessor systems [4].

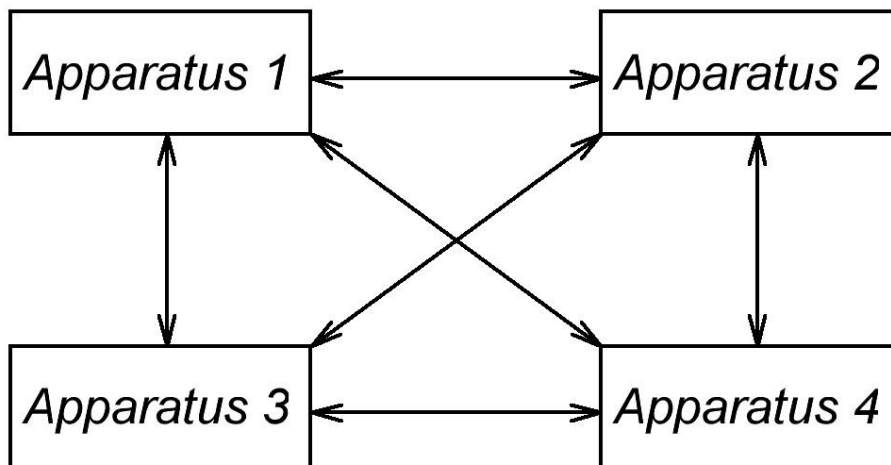


Fig. 2.7. Typical structure

In the typical structure of communication (Fig. 2.7) all signals and codes are transmitted between devices via separate lines. Each device included in the system transmits its signals and codes independently of other devices. In this case, the system has a lot of lines and different communication protocols.

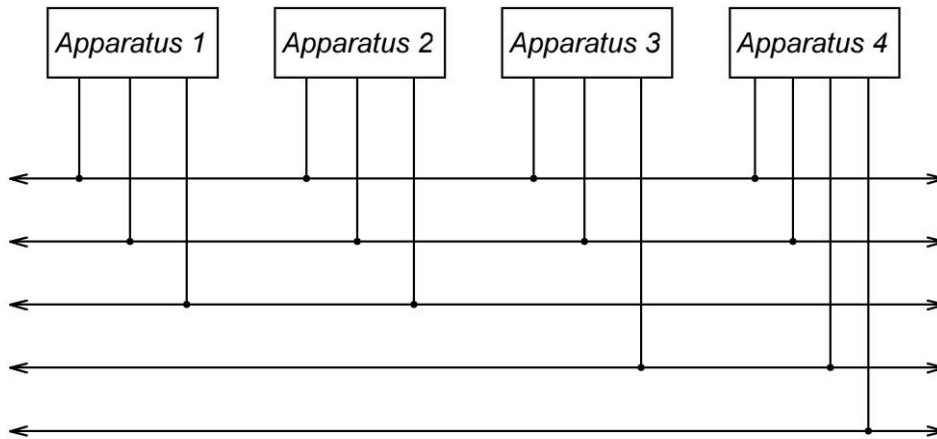


Fig. 2.8. Bus organized structure

In the bus organized structure (Fig. 2.8) all signals are transmitted between devices via the same lines of communication, but at different times (it is called a multiplexed transmission). And all the transmission lines can transmit addresses, data and control signals in both directions (it is called bi-directional transmission). As a result, the number of lines is significantly reduced, and the interchange rules (or protocols) are simplified. Group communication lines are called a *bus* [4].

With the bus organized structure it is easy to send all information flows in the right direction, for example, they can pass through a single processor and this is very important for the microprocessor system. However, with the bus organized structure all information is transmitted sequentially in time, which reduces the system performance compared to the classical structure.

One of the advantages of the bus organized structure is that all devices connected to the bus must transmit and receive information by the same rules (protocols for information exchange). So, all components of system should be uniform, standardized.

A significant disadvantage of the bus structure is related to the fact that all the devices are connected to each line in parallel. Therefore, any failure of any device can put the entire system out of operation. For this reason, check-out of bus structure system is rather complicated and usually requires special equipment [4].

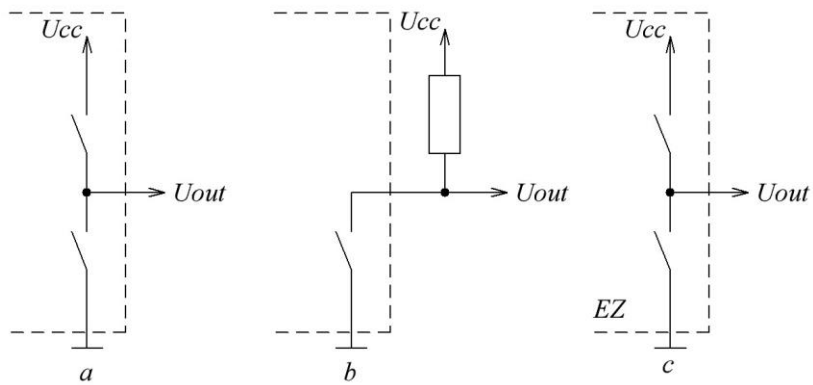


Fig. 2.9. Output types of digital microcircuits

In systems with bus structure all three existing types of output stages of digital microcircuits are used:

- Standard output (Fig. 2.9,a);
- Output with open collector (Fig. 2.9,b);
- Three-stages output (Fig. 2.9,c).

Digital microcircuits with open collector and three-stages output allow us to connect several outputs of microcircuits into multiplexed (Fig. 2.10,a) or bidirectional lines (Fig. 2.10,b).

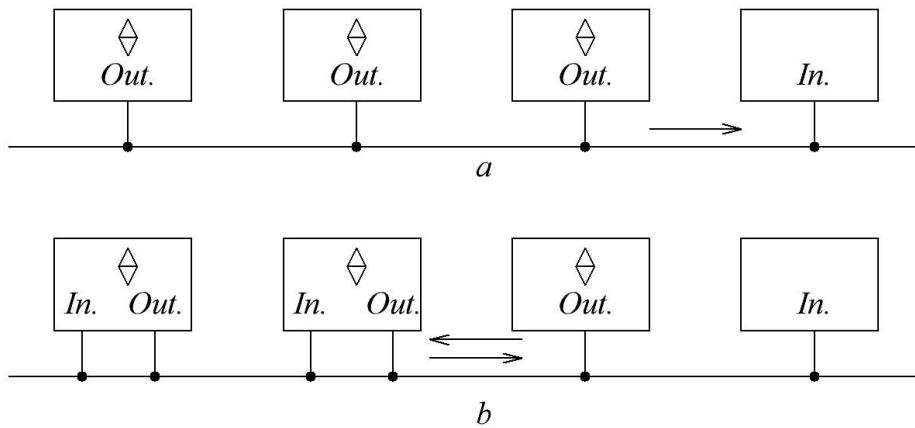


Fig. 2.10. Types of interconnections: a) multiplexed and b) bidirectional

2.6. THE STRUCTURE OF MICROPROCESSOR SYSTEM

Figure 2.11 shows a typical structure of microprocessor system.

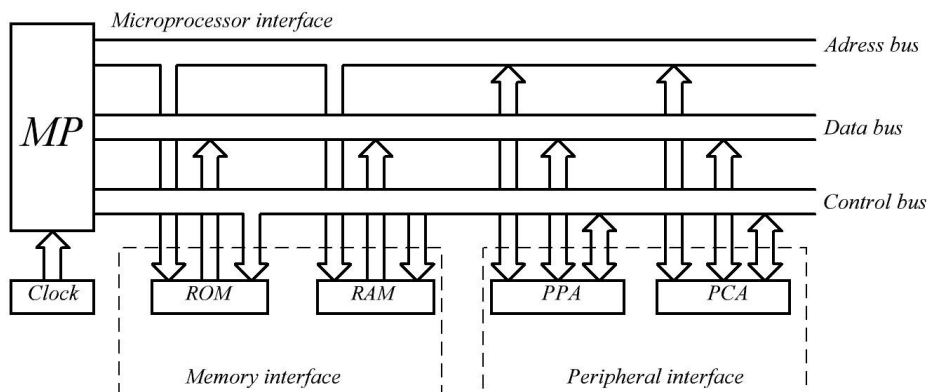


Fig. 2.11. The typical structure of microprocessor system

The central part of this structure is the microprocessor which performs ordinary operations:

1. arithmetic and logical operations with data;
2. control process of software for information processing;
3. organization of the communication of all devices in the system.

Microprocessor operation is specified by the synchronization signals. Individual functional blocks are standardized and finished modules with their control circuits and these blocks are constructed as one or several crystals of *LSI* or *ICI*.

Inter-module communication and interchange of information between modules are made via collective buses that are available to all the main modules of the system. At any moment in time only two modules of the system can exchange information. The exchange of information between the different modules occurs at different times.

The main principle requires:

- Information and logical compatibility of modules, which is realized through the use of common ways of presenting information;
- Control algorithm of interchange;
- Instruction formats;
- Synchronization method.

The memory section of the system is a place where digital data in binary form (1 and 0) are stored. The memory consists of cells organized in 8-bit groups. Each byte is given a unique numeric address, which represents its lo-

cation. Data are written into memory and read out of memory, based solely on their address.

Microprocessor systems usually have two kinds of addressable memory. The first is random-access memory (*RAM*), which allows the computer to read and write data at any of its addresses (it is also called read/write memory or *RWM*). All data in this type of memory are lost when the power is turned off and is called volatile memory. The second type of memory is read-only memory (*ROM*), which is similar to *RAM* except that new data cannot be written in; all data in *ROM* are loaded at the factory and cannot be changed by the programmer. This memory does not lose its data when power is turned off and is called non-volatile memory. Most microprocessor systems have both *RAM* and *ROM*. *RAM* is used for temporary program storage and as a temporary scratch-pad memory for the *CPU*. *ROM* is used to store programs and data that need to be always available. Actually, many microprocessor systems use an *EPROM* (erasable programmable read-only memory) or an *EEPROM* (electrically erasable programmable *ROM*) instead of a *ROM* for long-term memory. *EPROMS* can be erased with a strong *UV* light and reprogrammed. *EEPROMS* can be erased and reprogrammed electrically [5].

The input-output (*I/O*) section of the system allows it to interface with the outside circuits. The input section is the channel through which new programs and data are put into the system. And the output section allows the microprocessor to communicate its results. An *I/O* interface is called a port. An input port is a circuit that connects input devices to the system; examples of input devices are keyboards, sensors, and others. An output port is a circuit that connects the system to output devices. Examples of output devices are indicator lamps, actuators, and others [5].

Typically, a bus consists of 50 to hundreds of separate lines. On any bus the lines are grouped into three main function groups: data, address, and control. There may also be power distribution lines for attached modules.

The address bus is a group of wires that carries an address (in binary form) from the microprocessor to the memory and *I/O* circuits.

The address bus is used to determine the address (number) of the device with which the processor communicates presently. Each device (except the microprocessor), to each memory cell in the microprocessor system its own address is assigned. When the code of some addresses is exposed to the processor address bus, a device to which this address is assigned, understands that it is due to communicate. The address bus can be unidirectional or bidirectional [5].

The data bus is a main bus that is used to transfer information between all the devices of the microprocessor system. Usually, in the process of in-

formation transfer the processor transmits data to some device or memory or receives data from the device or from any memory cell. But the transfer of information between devices without the microprocessor is also possible. The data bus is always bidirectional.

The control bus, unlike the address bus and the data bus, consists of separate control lines. Each of these signals has its own function during the interchange of information. Some signals are used for gating the transferred or received data (that is, determining moments, when the information code is on the data bus). Other control signals can be used to confirm that the data have been received, to reset all devices to the initial state, to clock all devices, etc. Control bus lines can be unidirectional or bidirectional [5].

Power bus is necessary to power the system. It consists of supply lines and the common wire. In a microprocessor system, there can be one power supply (usually +5 V, 3.3 V or 1.8 V) and also additional power supplies (usually even -5 V, +12 V and -12 V). Each of them has its own voltage supply line connection. All devices connected to these lines are in parallel.

The peripherals used in microprocessor systems include displays, printing devices, analog to digital converters, digital to analog converters, etc.

The peripherals are connected to the buses of the microprocessor, not directly but through a programmable peripheral adapter (*PAP*) or a programmable connected adapter (*PCA*). A programmable peripheral adapter is necessary to transfer information in parallel code, whereas a programmable connected adapter is necessary to transfer information in serial code. Software configurable adapters make the input-output information system of microprocessor system more flexible and functional.

A common interface for memory and peripherals is necessary because of the limited number of pins (narrow interface), usually not more than 40, of microprocessor package.

Since the microprocessor interface is narrow it is necessary to use bidirectional data transmission lines, which is accompanied by the complication of buffer circuits. It is also necessary to use the time multiplexing of buses. Bus multiplexing allows one to transfer various information: address, data or commands. However, this requires the presence of special lines transferring identification information and reduces the information rate.

PART 3. INTEL 8080 MICROPROCESSOR

The *Intel* 8080 is a complete 8-bit parallel central processing unit. It is designed on a single *LSI* chip using Intel's n-channel silicon gate *MOS* process. This offers the user a high performance solution for control and processing applications. Figure 3.1 shows the block diagram of *I8080* microprocessor [6].

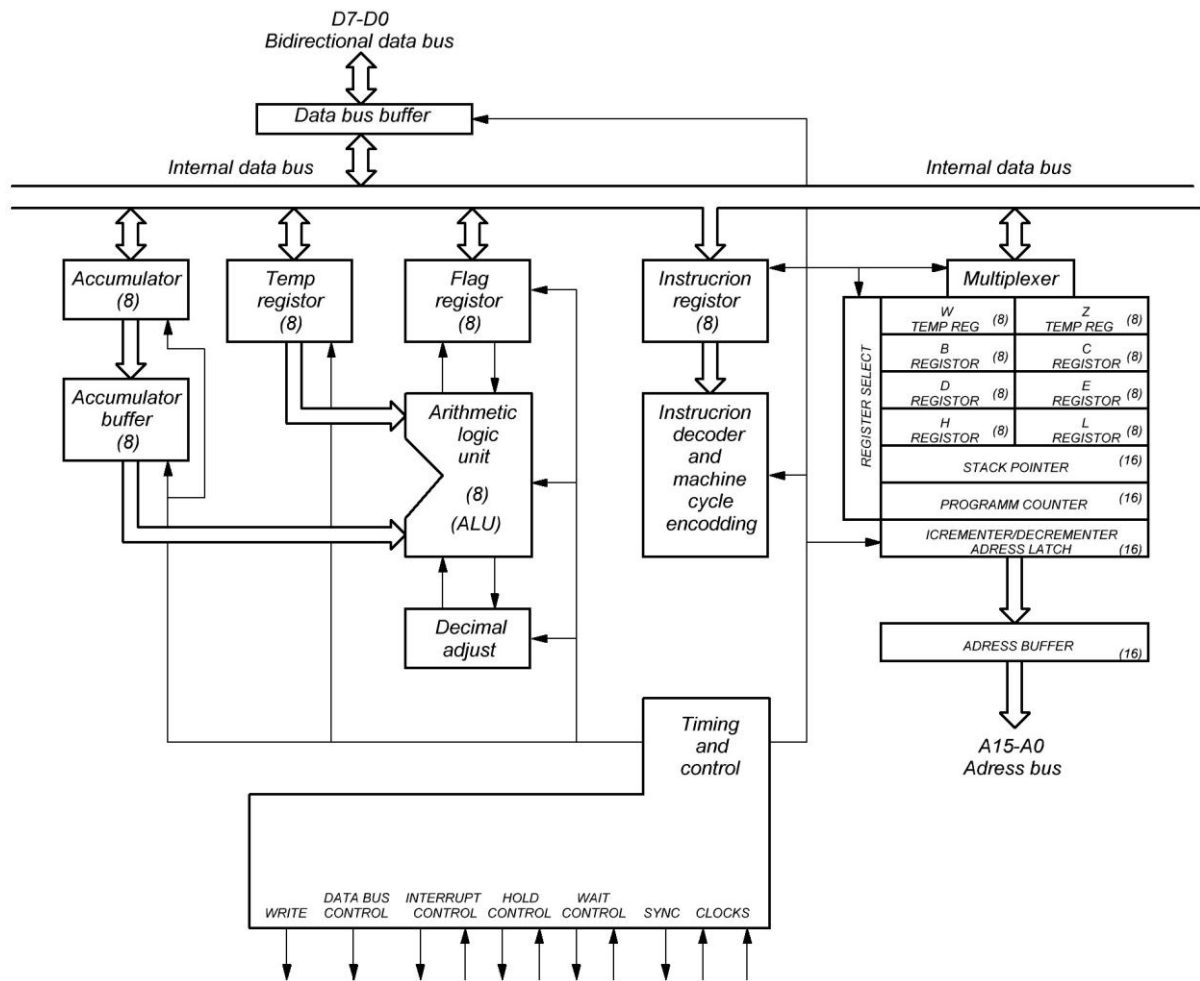


Fig. 3.1. The block diagram of *i8080* microprocessor

There are three internal buses in Intel 8080 microprocessor: a data bus, an address bus and a control bus.

- **Address Bus:** The address bus provides addressing to the memory (up to 64K 8-bit words) and to the 256 input and 256 output devices. A_0 is the least significant address bit.
- **Data Bus:** The data bus provides bi-directional communication between the *CPU*, memory, and *I/O* devices for processing instructions and data. Also, during the first clock cycle (state) of each machine cy-

cle, the *Intel* 8080A outputs a status word describing the current machine cycle, on the data bus. D_0 is the least significant bit.

- **Control bus** consists of 10 lines that are used to transmit control signals which determine the nature and functioning of the components of the *CPU*.

A) group of control signals of the state of microprocessor:

1. **RESET input signal.** While the **RESET** signal is activated, the content of the program counter is cleared. After **RESET**, the program will start at location 0 in memory. The **INTE** and **HLDA** flip/flops are also reset. Note that the flags, accumulator, stack pointer, and registers are not cleared;

2. **READY input signal.** The **READY** signal indicates to the 8080 that valid memory or input data is available on the 8080 data bus. This signal is used to synchronize the *CPU* with slower memory or *I/O* devices. If after sending an address out the 8080 does not receive a **READY** input, the 8080 will enter a **WAIT** state for as long as the **READY** line is low. **READY** can also be used to single step the *CPU*;

3. **WAIT output signal.** The **WAIT** signal acknowledges that the *CPU* is in a **WAIT**.

B) Signals group of control of the data and address buses:

1. **DBIN input signal.** The **DBIN** signal indicates to external circuits that the data bus is in the input mode. This signal should be used to enable the gating of data onto the 8080A data bus from memory or *I/O*;

2. **WR output signal.** The **WR** signal is used for memory **WRITE** or *I/O* output control. The data on the data bus are stable while the **WR** signal is active low ($WR = 0$);

3. **HOLD input signal.** The **HOLD** signal requests the *CPU* to enter the **HOLD** state. The **HOLD** state allows an external device to gain control of the 8080 address and data bus as soon as the 8080 has completed its use of these buses for the current machine cycle. It is recognized under the following conditions:

- the *CPU* is in the **HALT** state;
- the *CPU* is in the T_2 (second state) or T_w (wait state) state and the **READY** signal is active. As a result of entering the **HOLD** state the microprocessor Address bus ($A_{15}-A_0$) and Data bus (D_7-D_0) will be in their high impedance state;

- The *CPU* acknowledges its state with the **HOLD ACKNOWLEDGE (HLDA)** pin.

4. **HLDA output signal.** The **HLDA** signal appears in response to the **HOLD** signal and indicates that the data and address bus will go to the high impedance state. The **HLDA** signal begins at:

- T_3 for **READ** memory or input;
- The Clock Period following T_3 for **WRITE** memory or output operation. In either case, the **HLDA** signal appears after the rising edge of ϕ_2 .

C) group of interrupt signals:

1. **INT input signal.** The **CPU** recognizes an interrupt request on this line at the end of the current instruction or while halted. If the microprocessor is in the **HOLD** state or if the Interrupt Enable flip/flop is reset it will not honor the request;

2. **INTE output signal.** Indicates the content of the internal interrupt enable flip/flop. This flip/flop can be set or reset by the Enable and Disable Interrupt instructions and inhibits interrupts from being accepted by the **CPU** when it is reset. It is automatically reset (disabling further interrupts) at time T_1 of the instruction fetch cycle (M_1) when an interrupt is accepted and is also reset by the **RESET** signal.

D) Synchronization lines and supply

There are three synchronization lines in Intel 8080: ϕ_1 , ϕ_2 and **SYNC**.

Clock phases ϕ_1 and ϕ_2 are two externally supplied clock phases, (non TTL compatible).

Synchronizing signal SYNC. The **SYNC** pin provides a signal to indicate the beginning of each machine cycle.

Supply pins: +5 V, +12 V, -5v, **GND**.

Figure 3.2 shows the pin designations of **I8080** microprocessor [6].

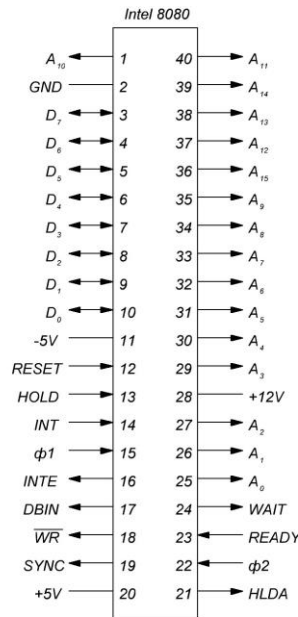


Fig. 3.2. The pin designations of *I8080* microprocessor

3.1. ARITHMETIC LOGIC UNIT (ALU)

The ALU contains the following registers [6]:

- An 8-bit accumulator (*ACC*);
- An 8-bit temporary accumulator;
- A 5-bit flag register: zero, carry, sign, parity and auxiliary carry;
- An 8-bit temporary register (*TMP*).

Arithmetic, logical and rotate operations are performed in the arithmetic logic unit. The *ALU* is fed by the temporary register, the temporary accumulator and carry flip-flop. The result of the operation can be transferred to the internal bus or to the accumulator; the arithmetic logic unit also feeds the flag register.

The temporary register receives information from the internal bus and can send all or portions of it to the arithmetic logic unit, the flag register and the internal bus.

The accumulator (*ACC*) can be loaded from the *ALU* and the internal bus and can transfer data to the temporary accumulator (*ACT*) and the internal bus. The contents of the accumulator (*ACC*) and the auxiliary carry flip-flop can be tested for decimal correction during the execution of the *DAA* instruction.

The functions of the arithmetic logic unit depend on the type of microprocessor, but the most common are the following: addition, subtraction, logical multiplication (*AND*), logical addition (*OR*), *XOR*, inversion, shifts to the right and to the left, increment and decrement.

3.2 MICROPROCESSOR REGISTERS

Registers are an important part of any microprocessor as they take part in the implementation of the basic logic functions, and are divided into special registers and general purpose registers (*GPR*) [6].

The number and function of registers depend on the architecture of the microprocessor. Intel 8080 has software-accessible 8-bit registers:

- accumulator;
- general purpose registers – *B,C,D,E,H,L*;
- flag register *F*.

It also has software -accessible 16-bit special registers:

- program counter (*PC*);
- stack pointer (*SP*);
- indirect address double register *HL*: *H* – high byte address register, *L* – low byte address register;

And software-inaccessible registers:

- 8-bit temporary registers *T, W, Z*;
- 8-bit instruction register *IR*;
- 16-bit address register *RA*.

We can also use the registers *B,C* and *D,E* as the double registers *BC* and *DE*.

3.3 ACCUMULATOR

Accumulator is the main register of microprocessor. Most arithmetic and logical operations are carried out by using the arithmetic logic unit and accumulator. Performance of any of these operations with two data words (operands) implies the placement of one of them in the accumulator, and the other in memory or another register. Thus, the result of addition of two words *B* and *D* located in the accumulator and memory, respectively, is placed into the accumulator, rewriting the word *B*. Therefore, the result of an *ALU* operation is always placed in the accumulator, and the initial content is lost [6].

Another function of an accumulator is a “programmable data transfer” from one part of the microprocessor to another. Performance of a "programmable data transfer" (data transfer) is carried out in two stages:

1. the data transfer from the source to the accumulator;
2. the data transfer from the accumulator to the receiver.

Also the microprocessor can perform some operations on the data directly in the accumulator. For example, the accumulator can be cleared by writing binary zeros into all its bits and can be set in one state by writing the

binary ones into all bits. The contents of the accumulator can be shifted to the left or to the right, can be inverted, etc.

There are no instructions for direct addressing of accumulator. An operation code indicates the use of an accumulator .

3.4. PROGRAM COUNTER (PC)

Program counter is one of the most important registers. As is known, a program is a sequence of instructions stored in the memory of a microcomputer, and they serve to instruct the microprocessor how to perform the task. If the instructions are fed in the correct order, the task is performed correctly. The program counter monitors which command is executed and which instruction will be executed next. Often the program counter has much more bits than the data word of microprocessor. Thus, in most 8-bit microprocessors, for addressing the memory of 65K, the number of bits of the program counter is 16. In any of the 65536 memory cells of micro-computers there can be general information about a particular step of the program, i.e. within the range of values between 0 and 65535 the program can begin and end in any cell. To access any of these addresses the program counter must have 16 bits. **Wherever the instruction is settled they follow each other in a certain order** [6].

When a microprocessor starts to work, the data from the memory area specified by the manufacturer of microprocessor is loaded in the program counter. Before starting the program it is necessary to place the starting address of the program in the memory area specified by the manufacturer. When the program starts the first value of the program counter will be this predetermined address. Unlike the accumulator the program counter cannot execute various types of operations.

Before the program starts it is necessary to load the program counter by the address of the memory where the first instruction of the program is located. Address register and memory address bus are located below the program counter. The memory address, where the first instruction is located, is sent from the program counter into the memory address register (*RA*), and the contents of both registers are identical [6].

Address location of the first program instruction is sent onto the address bus to the memory control circuit, as a result, the memory contents is read. Memory sends this instruction in a special register of microprocessor, called the instruction register (*IR*). After reading the instruction from memory, the microprocessor automatically increments the contents of the program counter. The program counter is incremented at the very moment when the microprocessor begins to execute an instruction. Consequently, the program coun-

ter indicates the address of the next instruction. The program counter holds the address of the next instruction during the time of execution of the current command. This is important because for programming micro-computers you may find it necessary to use the current counter value. It should be clearly understood that at any given time program counter does not indicate the address of current command and one after [6].

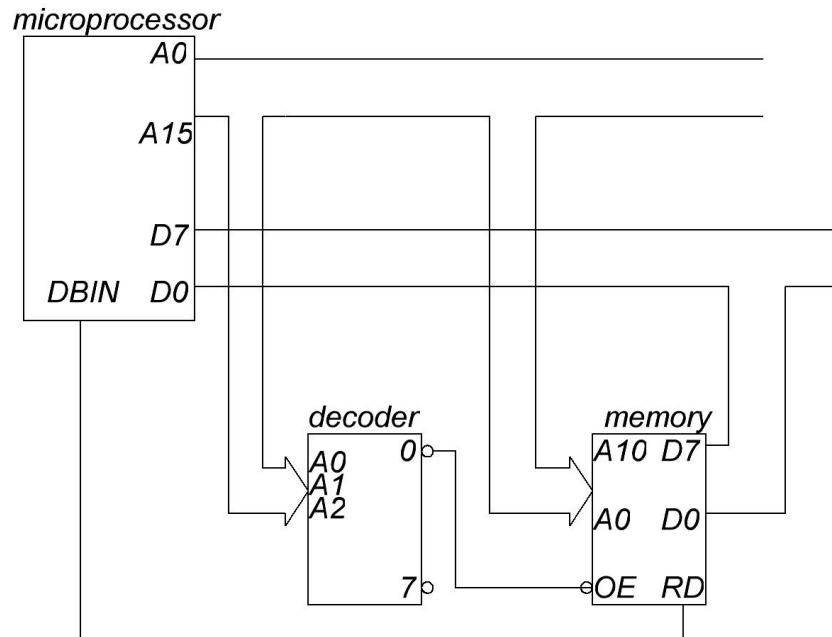


Fig. 3.3. Reading from memory structure

3.5. ADDRESS REGISTER

With each access to the micro-computer memory the address register indicates the address of the memory cell, which is to be used by the micro-processor. The output of this register is called the address bus and is used to select the memory cell or in some cases to select the input-output port.

During the sub-cycle of the instruction fetch from the memory, the address register and program counter have the same contents, i.e. the address register indicates the location of the instruction extracted from the memory. After decoding the command the program counter is incremented. **The memory address register is not incremented.**

During the sub-cycle of performance of the command the value of the address register depends on the current command. If the microprocessor must perform another memory access, the address register is to be reused in the process of this command. For some instructions addressable memory is not required (accumulator cleaning) and the address register is used only once.

Because the address register is connected to the internal data bus of the microprocessor, it can be downloaded from various sources. Most microprocessors have instructions that allow us to download the contents of this register from the program counter, general-purpose register or a memory cell [6].

3.6 INSTRUCTION REGISTER (IR)

The instruction register is intended for the storage of the current command that is being executed, and this function (storage) is performed by the microprocessor automatically with the start of fetching-executing cycle, that is called *machine cycle*. The instruction register is connected to the internal data bus, however, it only receives data and cannot send data to the bus. Although the functions of the instruction register are limited, its role in the microprocessor is great as the output of this register is a part of the instruction decoder. So, the first instruction is fetched from the memory, then the counter is configured to specify the instruction to be executed. After removing the instruction from the memory the copy of instruction is put on the internal data bus and sent to the instruction register. After that the instruction decoder reads the contents of the instruction register (IR) and informs the microprocessor what to do to execute the command. The number of bits of the instruction register depends on the type of microprocessor and usually coincides with the number of bits of the data word.

Example 3.1: Execution of a program:

JMP 0800h
MOV B,C
ADI 0C0h

The program memory is as follows:

Address	Code of operation
0000h	<i>JMP (0C3h)</i>
0001h	00 <i>LB</i>
0002h	08 <i>HB</i>
0003h	...
0004h	...
0800h	<i>MOV B,C</i>
0801h	<i>ADI</i>
0802h	<i>C0</i>

0803h	00
0804h	...

- After starting the system, or reset (**RESET**), the program counter is loaded by zero address. $(PC) \leftarrow 0000h$;
- This address is copied to the address register. And the code of operation of the first command is read into the instruction register: $(IR) \leftarrow 00C3h$;
- After that, the instruction decoder reads the contents of the instruction register (**IR**) and informs the microprocessor what should be done, and the address of the next instruction is written into the program counter: $(PC) \leftarrow 0003h$. The sequential reading of the code of operation and operands of the command occurs by incrementing the value of the address register;
- An instruction **JMP** provides a transition into the memory to the address 0800h, therefore, when it is executed a jump address is written in the program counter: $(PC) \leftarrow 0800h$;
- Then the code of operation located at 0800h is written to the instruction register . Since there is a single-byte instruction in this address, then 0801h is written to the program counter: $(PC) \leftarrow 0801h$;
- After the performance of the **MOV** command the code of operation of addition of the accumulator with 0C0h (**ADI 0C0h**) is written into the instruction register . The instruction decoder, analyzing that this command is a three-byte, writes 0804h to the program counter: $(PC) \leftarrow 0804h$. So the code of operation and operands of the command will sequentially read incrementing the value of the address register;
- etc.

3.7. FLAG REGISTER (F). STATUS REGISTER.

Flag Register is a specialized register. It is a set of memory cells (flip-flops) showing the state of the microprocessor. Each memory cell is a trigger, but the programmers prefer to call them **flags**. Each bit of this register contains the information that reflects the result of the last instruction of the program. This information is used for conditional jumps. In *Intel 8080* microprocessor individual condition code bits are defined as shown in Table 3.1 [6].

Table 3.1. Flag register of *Intel* 8080

Name	Rank	Description
S	7	<p>Sign. If the most significant bit of the result of the operation has the value 1, this flag is set; otherwise it is reset..</p> $\begin{array}{r} 0101\ 1110 \\ + 0101\ 1010 \\ \hline 1011\ 1000 \\ S=1 \end{array} \qquad \begin{array}{r} 0101\ 1110 \\ + 0001\ 1010 \\ \hline 0111\ 1000 \\ S=0 \end{array}$
Z	6	<p>Zero. If the result of an instruction has the value 0, this flag is set; otherwise it is reset.</p> $\begin{array}{r} 0101\ 1110 \\ + 0101\ 1010 \\ \hline 1011\ 1000 \\ Z=0 \end{array} \qquad \begin{array}{r} 1111\ 1111 \\ + 0000\ 0001 \\ \hline 1\ 0000\ 0000 \\ Z=1 \end{array}$
-	5	Not used and set to 0.
AC	4	<p>Auxiliary Carry: If the instruction caused a carry out of bit 3 and into bit 4 of the resulting value, the auxiliary carry is set; otherwise it is reset. This flag is affected by single precision additions, subtractions, increments, decrements, comparisons, and logical operations, but is principally used with additions and increments preceding a DAA (Decimal Adjust Accumulator).</p> <p>Example: Addition of two one-byte numbers.</p> $\begin{array}{r} 0110\ 1110 \\ + 0000\ 1000 \\ \hline 0111\ 0110 \\ AC=1 \end{array} \qquad \begin{array}{r} 0110\ 1110 \\ + 0000\ 0001 \\ \hline 0110\ 1111 \\ AC=0 \end{array}$ <p>The auxiliary carry flag is affected by all add, subtract, increment, decrement, compare, and all logical AND, OR, and exclusive OR instructions.</p> <p>The auxiliary carry flag and the DAA instruction allow us to treat the value in the accumulator as two 4-bit binary coded decimal numbers. Thus, the value 0001 1001 is equivalent to 19, if this value is interpreted as a binary number, it has the</p>

		<p>value 25. The following example shows that adding one to this value produces a non-decimal result:</p> $\begin{array}{r} 0001\ 1001 \\ + 0000\ 0001 \\ \hline 0001\ 1010 \end{array}$ <p>The <i>DAA</i> instruction converts hexadecimal values such as the <i>A</i> in the preceding example back into binary coded decimal format. The <i>DAA</i> instruction requires the auxiliary carry flag since the binary coded decimal format makes it possible for arithmetic operations to generate a carry from the low-order 4-bit digit into the high-order 4-bit digit. The <i>DAA</i> performs the following addition to correct the preceding example:</p> $\begin{array}{r} 0001\ 1010 \\ + 0000\ 0110 \\ \hline \end{array}$ $\begin{array}{r} 0001\ 0000 \\ + 0001\ 0000 \quad AC \\ \hline 0010\ 0000 \longrightarrow 20 \end{array}$
-	3	Not used and set to 0.
<i>P</i>	2	<p>Parity. If the modulo 2 sum of the bits of the result of the operation is 0, (i.e., if the result has even parity), this flag is set; otherwise it is reset (i.e., if the result has odd parity).</p> $\begin{array}{r} 0110\ 1110 \\ + 0000\ 1000 \\ \hline 0111\ 0110 \\ P=0 \end{array} \qquad \begin{array}{r} 0110\ 1110 \\ + 0000\ 0001 \\ \hline 0110\ 1111 \\ P=1 \end{array}$
-	1	Not used and set to 1.
<i>C</i>	0	<p>Carry. The carry flag is commonly used to indicate whether an addition causes a ‘carry’ into the next higher order digit. The carry flag is also used as a ‘borrow’ flag in subtractions. The carry flag is also affected by the logical <i>AND</i>, <i>OR</i>, an I exclusive <i>OR</i> instructions. These instructions set <i>ON</i> or <i>OFF</i> particular bits of the accumulator.</p> <p>The rotate instructions, which move the contents of the ac-</p>

		<p>cumulator one position to the left or right, treat the carry bit as though it were a ninth bit of the accumulator. <i>Example:</i> Addition of two one-byte numbers.</p> $ \begin{array}{r} 1110\ 1110 \\ + 1111\ 0000 \\ \hline 1\ 1101\ 1110 \\ C=1 \end{array} \qquad \begin{array}{r} 0110\ 1110 \\ + 0001\ 0001 \\ \hline 0111\ 1111 \\ C=0 \end{array} $ <p>An addition that causes a carry out of the high order bit sets the carry flag to 1, an addition that does not cause a carry resets the flag to zero.</p>
--	--	--

The contents of the accumulator and flag register *F* are called **processor status word (PSW)**. The presence of the carry flag allows one to organize at low bit processor the processing of data of any length by sequential treatment with byte operands. Also this register is used to create a closed loop by connecting the *MSB* and the *LSB* of the accumulator, which is necessary for the cyclic shift of the contents of the accumulator to the right or left according to the program.

3.8. BUFFER REGISTER OF ALU

Buffer ALU register is intended for temporary storage of one data word. This register is needed due to the absence of storage unit in the *ALU*. Arithmetic logic unit includes only combinational circuits, and therefore when receiving the initial data at the input of *ALU*, the result data immediately appears at its output as a result of the operations of this program. *ALU* must receive data from the internal bus of microprocessor, modify them, then put the processed data into the accumulator. But it is impossible without data temporary register. Buffer registers are available to the programmer.

Into buffer temporary register data can come only from the internal data bus, and into accumulator buffer the data can come from the output of the arithmetic-logical unit. Accumulator buffer allows one to avoid a situation when the input and output of the *ALU* are connected to the same point simultaneously [6].

3.9. GENERAL PURPOSE REGISTERS, REGISTER PAIRS

General purpose registers and register pairs are used to store the operands, intermediate and final results, etc. The **HL** register pair occupies a special place in *Intel* 8080 microprocessor and is called data/address register. It consists of two 8-bit registers which may be used together or separately.

They are labeled **H** and **L**, respectively, high and low bytes. When these two registers are used together we refer to a pair of **HL**. Registers **H** and **L** are universal similarly to the accumulator in the sense that they can be incremented, decremented, loaded with data and be a source of data. A pair of **HL** can also serve as an address register and store the destination address in the placement of data in memory, or the source of address in the commands of loading the accumulator. Thus, the registers **H** and **L** can be used to organize data manipulation and as an address pointer [6].

3.10. STACK POINTER (SP)

Stack is a reserved area of memory used to keep track of the program internal operations including functions, return addresses, passed parameters, etc [6].

The stack pointer SP is a 16-bit counter register, the contents of which is always the address. The function of stack is to save the current contents of the registers when the main program is interrupted. Figuratively speaking, a stack can be represented as written values on separate sheets of paper and folding them stacked. Removing always occurs in the reverse order. In other words, the principle of “last in - first out” (*LIFO*). At any time, the stack can include additional information. Using the stack we can organize nested subroutines. In this case, the main program calls a subroutine that can cause a new subroutine, etc. When we call the first subroutine, the return address to the main program is pushed to the stack. When we call the second subroutine, the return address to the first is again pushed to the stack, etc. As one performs subroutines the return address is popped from the stack as soon as it returns to the main program (Fig. 3.4).

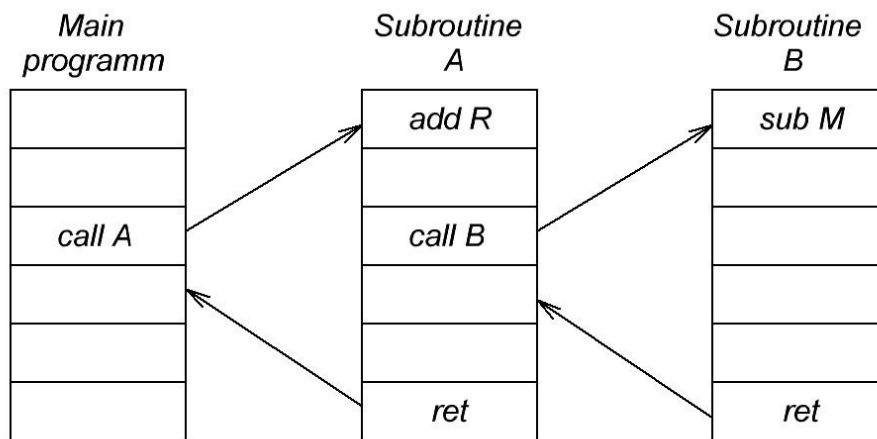


Fig. 3.4. Subroutines

However, the main purpose of the stack is to serve interrupts. When an interrupt takes place the contents of the *PC* and *F* as well as all current data from different registers are sequentially pushed into the stack. When an interrupt is completed, the program returns to the top of the stack and takes the data from the surface, respecting the principle of "last in - first out" (*LIFO*). Data are popped from the stack as long as the position is not restored before the interrupt, and as a result, the program returns to the final discontinued operations.

The microprocessor may have special registers for storing data, but more often it uses a designated sequence of memory cells. The Intel 8080 microprocessor as a stack uses a random area of *RAM*, and in the microprocessor only 16-bit register *SP* is placed. This register indicates the memory address of the top of the data list. The stack area is filled from higher to lower addresses, i.e. we can say that the stack grows up. Each call to the stack for writing the data is accompanied with the automatic reduction (decrement) of the contents of the stack pointer by 1, and each call to pop information from the stack is accompanied with the increment of stack pointer, i.e. addition of 1 to the contents of the stack pointer (Fig. 3.5).

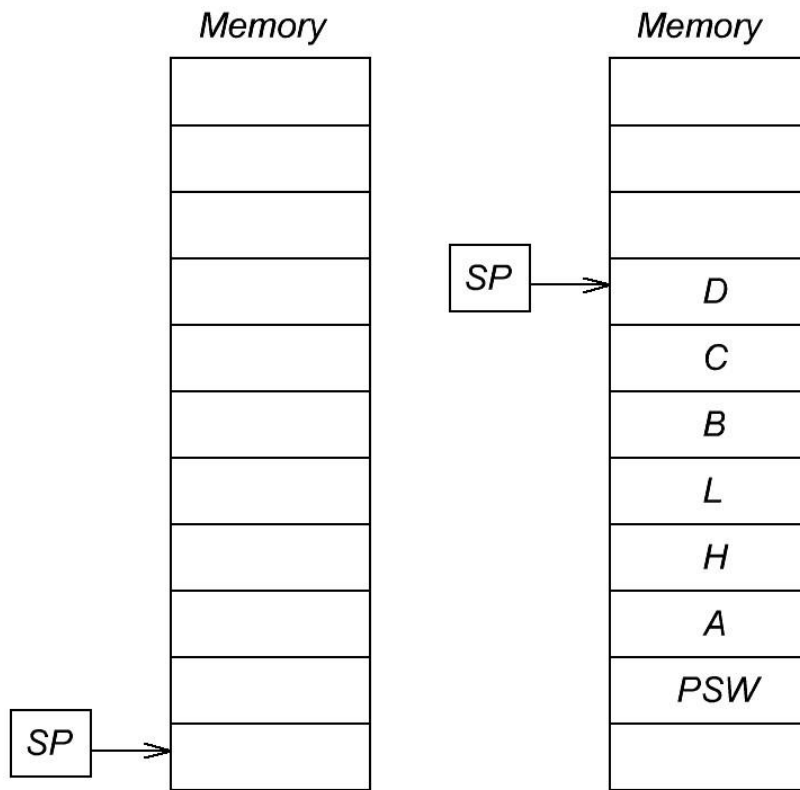


Fig. 3.5. Loading the stack of *Intel* 8080

Location of the stack is determined by the programmer. The stack pointer is loaded by the highest address, which is a top of the stack $220Ah$ (Fig. 3.6). Data can be written into the stack using the commands **PUSH** or **CALL**. And they can be read from the stack by the commands **POP** or **RET** (return).

Address	Data
$2204h$	
$2205h$	
$2206h$	
$2207h$	
$2208h$	
$2209h$	
$220Ah$	

$SP=220Ah$ →

Fig. 3.6. Stack

The algorithm for loading the stack (**PUSH**) is as follows (for example, pushing the contents of a register pair **HL** to the stack, Fig. 3.9):

1. The stack pointer is decremented from 220Ah to 2209h;
2. **SP** points to a memory cell 2209h on the address bus and a higher byte of data is pushed into stack;
3. The stack pointer is decremented from 2209h to 2208h;
4. **SP** points to a memory cell 2208h on the address bus and a lower byte of data is pushed into stack;

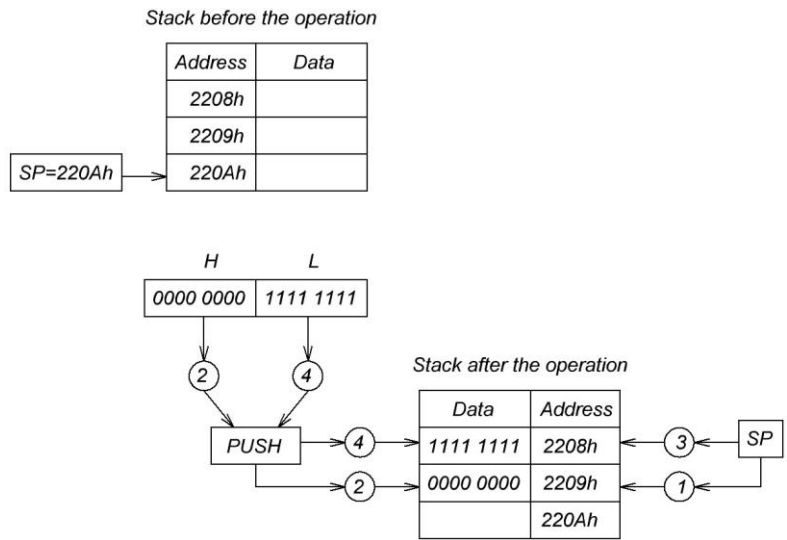


Fig. 3.9. Pushing the data from **HL** double register into stack

3.11. CONTROL CIRCUIT

The role of the control circuits on the chip is extremely important. The control circuit supports functioning sequence of all parts of microprocessor system. By means of the control circuits the next instruction is fetched from the instruction register, it is determined what has to be done with the data, and then generates a sequence of actions to implement the task.

Usually the operation of control circuits is programmed. One of the main functions of the control circuits is decoding the command located in the instruction register by the instruction decoder, which as a result gives the signals needed to execute the command.

One of the important input control lines that connects the microprocessor and external devices is a flow line with the clock generator which synchronizes microprocessor.

The control circuits perform some other special functions, such as control of interrupt processes. Interrupt is a kind of request incoming on the control circuit from different input-output devices. Interrupt occurs due to the

need of using the internal data bus of microprocessor by external devices. The control circuit determines when and in what sequence different devices can use the internal data bus [6].

3.12. STATUS WORD REGISTER

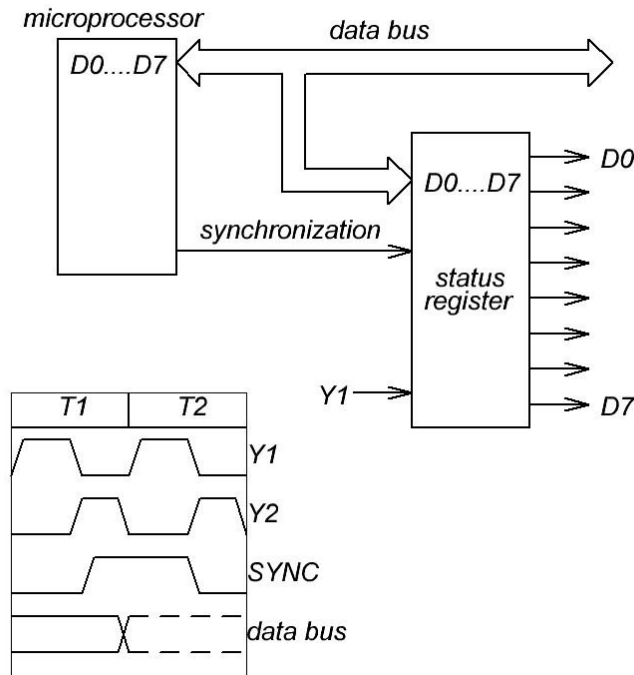


Fig. 3.10. Loading of status register

For the normal functioning of the microprocessor system, the control signals generated by the microprocessor are not enough. Microprocessor system in each machine cycle should be better informed about the state of microprocessor. For the case of narrow interface, when the external outputs for display of the internal state (status byte) of microprocessor are not enough, the multiplexing of data bus and the representation of the internal state in the external status word register are used.

The **SYNC** signal identifies the first state (T_1) in every machine cycle. As shown in Fig. 3.10, the **SYNC** signal is related to the leading edge of the Y_2 clock. There is a delay (t_{DC}) between the low-to-high transition of Y_2 and the positive-going edge of the **SYNC** pulse. There is also a corresponding delay (also t_{DC}) between the next 02 pulse and the falling edge of the **SYNC** signal. Status information is displayed on D_0 - D_7 during the same Y_2 to Y_1 interval. Switching of the status signals is likewise controlled by Y_2 .

The rising edge of Y_2 during T_1 also loads the processor's address lines (A_0 - A_{15}). These lines become stable within a brief delay (t_{DA}) of the Y_2 clocking pulse, and they remain stable until the first Y_2 pulse after state T_3 . This gives the processor ample time to read the data returned from memory.

If for any instruction the first clock cycle is the instruction fetch cycle, the other machine cycles can be in fairly random order, as determined by the operation code. Total *Intel* 8080 has 10 types of machine cycle and, accordingly, 10 codes of the status word, identifying these cycles. Each bit of the status word is put on the appropriate control input adapter or interface circuits with input/output device, thereby determining their mode of functioning in accordance with the current state of microprocessor.

Table 3.2. Status bit definitions

	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	
	<i>MEMR</i>	<i>IN</i>	<i>MI</i>	<i>OUT</i>	<i>HLTA</i>	<i>STACK</i>	<i>~WO</i>	<i>INTA</i>	
1	1	0	1	0	0	0	1	0	Instruction fetch
2	1	0	0	0	0	0	1	0	Memory read
3	0	0	0	0	0	0	0	0	Memory write
4	1	0	0	0	0	1	1	0	Stack read
5	0	0	0	0	0	1	0	0	Stack write
6	0	1	0	0	0	0	1	0	Input read
7	0	0	0	1	0	0	0	0	Output write
8	0	0	1	0	0	0	1	1	Interrupt acknowledge
9	1	0	0	0	1	0	0	0	Halt acknowledge
10	0	0	1	0	1	0	1	1	Interrupt acknowledge while halt

Thus, in a “narrow interface” the control of microprocessor system is implemented by generating a control action at two levels:

1. The control signals level $\sim WR$, $DBIN$, etc. (these signals are in each state);
2. Generating a status byte in each machine cycle.

The outputs of status register and control lines of microprocessor package form a control bus of microprocessor system. Twelve bus lines

provide the ability to control the microprocessor system with a complex of multi-function peripherals. The use of time division multiplexing of data bus for output of external status register control signals of microprocessor system reduces the system performance by 40%.

3.13. TIMING AND SYNCHRONIZATION OF THE MICROPROCESSOR SYSTEM

Every machine cycle within an instruction cycle consists of three to five active states (referred to as T_1 - T_5 or T_W). The actual number of states depends upon the instruction being executed, and on the particular machine cycle within the greater instruction cycle. The state transition diagram in Fig.3.11 shows how the *Intel* 8080 proceeds from state to state in the course of a machine cycle. The diagram also shows how the **READY**, **HOLD**, and **INTERRUPT** lines are sampled during the machine cycle, and how the conditions on these lines may modify the basic transition sequence. In the present discussion, we are concerned only with the basic sequence and with the **READY** function. The **HOLD** and **INTERRUPT** functions will be discussed later.

The first and required for all instructions is the machine cycle M_1 , which fetches the operational code from memory. In turn, each machine cycle requires three or five clock cycles: T_1 , T_2 , T_3 , T_4 , T_5 , two of which T_1 and T_2 are sent to the memory address, one T_3 , leads to the reading of the command or data from the memory, T_4 and T_5 correspond to executing of an instruction (see Fig. 3.12). Timing diagram defines the basic instruction cycle of microprocessor at a time when there is an external control signal **READY**, informing about the availability of peripheral equipment to the exchange with the microprocessor. In the first clock cycle T_1 microprocessor puts on address bus the next instruction address. Instruction fetch cycle begins. Simultaneously, on the synchronization line signal **SYNC** appears. And this signal:

1. Identifies information on the data bus as a status byte and loads it into the external status register;
2. Indicates the beginning of the machine fetch cycle.

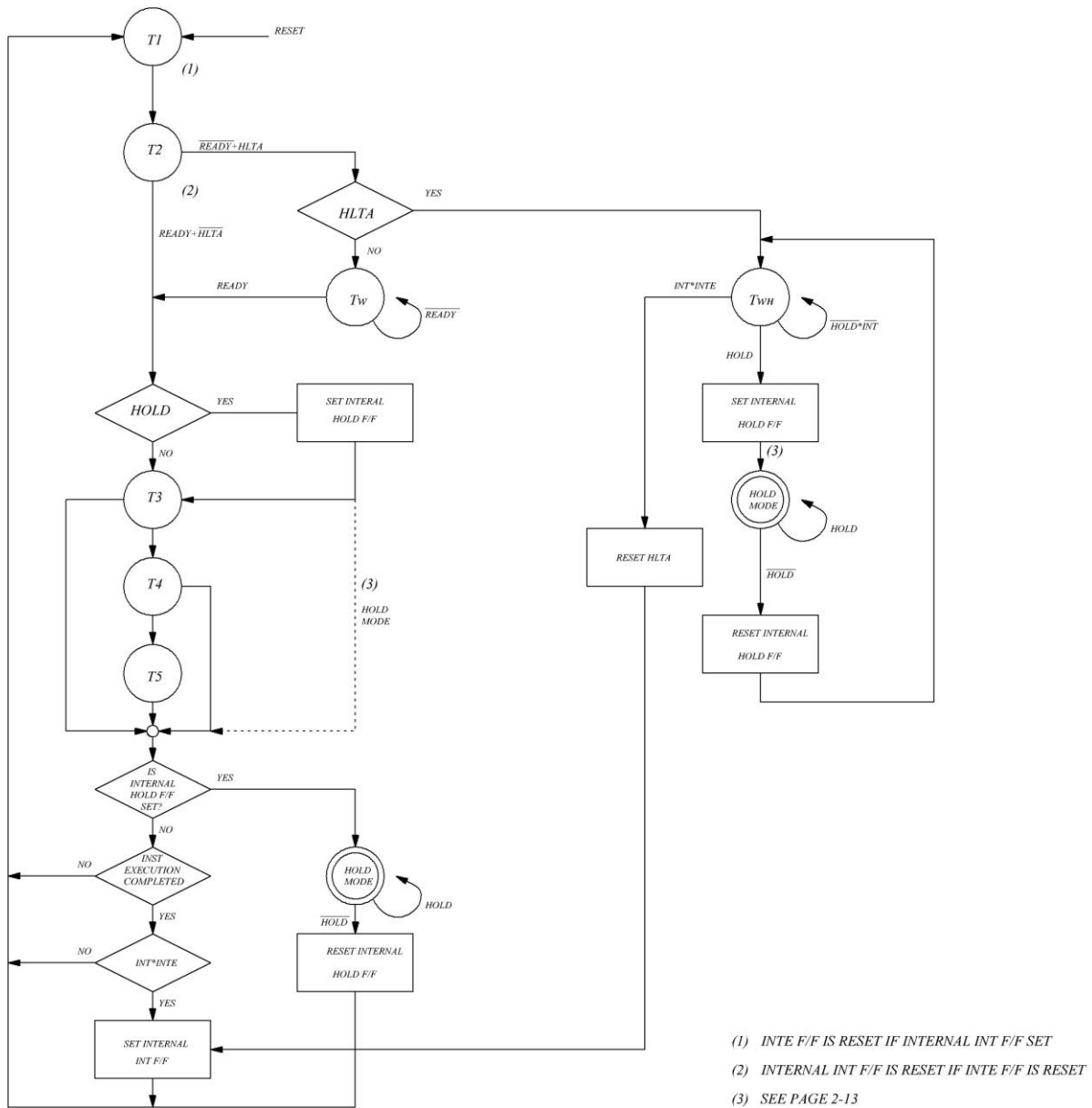


Fig. 3.11. The state transition diagram

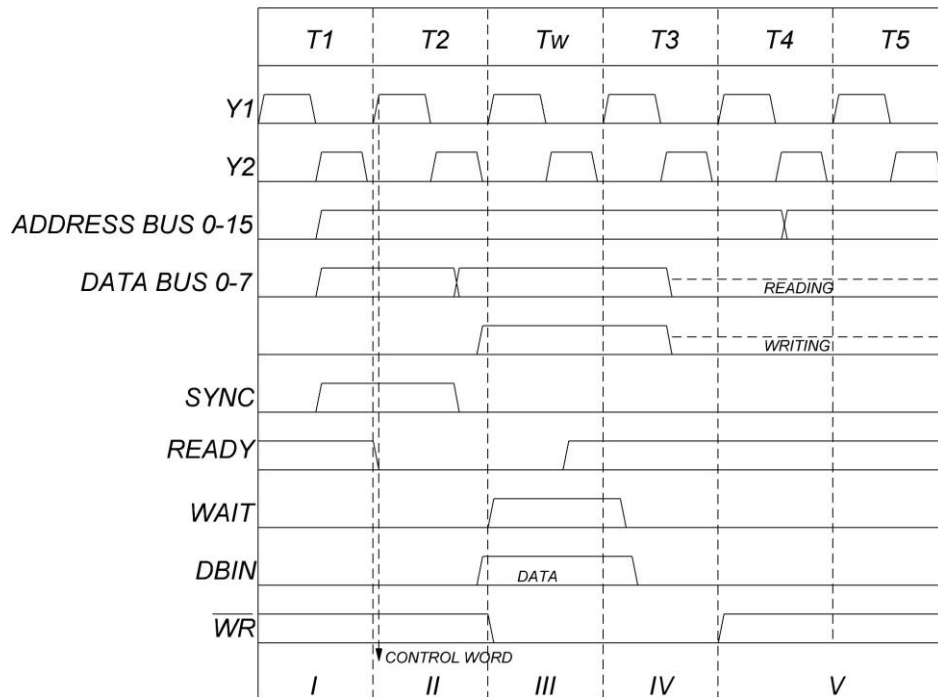


Fig. 3.12. The time diagrams of microprocessor:

- I – Formation of the control word
- II – Testing of control signals, RAEDY, HLTA, HOLD, INT
- III – Stopping or waiting of READY signal (not required)
- IV – Fetching of data, instruction or writing data
- V – Executing of command (not required)

After synchronization signal the data bus is in the input mode, as evidenced by signal **DBIN** (read) on control bus line. In cycle T_2 microprocessor checks an external device ready for exchange if the adapter memory or the external device generates a signal **READY**. Microprocessor goes into sleep mode. In this state, the microprocessor will be for as long as the control line **READY** does not appear, which would indicate that the memory or peripherals are ready to exchange. In cycle T_3 should be read or written words in memory, cycles T_4 and T_5 set for the execution of the operation specified by the instruction code.

3.14. OPERATION OF MICROPROCESSOR

We consider the actions that a microprocessor produces during the execution of program instructions.

A) Instruction cycle

In the *Intel* 8080 each instruction is executed in one to five machine cycles, which are called the processor cycles.

The number of instruction cycles is determined by the number of references to the external subsystems (memory, *I/O* and others) for the exchange of information between the microprocessor and the external addressable register. The only exception is the instruction ***DAD***, performed in three machine cycles with reference to the memory only 1 machine cycle. The first and required machine cycle of all instructions is fetching of operational code. Machine cycle consists of two phases. At the stage of required phase, called addressing phase, microprocessor addresses via the external register identifies and communicates between him and the microprocessor on the data bus. Not required phase, called executing phase, is decoding phase and the internal restructuring of the data phase.

Machine cycle consists of 3-5 states *T* called functional states or states of microprocessor. States are defined the time interval between two consecutive rising edges of clock signals *Y₁*, and their number is determined by the content in the microprocessor command being executed. Cycles can consist of 4-18 states.

The first three states of the microprocessor are unified and form an addressing phase. Addressing of external source register or recipient register is carried out in *T₁*. On the rising edge of the signal *Y₂* a microprocessor sets address signals on address bus (nominal delay stabilization of address signals is 200 ns, 270 ns max.) They remain stable until the rising edge of the clock *Y₂* follows the tact *T₃* of current machine cycle. The source of address can be the following registers: ***PC, SP, BC, DE, HL, WZ***.

T₂, which always follows the *T₁*, is taken to verify the necessary reactions to some control signals that affect the functioning of the microprocessor. On the rising edge of the signal *Y₂* external signals ***READY, DMA HOLD*** and an internal stop ***HLTA signal are verified***. Also in *T₂* of the last machine cycle of each instruction the level of external interrupt request signal *INT* is checked. Now we assume that these signals have such levels that do not alter the normal flow of machine cycle (***READY = 1, HOLD = 0, HLTA = 0, INT = 0***).

Depending on the features of the current machine cycle, the following actions will be performed in the *T₂* state:

1. If the current machine cycle is associated with an approaching to the memory, then the increment of the ***PC*** will be. ***PC = PC+I***;
2. If the current machine cycle is associated with reading (input) of data to the microprocessor, then the rising edge of the read signal is generated *Y₂ DBIN* (nominal delay of 130 ns, the maximum - 200 ns.).
3. If the current machine cycle is associated with the writing (output) of data from the microprocessor, then on the rising edge of the signal *Y₂*

formed output word signals (nominal delay of 220 ns, the maximum - 280 ns.).

State T_3 is required for the exchange of information between the microprocessor and the internal register addressed in the T_1 . When in the current machine cycle entering data in the microprocessor are carried out, the data signals from an external registry, switched by **DBIN** signal, should be stabilized at least 50 ns before the falling edge of Y_1 and remain stable for at least 130 ns after the rising edge of the clock Y_2 of T_3 state. If the current machine cycle of the microprocessor outputs data on the rising edge of Y_1 is generated L-active signal $\sim \mathbf{WR}$ (nominal delay of 70 ns, the maximum delay - 120 ns). **DBIN** signal is removed by the rising edge of Y_2 (with a maximum delay of 200ns), and $\sim \mathbf{WR}$ signal is removed by the next rising edge of the signal Y_1 (with a maximum delay of 120 ns).

In the T_4 and T_5 states of the first machine cycle of all instructions decoding of operational codes, necessary internal transfer and transform data are made.

These principles of formation of machine cycles allow one to determine the number of machine states in each command.

Example 3.2:

The single-byte command **MOV r2, r1**, is performed in a one machine cycle:

- a fetching of operation code (addressing phase);
- the transfer between the internal registers (execution phase).

Example 3.3:

The single-byte commands **MOV r, M** or **MOV M, r** are performed in a two machine cycles:

- a fetching of operation code (addressing phase – M_1);
- the transfer between the internal registers and memory location pointed by **HL** register pair (M_2).

Example 3.4:

ORI two-byte command consists of three machine cycles:

- a fetching of operation code (addressing phase – M_1);
- reading the second byte of the operand (M_2);
- executing of instruction (M_3).

Example 3.5:

Three-byte command **STA** consists of four machine cycles:

- three cycles of the instruction fetch;
- saving the contents of accumulator **A** in the memory cell.

Example 3.6:

One byte, but the longest instruction *XTHL* consists of five machine cycles:

- a fetching of operation code;
- two machine cycles for transmitting the contents of the top two cells of the stack in registers *W, Z*;
- two machine cycles for transmitting the contents of registers (*H, L*) in stack and transmitting the contents of (*W, Z*) in (*H, L*).

Example 3.7:

The number of machine cycles of eight conditional jump instructions (*SS, CNC, CZ*, etc.) depends on the values of their checked conditions, i.e. the state of the corresponding trigger flag:

- if the condition, which is analyzed in the T_4 and T_5 of first machine cycle is not satisfied, it is executed during cycles of M_1, M_2, M_3 ;
- if the condition is satisfied, the conditional jumps are executed in five machine cycles. In two additional cycles (M_4 and M_5) the contents of the program counter is booted from the stack.

B) Status byte.

We can assume that microprocessor system based on Intel 8080 consists of five external subsystems:

1. memory subsystem;
2. stack subsystem;
3. input subsystem;
4. output subsystem;
5. interrupt subsystem.

All subsystems are connected to the address bus and the data bus in parallel, and therefore the external register address, formed in T_1 , is simultaneously decoded by all subsystems.

However, in any machine cycle, the microprocessor can interconnect only with one subsystem. Therefore, the address information on the address bus is not enough to uniquely identify the external register and it is necessary to supplement "significant bytes" for addressing the subsystem. In addition, for debugging of microprocessor system the information on the current status of the microprocessor is usually required. Such additional information includes the status byte, which is set on data bus in each T_1 of all machine cycles on the rising edge of Y_2 . Pulse signal on the line identifying the status byte on data bus is the output SYNC signal, also generated in each T_1 of all machine cycles on the rising edge of Y_2 . Stable levels of status signals are saved until the rising edge of Y_2 in T_2 state. Acceptable delays are:

- nominal - 220 ns, maximum - 280 ns for status signals;

- nominal - 130 ns, maximum - 200 ns for *SYNC* signal.

Therefore, the *SYNC* signal can be directly used to load the status byte from the data bus to an external latch register, where it is saved until the next machine cycle.

The individual bits of the status byte has the following contents:

D_0 (*INTA*) — Acknowledge signal for INTERRUPT request. Signal should be used to gate a restart instruction onto the data bus when *DBIN* is active.;

D_1 (*WR#*) — Indicates that the operation in the current machine cycle will be a *WRITE* memory or *OUTPUT* function (*WO* = 0). Otherwise, a *READ* memory or *INPUT* operation will be executed;

D_2 (*STACK*) — Indicates that the address bus holds the pushdown stack address from the Stack Pointer;

D_3 (*HLTA*) — Acknowledge signal for *HALT* instruction;

D_4 (*OUT*) — Indicates that the address bus contains the address of an output device and the data bus will contain the output data when *WR* is active;

D_5 (*MI*)— Provides a signal to indicate that the *CPU* is in the fetch cycle for the first byte of an instruction;

D_6 (*INP*) — Indicates that the address bus contains the address of an input device and the input data should be placed on the data bus when *DBIN* is active;

D_7 (*MEMR*) — Designates that the data bus will be used for memory read data.

The main purpose of the status bits is generating the control signals for the external subsystems. Therefore, their definition is dictated by the simplicity of interface. There are 10 types of machine cycles which are given in Table 3.2.

We present internal memory pointers for all types of machine cycles, the contents of which is on the address bus:

1. The cycles of the program counter: fetching of operational code, interrupt, breakpoint, breakpoint interrupt;
2. Register pairs *BC*, *DE*, *HL*. Cycles: reading from the memory, writing to the memory;
3. The stack pointer cycles: reading from the stack, writing to the stack;
4. Register pair - cycles: input, output.

When commands *IN* and *OUT* run 1-byte address of the input or output port, which is the second byte $\langle B2 \rangle$ of commands in the machine cycle M_2 loaded in parallel into the registers *W* and *Z*.

In machine cycle M3 port address is given at the same time on the line of the $A_{15} - A_8$ and $A_7 - A_0$. This method is used to equalize the load on the address lines, as one of the ports can be connected to the lines $A_{15} - A_8$, and the other - to the lines of the $A_7 - A_0$. Knowing the contents of instruction, types of machine cycles and the organizing principle of the machine cycle, it is easy to construct a sequence of types of machine cycles for any instruction.

Example 3.8:

Command *MOV r₁,r₂* ($r_1, r_2 \neq M$) has one machine cycle:

- «FETCHING OF OPERATIONAL CODE».

Example 3.9:

Command *MOV M,r* has two machine cycles:

- «FETCHING OF OPERATIONAL CODE»;
- «WRITING TO THE MEMORY».

Example 3.10:

Command *OUT* has three machine cycles:

- «FETCHING OF OPERATIONAL CODE»;
- «READING FROM THE MEMORY»;
- «OUTPUT».

Example 3.11:

Command *LDA* has four machine cycles:

- «FETCHING OF OPERATIONAL CODE»;
- «READING FROM THE MEMORY»;
- «READING FROM THE MEMORY»;
- «READING FROM THE MEMORY».

Example 3.12:

Command *CALL* has five machine cycles:

- «FETCHING OF OPERATIONAL CODE»;
- «READING FROM THE MEMORY»;
- «READING FROM THE MEMORY»;
- «WRITING TO THE STACK»;
- «WRITING TO THE STACK».

In Fig. 3.12 the basic circuit for forming the control signals that are used in microprocessor systems based on Intel 8080 microprocessor is shown. One can see from the figure that all the control signals are L-active. This encoding simplifies the interface with external subsystems, which are generally L-active selection device inputs *DS* or chip select inputs *CS*.

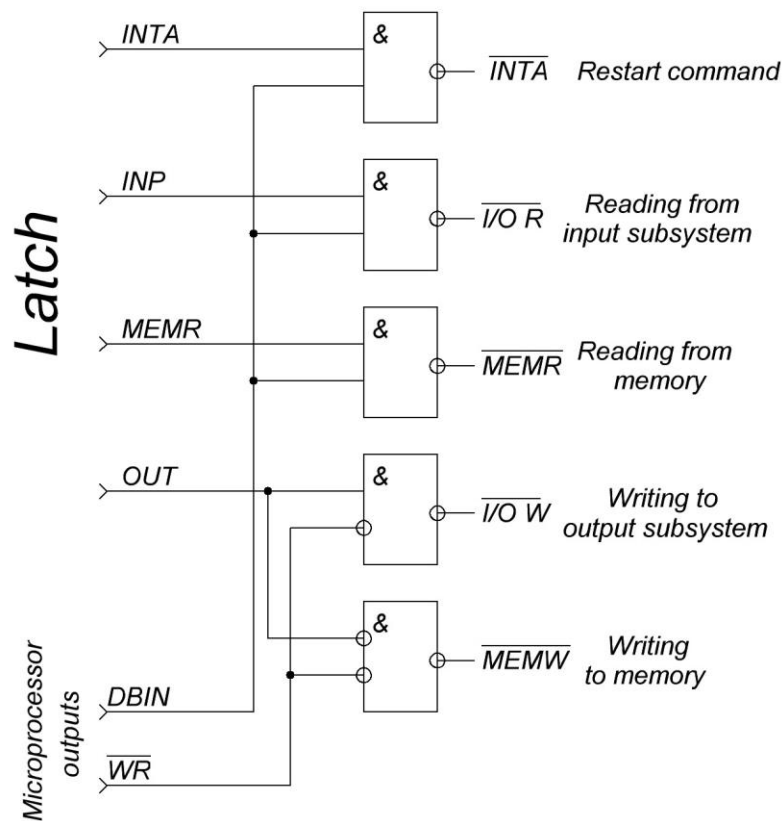


Fig. 3.12. Formation control signal circuit [6]

C) Special machine cycles

It was noted earlier that in T_2 state of machine cycles microprocessor checks the values of the input signal readiness **READY**, request of direct memory access **HOLD**, and the value of the internal acknowledge signal of breakpoint **HLTA**. In addition, in the last machine cycle of all instructions checks the level of the interrupt signal **INT**. Consider the reaction of the microprocessor in the specified signals.

3.15. REACTIONS OF MICROPROCESSOR ON SIGNAL **READY**.

When the speed of the external subsystems, such as memory, not enough for synchronous communication with the microprocessor, it can suspend the action of the microprocessor unit by setting L -level to line **READY**, i.e., the “stretch” the machine cycle for integer number of periods of synchronization. To respond to the microprocessor on L -level **READY** in the current machine cycle this level should be stabilized for at least 180 ns before the falling edge of the signal Y_2 and kept stable until the end of the pulse Y_2 .

Subject to this condition microprocessor does not go to the state T_3 and enters the standby state T_w (Fig. 3.13). In this state on the address bus remains the address of external register and **DBIN** signal if the current cycle of M_i associated with transfer of data to the microprocessor. For the acknowledgment of transition to the T_w states microprocessor generates the **H**-level on line **WAIT** on the rising edge of the signal Y_1 .

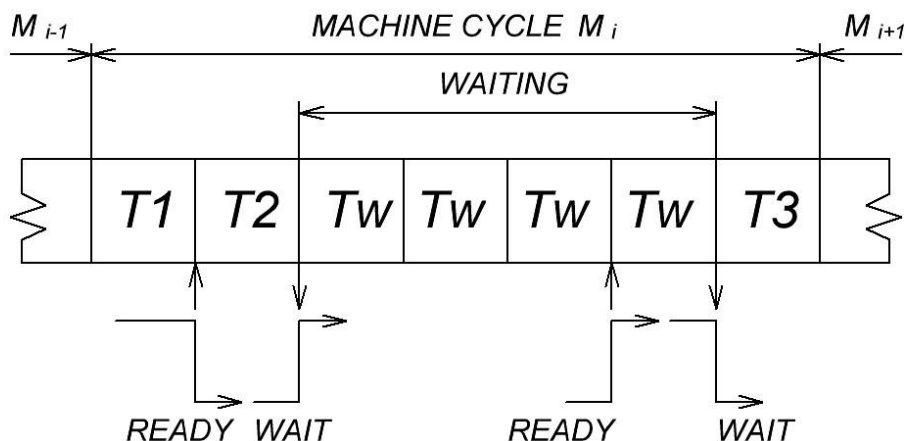


Fig. 3.13. Reactions of the microprocessor on signal **READY**

Duration of wait states is determined by the action of **L**-level on **READY** input. When the external subsystem is ready to exchange data, it should set line **READY** in **H**-level. To respond to the readiness of the microprocessor, the **H**-level should be stabilized at least 180 ns before the falling edge of Y_2 . The microprocessor then introduces T_3 and on the rising edge of F_1 removes the **WAIT** signal.

The nominal delay of the start and end of the **WAIT** signal relative to the rising edge of Y_1 is 70 ns and 120 ns maximum. Of course, the introduction of wait slightly reduces the performance of the microprocessor system [6].

3.16. REACTIONS OF MICROPROCESSOR ON SIGNAL **HOLD**.

If in the microprocessor system there are peripherals with high speed data transmission, such as a floppy disk, interchange between them organized in a direct memory access mode (**DMA**). When a peripheral device (**DMA** controller) initiates a request on **DMA**, the microprocessor suspends the execution of the program and transfers the internal buffer address and data buses in a high impedance state. **DMA** controller begins to control the buses, organizing the exchange of data between the peripheral devices and the memory of microprocessor system.

Initiating of **DMA** is accomplished by setting the *H*-level to the input **DMA** request lines **HOLD**. *H*-level of **HOLD** signal maintains until data block end, but the **DMA** can be implemented for transmission of individual bytes. The microprocessor responds to the *H*-level **HOLD** signal in the current machine cycle M_i if this level has stabilized for at least 180 ns before the rising edge of Y_2 (Fig. 3.14). They recommend **HOLD** signal to synchronize with the signal Y_1 .

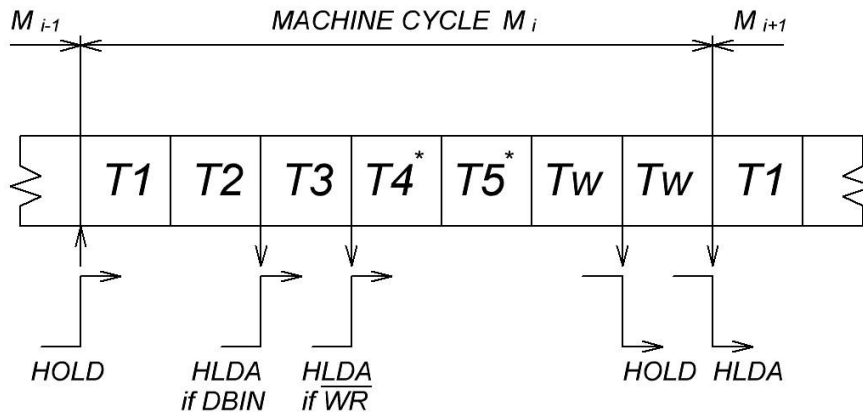


Fig. 3.14. Reactions of microprocessor on signal **HOLD**

Recall that the microprocessor in T_1 state addresses the external register, but the exchange with it will be only in state T_3 . If one allows for **DMA** before T_3 state, then in the future one would have to re-address the external register, which would complicate the control device and the timing diagram of the microprocessor. Therefore, the microprocessor necessarily ends interchanges with external register and only then enters the **DMA** states T_w . Note that such mode **READY** signal has higher priority than the **HOLD** signal.

After a T_3 state current machine cycle M_i can follow not required states T_4 and T_5 to be allocated for the internal transformation of the data, or T_1 state of next machine cycle. The microprocessor generates the *H*-level **HLDA** signal of **DMA** request on the rising edge of Y_1 (nominal delay of 70 ns, 120 ns max) and on the rising edge of Y_2 with a delay no more than 200 ns translates buffers of the address and data buses in a high impedance state. If in current machine cycle M_i is reading into the microprocessor (valid signal **DBIN**), then generates **HLDA** signal and disconnect address and data buses is in T_3 state, and if in current machine cycle M_i is writing from the microprocessor (valid signal \overline{WR}), the same steps are carried out in state following after the T_3 state, and if the current machine cycle contains T_4 and

T_5 states they are combined with the **DMA**. After the end of the direct memory access mode the microprocessor enters state T_1 always of the next machine cycle M_{i+1} .

When between a peripheral device and microprocessor system memory transmitted necessary data block **DMA** sets L -level at **HOLD** line and the microprocessor goes out of T_w [6].

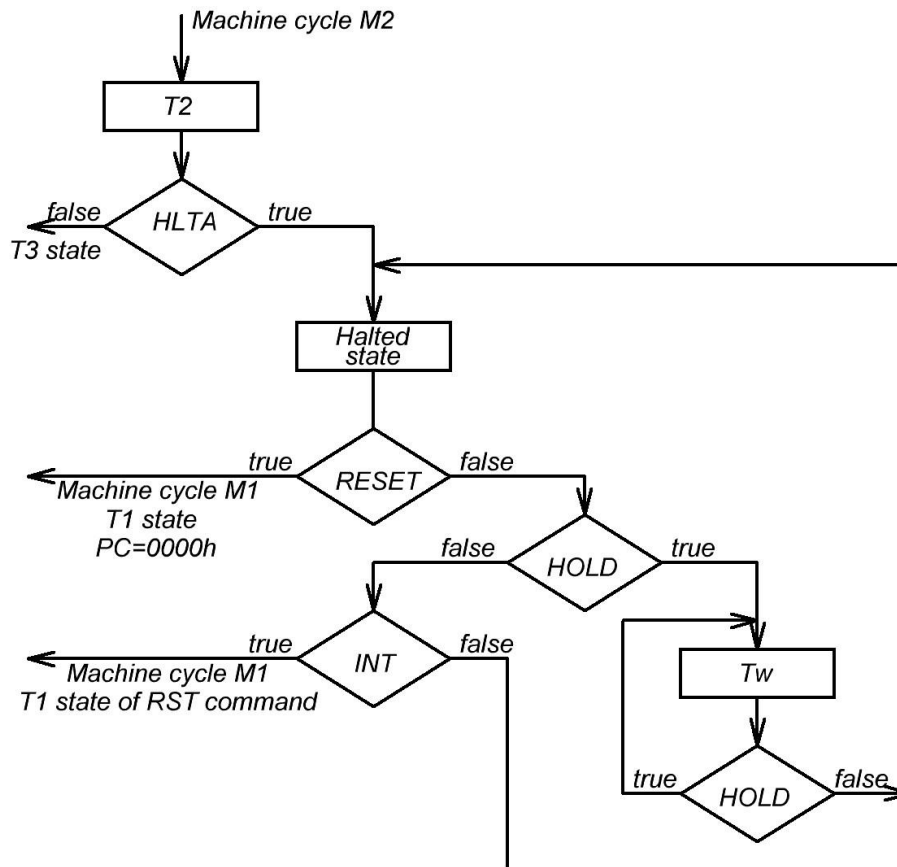
3.17. REACTIONS OF MICROPROCESSOR ON COMMAND **HLT**.

One-byte command **HLT** is as follows. In the machine cycle M_1 , consisting of four cycles $T_1 - T_4$, fetch and decoding of command, and in machine cycle M_2 is the execution of the command. In state T_1 of machine cycle M_2 microprocessor outputs to the address bus the contents of the program counter and to the data bus - a status byte with set acknowledge breakpoint bit **HLTA**. In state T_2 on the rising edge of Y_2 with a maximum delay of 200 ns internal buffers of address and data buses are transferred to a high-impedance state, and on the rising edge of Y_1 in the next state forms the H -level on the **WAIT** line. Execution of the program is stopped and the halted state of the microprocessor can be in any time.

From the halted state microprocessor is put out in the following ways (Fig. 3.15):

1. By setting the H -level to the **RESET** input with a duration at least three periods of synchronization. After that when **RESET** line set in L -level on the rising edge of the signal Y_1 internal reset signal is generated. It loads the program counter to zero and causes the control circuit to form the next state as T_1 state of machine cycle M_1 - fetching of operational code. Therefore, the microprocessor refers to a 0000 cell which typically is the starting address of system initialization;
2. By setting the H -level to the interrupt input **INT**. Microprocessor responds to this signal only if the trigger interrupt enable internal is set ($INTE = 1$). Consequently, before **HLT** command it is necessary to enable interrupts by using the **EI** command. In response to the signal **INT** microprocessor enters a machine cycle M_1 of fetching restart command. If the microprocessor is halted and $INTE = 0$, then putting out the microprocessor from the halted state can be done only by **RESET** signal.

It is important that the halted state microprocessor regularly responds to the signal **HOLD** of **DMA** request with the formation of the **HLDA** signal. The activity of microprocessor in halted state is shown in Fig. 3.15.



3.15. The activity of microprocessor in halted state

3.18. REACTIONS OF MICROPROCESSOR ON *INT* SIGNAL.

In *LSI* of microprocessor has simple interrupt processing schemes with two external interrupt signals:

- to the input interrupt line *INT* we connect the enable signals of slow peripheral devices to exchange data;
- and the output signal *INTE* of Interrupt Enable determines whether the microprocessor to respond to the interrupt requests (*INTE* = 1) or not (*INTE* = 0).

In the interrupt request (*INT* = 1, *INTE* = 1) it is necessary:

- halt the current program;
- temporarily save the contents of the program counter as a return address;
- transfer the control to an interrupt handler of the peripheral device, the request is accepted by;
- restore the interrupted program and resume it.

To temporarily save of the contents of program counter and other internal registers, which will modify the interrupt handler, it is convenient to use the stack.

In microprocessor systems based on *Intel* 8080 vector priority interrupt system is implemented. Asynchronous interrupt signal identified by the H-level on the line *INT*, may appear at any time of the machine cycle. The internal circuit synchronizes the external request and ensures the ending of the current command according to the system timing signals. In the last state of the last machine cycle of all commands (except of interrupt enable command) if the *INT* = 1 and *INTE* = 1 on the rising edge of the signal *Y₂* is set internal interrupts trigger. This leads to the fact that the next state is *T₁* state of machine cycle "*INTERRUPT*". It resembles the machine cycle "*FETCHING OF OPERATIONAL CODE*", as in the status byte bit *M₁* is set. But at the same time acknowledge interrupt bit *INTA* is set too and the reading from memory bit *MEMR* is reset.

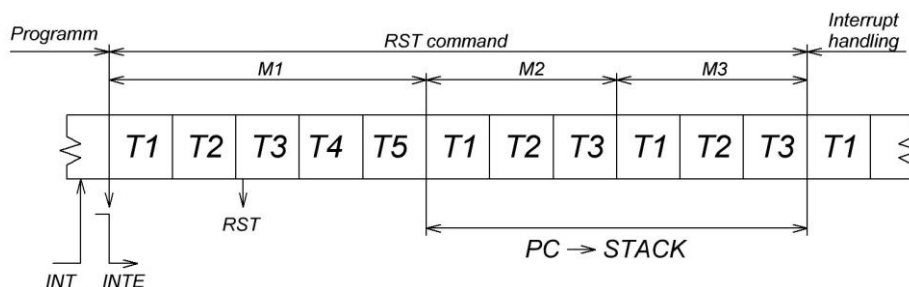


Fig. 3.16. Reactions of microprocessor on signal *INT*

In state *T₁* the microprocessor sets the contents of program counter on the address bus, and the status byte (*M₁* = 1, *INTA* = 1, *MEMR* = 0) on the data bus. At the same state on the rising edge of *Y₂* with a maximum delay of 200 ns L-level output *INTE* signal is generated (Fig. 3.16). Consequently, the microprocessor will ignore further requests until the interrupt enable trigger is not set by the command *EI*.

In *T₂* state, the reading signal *DBIN* is generated, which in an ordinary cycle *M₁* enters the operation code from the program memory in the instruction register. But in a cycle of "*INTERRUPT*" an appeal to the program memory is disabled (*MEMR* = 0), and therefore the operation code should be generated by the interrupt subsystem. Note that in the state *T₂* cycle "*INTERRUPT*" increment *PC* is not made, so it saves the command's address, which would be carried out without interrupt. And, in *T₂* state the internal interrupt trigger is reset.

To download the contents of program counter to the stack interrupt subsystem one should form a call instruction. Standard call instruction *CALL* is a 3-byte and runs for five machine cycles. To speed up the reaction of the microprocessor and simplify the interface with the interrupt subsystem a special one-byte call instruction, called restart (*RESTART*, or *RST*) with operation code 11AAA111 is included in a command system .

Three bits AAA is called the vector and should be generated by a peripheral device which forms the request to the microprocessor. In T_3 state restart command from the data bus is loaded into the instruction register, and states T_4 and T_5 are reserved to decode *RST* command.

RST command initiates execution of two actions:

1. Contents of program counter *PC* is loaded into the stack. For this, three-states cycles M_2 and M_3 "*WRITING INTO STACK*" are introduced;

2. In the program counter code 00000000 00AAA000 is transmitted that "direct" microprocessor to the start address of the interrupt handler. Thus, bits AAA uniquely identify a peripheral device which forms the interrupt request.

After these actions by the first command of interrupt handler the cycle of "*FETCHING* of *OPERATION CODE*" begins, and it makes the necessary steps for data exchange with the peripheral device. The last command of the interrupt handler must be a *RETURN* command, which loads the return address saved in the stack to the program counter.

When analyzing the temporal relation of interrupt, be aware that when the runs interrupt enable command *EI* microprocessor does not respond to the signal *INT*, and after *EI* will be the next instruction. On the line should be maintained *INT* H-level to issue an acknowledge bit interrupt *INTA*.

The following example shows an interrupt sequence:

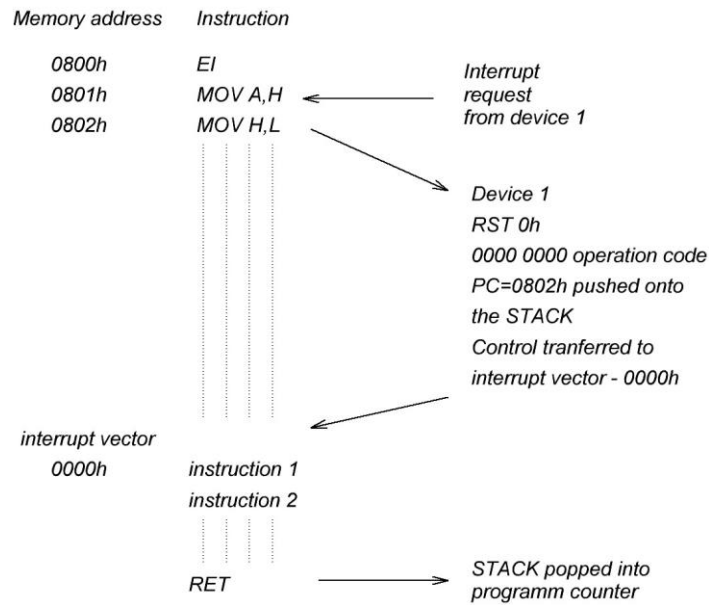


Fig. 3.17. Interrupt sequence of *Intel 8080*

Interrupt subroutines

In general, any registers or condition bits changed by an interrupt subroutine must be restored before returning to the interrupted program.

Like any other subroutine then, any interrupt subroutine should save at least the condition bits and restore them before performing a ***RETURN*** operation. The most convenient way to do this is to save the data in the stack, using ***PUSH*** and ***POP*** operations.

Further, the interrupt enable system is automatically disabled whenever an interrupt is acknowledged. Except in special cases, therefore, an interrupt subroutine should include an ***EI*** instruction somewhere to permit detection and handling of future interrupts. Any time after an ***EI*** is executed, the interrupt subroutine may be also interrupted.

A typical interrupt subroutine, then, could appear as follows:

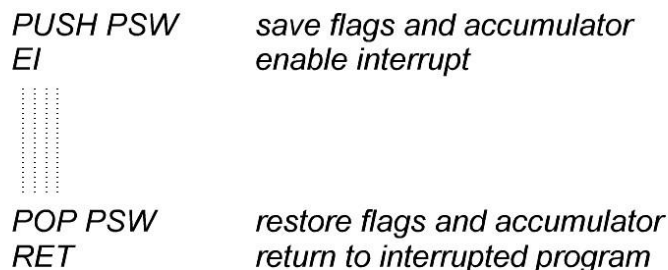


Fig. 3.18. Interrupt subroutine

PART 4. ASSEMBLY LANGUAGE OF *INTEL 8080*

Upon examining the contents of computer memory, a program would appear as a sequence of hexadecimal digits, which are interpreted by the *CPU* as instruction codes, addresses, or data. It is possible to write a program as a sequence of digits (just as they appear in memory), but that is slow and uncomfortable. For example, many instructions reference memory to address either a data byte or another instruction (Fig. 4.1) [7, 8]:

<i>Hexadecimal memory address</i>	<i>Hexadecimal digits</i>
0800h	7Eh
0801h	03h
0802h	C3h
0803h	C4h
0804h	14h
0805h	3Eh
⋮	⋮
0810h	7Eh
0811h	03h
0812h	C3h

Fig. 4.1. Location of assembly program in a memory

Writing programs in assembly language is the first and most significant step towards economical programming; it provides a loadable notation (or instructions, and separates the programmer from a need to know or specify absolute memory addresses.

Assembly language programs are written as a sequence or instructions which are converted to executable hexadecimal code by a special program called an **ASSEMBLER**.

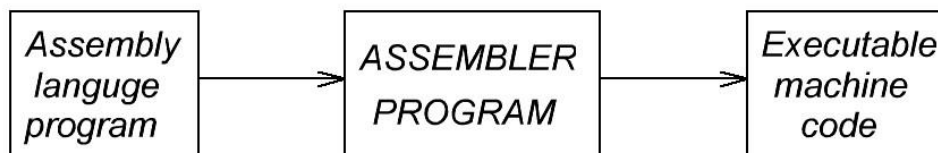


Fig. 4.2. Assembler program converts

As illustrated in Figure 30, the assembly language program generated by a programmer into an equivalent executable machine code, which consists of a sequence of binary codes that can be loaded into memory and executed.

Assembly language instructions and assembly directives may consist of up to four fields, as follows:

Label Name: *Operational code* *Operands* ;*Comment*

The fields may be separated by any number of blanks, but must be separated by at least one delimiter. Each instruction and directive must be entered on a single line terminated by a carriage return and a line feed. No continuation lines are possible, but one may have lines consisting entirely of comments.

Labels are always optional. An instruction label is a symbol name whose value is the location where the instruction is assembled. A label may contain from one to six alphanumeric characters, but the first character must be alphabetic or the special characters '?' or '@'. The label name must be terminated with a colon. A symbol used as a label can be defined only once in the program. Alphanumeric characters include the letters of the alphabet, the question mark character, and the decimal digits 0 through 9. Names are required for the *SET*, *EQU* and *MACRO* directives. Names follow the same coding rules as labels, except that they must be terminated with a blank rather than a colon. The label name field must be empty for the *LOCAL* and *ENDM* directives.

Operational code required field contains the mnemonic operation code for the *Intel 8080* instruction or assembler directive to be performed.

The operand field identifies the data to be operated on by the specified operational code. Some instructions require no operands. Others require one or two operands. As a general rule, when two operands are required (as in data transfer and arithmetic operations), the first operand identifies the destination (or target) of the operation's result, and the second operand specifies the source data.

The optional comment field may contain any information one deem useful for annotating your program. The only coding requirement for this field is that it must be preceded by a semicolon. Because the semicolon is a delimiter, there is need to separate the comment from the previous field with one or more spaces. However, spaces are commonly used to improve the readability of the comment. Although comments are always optional, one should use them liberally since it is easier to debug and maintain a well-documented program.

4.1. DATA AND INSTRUCTIONS FORMATS.

Memory of *Intel 8080* microprocessor is organized by bytes. Each byte has a unique 16-bit binary address corresponding to its position in memory. *Intel 8080* microprocessor can directly address up to 65,536 bytes of memory, which may consist of both read-only memory (*ROM*) elements and random-access memory (*RAM*) elements [7, 8].

Data is stored in 8-bit binary form:

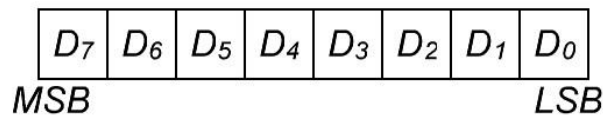


Fig. 4.3. Data word

When a register or data word contains a binary number, it is necessary to establish the order in which the bits of the number are written (Fig. 4.3). In the *Intel* 8080, *BIT* 0 is referred to as the Least Significant Bit (*LSB*), and *BIT* 7 is referred to as the Most Significant Bit (*MSB*).

The *Intel* 8080 microprocessor instructions can be one, two or three bytes in length (Fig. 4.4). Multiple byte instructions must be stored in successive memory locations; the address of the first byte is always used as the address of the instructions. The exact instruction format will depend on a particular operation to be executed.

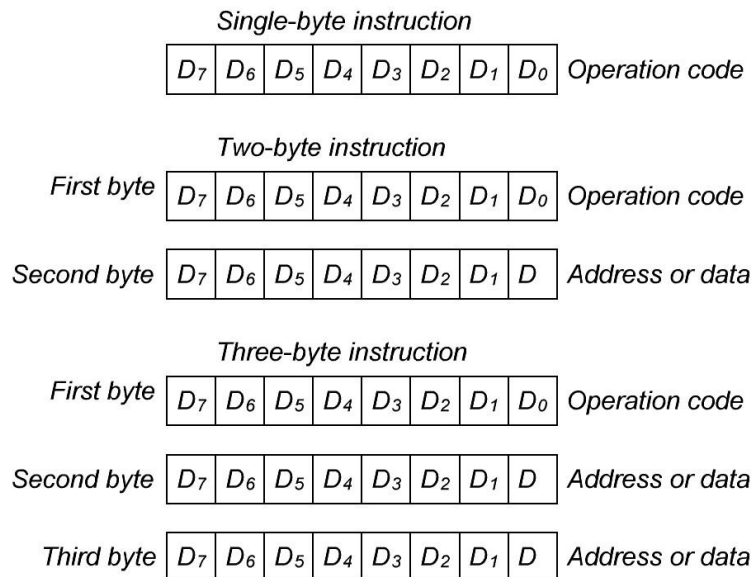


Fig. 4.4– Single, two and three byte instructions

In *Intel* 8080 assembler the following instruction operand symbols are reserved:

Table 4.1. Reserved operand symbols

Symbol	Descriptrion
\$	Location counter reference
A	Accumulator register
B	Register B or register pair B and C

<i>C</i>	Register <i>C</i>
<i>D</i>	Register <i>D or</i> register pair <i>D</i> and <i>E</i>
<i>E</i>	Register <i>E</i>
<i>H</i>	Register <i>H or</i> register pair <i>H</i> and <i>L</i>
<i>L</i>	Register <i>L</i>
<i>SP</i>	Stack pointer register
<i>PSW</i>	Program status word (Contentss of <i>A</i> and status flags)
<i>M</i>	Memory reference code using address in <i>H</i> and <i>L</i>

4.2. MEMORY ADDRESSING

The *Intel* 8080 microprocessor has four different modes for addressing data stored in memory or in registers:

- **Implied addressing.** The addressing mode of certain instructions is implied by the instruction's function. For example, the *STC* (set carry flag) instruction deals only with the carry flag; the *DAA* (decimal adjust accumulator) instruction deals with the accumulator.

- **Direct addressing** — Jump instructions include a 16-bit address as part of the instruction. For example, the instruction *JMP 1000h* causes a jump to the hexadecimal address *1000h* by replacing the current contentss of the program counter with the new value *1000h*. Instructions that include a direct address require three bytes of storage: one for the instruction code and two for the 16-bit address. **Bytes 2** and **3** of the instruction contain the exact memory address of the data. In this case the low-order bits of the address are in **byte 2**, the high-order bits in **byte 3**.

- **Register addressing**— Quite a large set of instructions call for register addressing. With these instructions, you must specify one of the registers *A* through *E*, *H* or *L* as well as the operation code. With these instructions, the accumulator is implied as a second operand. For example, the instruction *CMP E* may be interpreted as the comparison of the contents of the *E* register and the contents of the accumulator. Most of the instructions that use register addressing deal with 8-bitvalues. However, a few of these instructions deal with 16-bitregister pairs. For example, the *PCHL* instruction exchanges the contents of the program counter with the contents of the *H* and *L* registers.

- **Register indirect addressing** — Register indirect instructions reference memory via a register pair. Thus, the instruction *MOV M,C* moves the contents of the *C* register into the memory address stored in the *H* and *L* register pair. The instruction *LDAX B* loads the accumulator with the byte of data specified by the address in the *B* and *C* register pair.

- **Immediate addressing** — Instructions that use immediate addressing have data assembled as a part of the instruction itself. For example, the instruction *CPI C* maybe interpreted as the comparison of the contents of the accumulator and the contents of the register *C*. When assembled, this instruction has the hexadecimal value *FE 43*. Hexadecimal *43* is the internal representation for the letter *C*. When this instruction is executed, the processor fetches the first instruction byte and determines that it must fetch one more byte. The processor fetches the next byte into one of its internal registers and then performs the comparison operation.

Notice that the names of the immediate instructions indicate that they use immediate data. Thus, the name of an add instruction is *ADD*; the name of an add immediate instruction is *ADI*.

All but two of the immediate instructions use the accumulator as an implied operand, as in the *CPI* instruction shown previously. The *MVI* (move immediate) instruction can move its immediate data to any of the working registers, including the accumulator, or to memory. Thus, the instruction *MVI D,hFFh* moves the hexadecimal value *FFh* to the *D* register.

The *LXI* instruction (load register pair immediate) is even more unusual in that its immediate data is a 16-bit value. This instruction is commonly used to load addresses into a register pair. As mentioned previously, your program must initialize the stack pointer; *LXI* is the instruction most commonly used for this purpose. For example, the instruction *LXI SP,30FFh* loads the stack pointer with the hexadecimal value *30FFh*.

The *RST* instruction is a special one-byte call instruction (usually used during interrupt sequences). *RST* includes a three-bit field; program control is transferred to the instruction whose address is eight times the contents of this three-bit field.

- **Combed addressing modes** – Some instructions use a combination of addressing modes. A *CALL* instruction, for example, combines direct addressing and register Indirect addressing. The direct addressing a *CALL* instruction specifies the address of the desired subroutine; the register indirect address is the stack pointer. The *CALL* instruction pushes the current contents of the program counter into the memory location specified by the stack pointer.

- **Timing effect of addressing modes.** Addressing modes affect both the amount of time required for executing an instruction and the amount of memory required for its storage. For example, instructions that use immediate or register addressing are executed very quickly since they deal directly with the processor hardware or with data already present in hardware registers. More important, however, is that the entire instruction can be fetched with a

single memory access. The number of memory accesses required is the single greatest factor in determining execution timing. More memory accesses require more execution time. A *CALL* instruction, for example, requires five memory accesses: three to access the entire instruction, and two more to push the contents of the program counter onto the stack.

The processor can access memory once during each processor cycle. Each cycle comprises a variable number of states. The length of a state depends on the clock frequency specified for your system, and may range from 48 nanoseconds to 2 microseconds. Thus, the timing of a four state instruction may range from 1.920 microseconds through 8 microseconds.

4.3. INTEL 8080 INSTRUCTIONS: DESCRIPTION AND APPLICATION

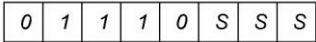
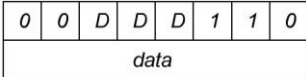
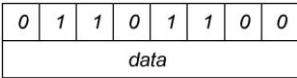
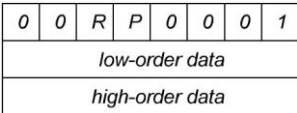
The 8080 instruction set includes five different types of instructions.

1. **Data Transfer Group** —move data between registers or between memory and registers [7, 8].

This group of instructions transfers data to and from registers and memory. The Instruction in this group does not affect the conditions flags. In Table 4.2 different instructions of this group and their properties are shown.

Table 4.2. Data transfer instructions

Instruction	Action	Instruction format	Cycles (states)	Addressing	Flags								
<i>MOV r1, r2</i>	The contents of register <i>r2</i> is moved to register <i>r1</i>	<div style="text-align: center;"> <table border="1" style="display: inline-table; border-collapse: collapse;"> <tr> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">D</td> <td style="padding: 2px 5px;">D</td> <td style="padding: 2px 5px;">D</td> <td style="padding: 2px 5px;">S</td> <td style="padding: 2px 5px;">S</td> <td style="padding: 2px 5px;">S</td> </tr> </table> </div> <p><i>r1, r2</i> – One of the registers <i>A, B, C, D, E, H, L</i> <i>DDD</i> – destination register <i>SSS</i> – source register</p>	0	1	D	D	D	S	S	S	1 (5)	register	none
0	1	D	D	D	S	S	S						
<i>MOV r, M</i>	The contents of the memory location, whose address is in registers <i>H</i> and <i>L</i> , is moved to register <i>r</i>	<div style="text-align: center;"> <table border="1" style="display: inline-table; border-collapse: collapse;"> <tr> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">D</td> <td style="padding: 2px 5px;">D</td> <td style="padding: 2px 5px;">D</td> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">0</td> </tr> </table> </div> <p><i>r</i> – One of the registers <i>A, B, C, D, E, H, L</i></p>	0	1	D	D	D	1	1	0	2 (7)	register indirect	none
0	1	D	D	D	1	1	0						

MOVM, r	The contents of register r is moved to the memory location whose address is in registers H and L .		2 (7)	register indirect	none
MVI r, data	The contents of byte 2 of the instruction is moved to register r .	 <p><i>data</i> - 8-bit data quantity</p>	2 (7)	immediate	none
MVI M, data	The contents of byte 2 of the instruction is moved to the memory location whose address is in registers H and L .	 <p><i>data</i></p>	3 (10)	Immediate/register indirect	none
LXI rp, data16	Byte 3 of the instruction is moved into the high-order register (rh) of the register pair rp . Byte 2 of the instruction is moved into the low-order register (rl) of the register pair rp .	 <p><i>low-order data</i></p> <p><i>high-order data</i></p> <p>byte 2 - The second byte of the instruction byte 3 - The third byte of the instruction rp - One of the register pairs: B represents the B,C pair with B as the high-order register and C as the low-order register; D represents the D,E pair with D as the high-order register and E as the low-order register; H represents the H,L pair with H as the high-order register and L as the low-order register;</p>	3 (10)	Immediate	none

		<p>SP represents the 16-bit stack pointer register.</p> <p>rh – the first (high-order) register of a designated register pair.</p> <p>rl – the second (low-order) register of a designated register pair.</p>																											
LDA addr	The contents of the memory location, whose address is specified in byte 2 and byte 3 of the instruction, is moved to register A .	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">0</td><td style="text-align: center;">0</td><td style="text-align: center;">1</td><td style="text-align: center;">1</td><td style="text-align: center;">1</td><td style="text-align: center;">0</td><td style="text-align: center;">1</td><td style="text-align: center;">0</td> </tr> <tr> <td colspan="8" style="text-align: center;"><i>low-order address</i></td> </tr> <tr> <td colspan="8" style="text-align: center;"><i>high-order address</i></td> </tr> </table> <p>addr – 16-bit address quantity</p>	0	0	1	1	1	0	1	0	<i>low-order address</i>								<i>high-order address</i>								4 (13)	direct	none
0	0	1	1	1	0	1	0																						
<i>low-order address</i>																													
<i>high-order address</i>																													
STA addr	The contents of the accumulator is moved to the memory location whose address is specified in byte 2 and byte 3 of the instruction.	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">0</td><td style="text-align: center;">0</td><td style="text-align: center;">1</td><td style="text-align: center;">1</td><td style="text-align: center;">0</td><td style="text-align: center;">0</td><td style="text-align: center;">1</td><td style="text-align: center;">0</td> </tr> <tr> <td colspan="8" style="text-align: center;"><i>low-order address</i></td> </tr> <tr> <td colspan="8" style="text-align: center;"><i>high-order address</i></td> </tr> </table>	0	0	1	1	0	0	1	0	<i>low-order address</i>								<i>high-order address</i>								4 (13)	direct	none
0	0	1	1	0	0	1	0																						
<i>low-order address</i>																													
<i>high-order address</i>																													
LHLD addr	The contents of the memory location, whose address is specified in byte 2 and byte 3 of the instruction, is moved to register L . The contents of the memory lo-	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">0</td><td style="text-align: center;">0</td><td style="text-align: center;">1</td><td style="text-align: center;">0</td><td style="text-align: center;">1</td><td style="text-align: center;">0</td><td style="text-align: center;">1</td><td style="text-align: center;">0</td> </tr> <tr> <td colspan="8" style="text-align: center;"><i>low-order address</i></td> </tr> <tr> <td colspan="8" style="text-align: center;"><i>high-order address</i></td> </tr> </table>	0	0	1	0	1	0	1	0	<i>low-order address</i>								<i>high-order address</i>								5 (16)	direct	none
0	0	1	0	1	0	1	0																						
<i>low-order address</i>																													
<i>high-order address</i>																													

	<p>cation at the succeeding address is moved to register <i>H</i>.</p>																												
<i>SHLD addr</i>	<p>The contents of register <i>L</i> is moved to the memory location whose address is specified in byte 2 and byte 3. The contents of register <i>H</i> is moved to the succeeding memory location.</p>	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">0</td><td style="text-align: center;">0</td><td style="text-align: center;">1</td><td style="text-align: center;">0</td><td style="text-align: center;">0</td><td style="text-align: center;">0</td><td style="text-align: center;">1</td><td style="text-align: center;">0</td> </tr> <tr> <td colspan="8" style="text-align: center;"><i>low-order address</i></td> </tr> <tr> <td colspan="8" style="text-align: center;"><i>high-order address</i></td> </tr> </table>	0	0	1	0	0	0	1	0	<i>low-order address</i>								<i>high-order address</i>								5 (16)	direct	none
0	0	1	0	0	0	1	0																						
<i>low-order address</i>																													
<i>high-order address</i>																													
<i>LDAX rp</i>	<p>The contents of the memory location, whose address is in the <i>rp</i>, is moved to register <i>A</i>. Only register pairs <i>rp=B</i> or <i>rp=D</i> may be specified.</p>	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">0</td><td style="text-align: center;">0</td><td style="text-align: center;"><i>R</i></td><td style="text-align: center;"><i>P</i></td><td style="text-align: center;">1</td><td style="text-align: center;">0</td><td style="text-align: center;">1</td><td style="text-align: center;">0</td> </tr> </table>	0	0	<i>R</i>	<i>P</i>	1	0	1	0	2 (7)	register indirect	none																
0	0	<i>R</i>	<i>P</i>	1	0	1	0																						
<i>STAX rp</i>	<p>The contents of <i>A</i> is moved to the memory cell location whose address is in the <i>rp</i>. Only <i>rp=B</i> or <i>rp=D</i> may be specified.</p>	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">0</td><td style="text-align: center;">0</td><td style="text-align: center;"><i>R</i></td><td style="text-align: center;"><i>P</i></td><td style="text-align: center;">0</td><td style="text-align: center;">0</td><td style="text-align: center;">1</td><td style="text-align: center;">0</td> </tr> </table>	0	0	<i>R</i>	<i>P</i>	0	0	1	0	2 (7)	register indirect	none																
0	0	<i>R</i>	<i>P</i>	0	0	1	0																						
<i>XCHG</i>	<p>The contents of registers</p>	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">1</td><td style="text-align: center;">1</td><td style="text-align: center;">1</td><td style="text-align: center;">0</td><td style="text-align: center;">1</td><td style="text-align: center;">0</td><td style="text-align: center;">1</td><td style="text-align: center;">1</td> </tr> </table>	1	1	1	0	1	0	1	1	1 (4)	register	none																
1	1	1	0	1	0	1	1																						

	<i>H</i> and <i>L</i> are exchanged with the contents of <i>D</i> and <i>E</i> .				
--	--	--	--	--	--

Example 4.1: Using *MVI r,data* commands.

Mnemonic	Operands	Machine code	Comment
MVI	A,00	3E 00	loading of <i>A</i> register: <i>A</i> ← 00h
MVI	B,01	06 01	loading of <i>B</i> register: <i>B</i> ← 01h
MVI	C,02	0E 02	loading of <i>C</i> register: <i>C</i> ← 02h
MVI	D,03	16 03	loading of <i>D</i> register: <i>D</i> ← 03h
MVI	E,04	1E 04	loading of <i>E</i> register: <i>E</i> ← 04h
MVI	H,05	26 05	loading of <i>H</i> register: <i>H</i> ← 05h
MVI	L,06	2E 06	loading of <i>L</i> register: <i>L</i> ← 06h

The output of the program execution:

A = **00h**
B = **01h**
C = **02h**
D = **03h**
E = **04h**
H = **05h**
L = **06h**

Example 4.2: Using *MOV r1, r2* commands.

Mnemonic	Operands	Machine code	Comment
MVI	A,FFh	3E FF	loading of <i>A</i> register: <i>A</i> ← FFh
MOV	B,A	47	the contents of register <i>A</i> is moved to register <i>B</i>
MOV	C,B	48	the contents of register <i>B</i> is moved to register <i>C</i>
MOV	D,C	51	the contents of register <i>C</i> is moved to register <i>D</i>
MOV	E,D	5A	the contents of register <i>D</i> is moved to register <i>E</i>
MOV	H,E	63	the contents of register <i>E</i> is moved to register <i>H</i>
MOV	L,H	6C	the contents of register <i>H</i> is moved

			to register <i>L</i>
--	--	--	----------------------

The output of the program execution:

A = FFh
B = FFh
C = FFh
D = FFh
E = FFh
H = FFh
L = FFh

Example 4.3: Using *LXI rp,data16* commands.

Mnemonic	Operands	Machine code	Comment
LXI	B,3132h	01 32 31	loading of <i>BC</i> register pair: <i>BC</i> ← <i>3132h</i> . Byte <i>31h</i> of the instruction is moved into the <i>B</i> register. Byte <i>32h</i> of the instruction is moved into the <i>C</i> .
LXI	D,3334h	11 34 33	loading of <i>DE</i> register pair: <i>DE</i> ← <i>3334h</i> . Byte <i>33h</i> of the instruction is moved into the <i>D</i> register. Byte <i>34h</i> of the instruction is moved into the <i>E</i> .
LXI	H,3536h	21 36 35	loading of <i>HL</i> register pair: <i>HL</i> ← <i>3536h</i> . Byte <i>35h</i> of the instruction is moved into the <i>H</i> register. Byte <i>36h</i> of the instruction is moved into the <i>L</i> .
LXI	SP,0B01h	31 01 0B	loading of stack pointer: <i>SP</i> ← <i>0B01h</i> .

The output of the program execution:

B = 31h
C = 32h
D = 33h
E = 34h
H = 35h
L = 36h
SP=0B01h

Example 4.4: Using *STA addr* and *SHLD addr* commands.

Mnemonic	Operands	Machine code	Comment
MVI	A,FFh	3E FF	loading of <i>A</i> register: $A \leftarrow FFh$
STA	0110h	32 10 01	The contents of the accumulator is moved to the memory location whose address is specified in the instruction as <i>0110h</i> .
LXI	H,3536h	21 36 35	loading of <i>HL</i> register pair: $HL \leftarrow 3536h$. Byte <i>35h</i> of the instruction is moved into the <i>H</i> register. Byte <i>36h</i> of the instruction is moved into the <i>L</i> .
SHLD	0150h	22 50 01	The contents of register <i>L</i> is moved to the memory location whose address specified of the instruction as <i>0150h</i> . The contents of register <i>H</i> is moved to the address <i>0151h</i> .

The output of the program execution:

$A = FFh$
 $H = 35h$
 $L = 36h$
 $(0110h) = FFh$
 $(0150h) = 36h$
 $(0151h) = 35h$

Example 4.5: Using *LDA addr* and *LHLD addr* commands.

Mnemonic	Operands	Machine code	Comment
LDA	0190h	3A 90 01	The contents of the memory location, whose address is specified in the instruction as <i>0190h</i> , is moved to register <i>A</i> .
LHLD	0190h	2A 90 01	The contents of the memory location, whose address is specified in the instruction as <i>0190h</i> , is moved to register <i>L</i> . The contents of the memory location with address <i>0191h</i> is moved to register <i>H</i> .

The output of the program execution $A = (0190h)$

$H = (0191h)$

$L = (0190h)$

Example 4.6: Using *MOV r,M* and *MOV M,r*, *MVI M, data* commands.

<i>MOV M,r</i> commands			
Mnemonic	Operands	Machine code	Comment
MVI	A,0AAh	3E AA	loading of <i>A</i> register: $A \leftarrow AAh$
MVI	B,0BBh	06 BB	loading of <i>B</i> register: $B \leftarrow BBh$
MVI	C,0CCh	0E CC	loading of <i>C</i> register: $C \leftarrow CCh$
MVI	D,0DDh	16 DD	loading of <i>D</i> register: $D \leftarrow DDh$
MVI	E,0EEh	1E EE	loading of <i>E</i> register: $E \leftarrow EEh$
LXI	H,0100h	21 00 01	loading of <i>HL</i> register pair: $HL \leftarrow 0100h.$
MOV	M,A	77	the contents of register <i>A</i> is moved to the memory location whose address is in register pair <i>HL</i> . $(HL) \leftarrow A$
LXI	H,0101h	21 01 01	$HL \leftarrow 0101h.$
MOV	M,C	77	$(HL) \leftarrow C$
LXI	H,0102h	21 02 01	$HL \leftarrow 0102h.$
MOV	M,B	71	$(HL) \leftarrow B$
LXI	H,0103h	21 03 01	$HL \leftarrow 0103h.$
MOV	M,E	70	$(HL) \leftarrow E$
LXI	H,0104h	21 04 01	$HL \leftarrow 0104h.$
MOV	M,D	72	$(HL) \leftarrow D$
LXI	H,0105h	21 05 01	$HL \leftarrow 0105h.$
MOV	M,H	74	$(HL) \leftarrow H$
LXI	H,0106h	21 06 01	$HL \leftarrow 0106h.$
MOV	M,L	75	$(HL) \leftarrow L$
<i>MOV r,M</i> commands			
LXI	H,0100h	21 00 01	$HL \leftarrow 0100h.$
MOV	E,M	5E	$E \leftarrow (HL)$
LXI	H,0105h	21 05 01	$HL \leftarrow 0105h.$
MOV	D,M	56	$D \leftarrow (HL)$
<i>MVI M, data</i> commands			
LXI	H,0107h	21 07 01	$HL \leftarrow 0107h.$
MVI	M,0Fh	6C	$(HL) \leftarrow 0Fh$

The output of the program execution:

0100h = AAh

0101h = CCh

0102h = BBh

0103h = EEh

0104h = DDh

0105h = 01h

0106h = 06h

E=AAh

D=01h

0107h=0Fh

Example 4.7: Using *STAX rp* and *LDAX rp* commands.

<i>STAX rp</i> commands			
Mnemonic	Operands	Machine code	Comment
LXI	B,0100h	01 00 01	loading of BC register pair: BC←0100h.
MVI	A,0Fh	3E 0F	loading of A register: A←0Fh
STAX	B	02	The contents of A is moved to the memory cell location whose address is in the BC .
LXI	D,0110h	11 10 01	loading of DE register pair: DE←0110h.
MVI	A,F0h	3E F0	loading of A register: A←F0h
STAX	D	12	The contents of A is moved to the memory cell location whose address is in the DE .
<i>LDAX rp</i> commands			
LXI	D,0100h	11 00 01	loading of DE register pair: DE←0100h.
LDAX	D	1A	The contents of the memory location, whose address is in the DE , is moved to register A .

The output of the program execution:

0100h = 0Fh

0110h = F0h

A=0Fh

Example 4.8: Using *XCHG* command.

<i>STAX rp</i> commands			
Mnemonic	Operands	Machine code	Comment
LXI	H,AABBh	21 00 01	loading of <i>HL</i> register pair: <i>BC</i> ← <i>AABBh</i> .
LXI	D,CCDDh	11 00 01	loading of <i>DE</i> register pair: <i>BC</i> ← <i>CCDDh</i> .
SCHG		EB	The contents of register pair <i>HL</i> are exchanged with the contents of <i>DE</i> .

The output of the program execution:

H = CCh

L = DDh

D = AAh

E = BBh

2. Arithmetic Group – add, subtract, increment or decrement data in registers or in memory [7, 8].

Table 4.3. Arithmetic group instructions

Instruction	Action	Instruction format	Cycles (states)	Addressing	Flags								
<i>ADD r</i>	The contents of register <i>r</i> is added to the accumulator. The result is placed in the accumulator.	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>S</td><td>S</td><td>S</td> </tr> </table>	1	0	0	0	0	S	S	S	1 (4)	register	S,Z,P, CY, AC
1	0	0	0	0	S	S	S						
<i>ADD M</i>	The contents of the memory location whose address is contained in the <i>H</i> and <i>L</i> registers is added to the contents of the accumulator. The	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td> </tr> </table>	1	0	0	0	0	1	1	0	2 (7)	register indirect	S,Z,P, CY, AC
1	0	0	0	0	1	1	0						

	result is placed in the accumulator.																				
ADI data	The contents of the second byte of the instruction is added to the contents of the accumulator. The result is placed in the accumulator.	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td> </tr> <tr> <td colspan="8" style="text-align: center;">data</td> </tr> </table>	1	1	0	0	0	1	1	0	data								2 (7)	immediate	S,Z,P ,CY, AC
1	1	0	0	0	1	1	0														
data																					
ADC r	The contents of register <i>r</i> and the contents of the carry bit are added to the contents of the accumulator. The result is placed in the accumulator.	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>S</td><td>S</td><td>S</td> </tr> </table>	1	0	0	0	1	S	S	S	1 (4)	register	S,Z,P ,CY, AC								
1	0	0	0	1	S	S	S														
ADC M	The contents of the memory location whose address is contained in the <i>H</i> and <i>L</i> registers and the contents of the <i>CY</i> flag are added to the accumulator. The result is placed in the accumulator.	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td> </tr> </table>	1	0	0	0	1	1	1	0	2 (7)		S,Z,P ,CY, AC								
1	0	0	0	1	1	1	0														
ACI data	The contents of the second byte of the instruction	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td> </tr> <tr> <td colspan="8" style="text-align: center;">data</td> </tr> </table>	1	1	0	0	1	1	1	0	data								2 (7)	immediate	S,Z,P ,CY, AC
1	1	0	0	1	1	1	0														
data																					

	and the contents of the CY flag are added to the contents of the accumulator. The result is placed in the accumulator.																				
<i>SUB r</i>	The contents of register r is subtracted from the contents of the accumulator. The result is placed in the accumulator.	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>S</td><td>S</td><td>S</td> </tr> </table>	1	0	0	1	0	S	S	S	1 (4)	register	S,Z,P ,CY, AC								
1	0	0	1	0	S	S	S														
<i>SUB M</i>	The contents of the memory location whose address is contained in the H and L registers is subtracted from the contents of the accumulator. The result is placed in the accumulator.	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td> </tr> </table>	1	0	0	1	0	1	1	0	2 (7)	register indirect	S,Z,P ,CY, AC								
1	0	0	1	0	1	1	0														
<i>SUI data</i>	The contents of the second byte of the instruction is subtracted from the contents of the accumulator. The result is	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td> </tr> <tr> <td colspan="8" style="text-align: center;"><i>data</i></td> </tr> </table>	1	1	0	1	0	1	1	0	<i>data</i>								2 (7)	immediate	S,Z,P ,CY, AC
1	1	0	1	0	1	1	0														
<i>data</i>																					

	placed in the accumulator.																				
<i>SBB r</i>	The contents of register <i>r</i> and the contents of the <i>CY</i> flag are both subtracted from the accumulator. The result is placed in the accumulator.	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>S</td><td>S</td><td>S</td> </tr> </table>	1	0	0	1	1	S	S	S	1 (4)	register	S,Z,P ,CY, AC								
1	0	0	1	1	S	S	S														
<i>SBB M</i>	The contents of the memory location whose address is contained in the <i>H</i> and <i>L</i> registers and the contents of the <i>CY</i> flag are both subtracted from the accumulator. The result is placed in the accumulator.	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td> </tr> </table>	1	0	0	1	1	1	1	0	2 (7)	register indirect	S,Z,P ,CY, AC								
1	0	0	1	1	1	1	0														
<i>SBI data</i>	The contents of the second byte of the instruction and the contents of the <i>CY</i> flag are both subtracted from the accumulator. The result is placed in the accumulator.	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td> </tr> <tr> <td colspan="8" style="text-align: center;"><i>data</i></td> </tr> </table>	1	1	0	1	1	1	1	0	<i>data</i>								2 (7)	immediate	S,Z,P ,CY, AC
1	1	0	1	1	1	1	0														
<i>data</i>																					

<i>INR r</i>	The contents of register <i>r</i> is incremented by one.	<table border="1"><tr><td>0</td><td>0</td><td>D</td><td>D</td><td>D</td><td>1</td><td>0</td><td>0</td></tr></table>	0	0	D	D	D	1	0	0	1 (5)	register	S,Z,P,AC
0	0	D	D	D	1	0	0						
<i>INR M</i>	The contents of the memory location whose address is contained in the <i>H</i> and <i>L</i> registers is incremented by one.	<table border="1"><tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td></tr></table>	0	0	1	1	0	1	0	0	3 (10)	register indirect	S,Z,P,AC
0	0	1	1	0	1	0	0						
<i>DCR r</i>	The contents of register <i>r</i> is decremented by one.	<table border="1"><tr><td>0</td><td>0</td><td>D</td><td>D</td><td>D</td><td>1</td><td>0</td><td>1</td></tr></table>	0	0	D	D	D	1	0	1	1 (5)	register	S,Z,P,AC
0	0	D	D	D	1	0	1						
<i>DCR M</i>	The contents of the memory location whose address is contained in the <i>H</i> and <i>L</i> registers is decremented by one.	<table border="1"><tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr></table>	0	0	1	1	0	1	0	1	3 (10)	register indirect	S,Z,P,AC
0	0	1	1	0	1	0	1						
<i>INX rp</i>	The contents of the register pair <i>rp</i> is incremented by one.	<table border="1"><tr><td>0</td><td>0</td><td>R</td><td>P</td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table>	0	0	R	P	1	0	1	1	1 (5)	Register	none
0	0	R	P	1	0	1	1						
<i>DCX rp</i>	The contents of the register pair <i>rp</i> is decremented by one.	<table border="1"><tr><td>0</td><td>0</td><td>R</td><td>P</td><td>1</td><td>0</td><td>1</td><td>0</td></tr></table>	0	0	R	P	1	0	1	0	1 (5)	Register	none
0	0	R	P	1	0	1	0						
<i>DAD rp</i>	The contents of the register pair <i>rp</i> is added to the contents of the register	<table border="1"><tr><td>0</td><td>0</td><td>R</td><td>P</td><td>1</td><td>0</td><td>0</td><td>1</td></tr></table>	0	0	R	P	1	0	0	1	3 (10)	Register	CY
0	0	R	P	1	0	0	1						

	pair <i>H</i> and <i>L</i> . The result is placed in the register pair <i>H</i> and <i>L</i> .												
<i>DAA</i>	<p>The eight-bit number in the accumulator is adjusted to form two four-bit Binary-Coded-Decimal digits by the following process:</p> <p>1) If the value of the least significant 4 bits of the accumulator is greater than 9 or if the <i>AC</i> flag is set, 6 is added to the accumulator.</p> <p>2) If the value of the most significant 4 bits of the accumulator is now greater than 9, or if the <i>CY</i> flag is set, 6 is added to the most significant 4 bits of the accumulator.</p>	<table border="1" style="margin: auto;"> <tr> <td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td> </tr> </table>	0	0	1	0	0	1	1	1	1 (4)	Implied	<i>S</i> , <i>Z</i> , <i>P</i> , <i>CY</i> , <i>AC</i>
0	0	1	0	0	1	1	1						

Example 4.9: Using *ADD r*, *ADD M* and *ADI data* commands.

Mnemonic	Operands	Machine code	Comment
MVI	A,00	3E 00	loading of <i>A</i> register: $A \leftarrow 00h$
MVI	B,01	06 01	loading of <i>B</i> register: $B \leftarrow 01h$
LXI	H, 0100h	21 00 01	loading of <i>HL</i> register pair: $HL \leftarrow 0100h$
MOV	M,A	77	$(HL) \leftarrow A$
ADD	B	80	$A \leftarrow A+B$
ADD	M	86	$A \leftarrow A+(M)$
ADI	05h	C6 05	$A \leftarrow A+5$

The output of the program execution:

$$\begin{aligned}
 &B=01h \\
 &HL=0100h \\
 &(0100h)=00h \\
 &A=B+(M)+05h=06h
 \end{aligned}$$

Example 4.10: Using *ADC r*, *ADC M* and *ACI data* commands.

Mnemonic	Operands	Machine code	Comment
MVI	A,0FFh	3E FF	loading of <i>A</i> register: $A \leftarrow FFh$
ADI	02h	C6 02	$A \leftarrow A+1=01h$ and carry flag will be <i>1</i>
MVI	B,0FEh	06 FE	loading of <i>B</i> register: $B \leftarrow FEh$
LXI	H, 0100h	21 00 01	loading of <i>HL</i> register pair: $HL \leftarrow 0100h$
MOV	M,A	77	$(HL) \leftarrow A$
ADC	B	80	$A \leftarrow A+B+Carry=0$ and carry flag – <i>1</i>
ADC	M	86	$A \leftarrow A+(M)+Carry=2$ and carry flag – <i>0</i>
ACI	05h	C6 05	$A \leftarrow A+5+Carry=7$ and carry flag – <i>0</i>

The output of the program execution:

$$\begin{aligned}
 &B=FEh \\
 &HL=0100h \\
 &(0100h)=01h \\
 &A=B+(M)+05h+Carry=07h
 \end{aligned}$$

Example 4.11: Using *SUB r*, *SUB M* and *SUI data* commands.

Mnemonic	Operands	Machine code	Comment
MVI	A,0FFh	3E FF	loading of <i>A</i> register: $A \leftarrow FFh$
MVI	B,05h	06 05	loading of <i>B</i> register: $B \leftarrow 05h$
LXI	H, 0100h	21 00 01	loading of <i>HL</i> register pair: $HL \leftarrow 0100h$
MOV	M,B	71	$(HL) \leftarrow B$
SUB	B	90	$A \leftarrow A - B$
SUB	M	96	$A \leftarrow A - (M)$
SUI	0Fh	DE 0F	$A \leftarrow A - 15$

The output of the program execution:

$B = 05h$
 $HL = 0100h$
 $(0100h) = 05h$
 $A = B - (M) - 0Fh = E6h$

Example 4.12: Using *SBB r*, *SBB M* and *SBI data* commands.

Mnemonic	Operands	Machine code	Comment
MVI	A,01h	3E 01	loading of <i>A</i> register: $A \leftarrow 01h$
MVI	B,05h	06 05	loading of <i>B</i> register: $B \leftarrow 05h$
LXI	H, 0100h	21 00 01	loading of <i>HL</i> register pair: $HL \leftarrow 0100h$
MOV	M,B	71	$(HL) \leftarrow B$
SBB	B	98	$A \leftarrow A - B - Carry = FCh$ and carry- <i>1</i>
SBB	M	9E	$A \leftarrow A - (M) - Carry = F7h$ and carry- <i>0</i>
SBI	0FFh	DE FF	$A \leftarrow A - 15 - Carry = F8h$ and carry- <i>1</i>

The output of the program execution:

$B = 05h$
 $HL = 0100h$
 $(0100h) = 05h$
 $A = B - (M) - FFh = F8h$

Example 4.13: Using *INC r*, *INC M* and *INX rp* commands.

Mnemonic	Operands	Machine code	Comment
MVI	A,01h	3E 01	loading of <i>A</i> register: $A \leftarrow 01h$
LXI	H, 0100h	21 00 01	loading of <i>HL</i> register pair: $HL \leftarrow 0100h$

MOV	M,A	77	$(HL) \leftarrow A$
INC	A	3C	$A \leftarrow A+1$
INC	M	34	$(M) \leftarrow (M)+1$
INX	H	23	$HL \leftarrow HL+1$

The output of the program execution:

$HL=0101h$
 $(0100h)=02h$
 $A=02h$

Example 4.14: Using *DCR r*, *DCR M* and *DCX rp* commands.

<i>Mnemonic</i>	<i>Operands</i>	<i>Machine code</i>	<i>Comment</i>
MVI	A,04h	3E 04	loading of A register: $A \leftarrow 04h$
LXI	H, 0101h	21 01 01	loading of <i>HL</i> register pair: $HL \leftarrow 0101h$
MOV	M,A	77	$(HL) \leftarrow A$
DCR	A	3D	$A \leftarrow A-1$
DCR	M	35	$(M) \leftarrow (M)-1$
DCX	H	2B	$HL \leftarrow HL-1$

The output of the program execution:

$HL=0100h$
 $(0101h)=03h$
 $A=03h$

Example 4.15: Using *DAD rp* commands.

Mnemonic	Operands	Machine code	Comment
LXI	H, 0101h	21 01 01	loading of <i>HL</i> register pair: $HL \leftarrow 0101h$
DAD	H	29	$HL \leftarrow HL+HL$

The output of the program execution:

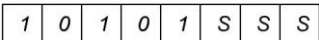

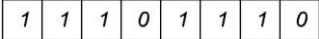

$HL=0202h.$

- **Logical Group** — *AND*, *OR*, *EXCLUSIVE-OR*, compare, shift, rotate or complement data in registers or in memory [7, 8];

Table 4.4. Logical group instructions

Instruction	Action	Instruction format	Cycles	Ad-	Flag
-------------	--------	--------------------	--------	-----	------

			(states)	dress- ing	s																
ANA <i>r</i>	The contents of register <i>r</i> is logically added with the contents of the accumulator. The result is placed in the accumulator. The CY flag is cleared.	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>S</td><td>S</td><td>S</td> </tr> </table>	1	0	1	0	0	S	S	S	1 (4)	regis- ter	S,Z,P ,CY, AC								
1	0	1	0	0	S	S	S														
ANA <i>M</i>	The contents of the memory location whose address is contained in the H and L registers is logically added with the contents of the accumulator. The result is placed in the accumulator. The CY flag is cleared.	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td> </tr> </table>	1	0	1	0	0	1	1	0	2 (7)	regis- ter in- direct	S,Z,P ,CY, AC								
1	0	1	0	0	1	1	0														
ANI <i>data</i>	The contents of the second byte of the instruction is logically added with the contents of the accumulator. The result is placed in the accumulator. The CY and	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td> </tr> <tr> <td colspan="8" style="text-align: center;"><i>data</i></td> </tr> </table>	1	1	1	0	0	1	1	0	<i>data</i>								2 (7)	imme- diate	S,Z,P ,CY, AC
1	1	1	0	0	1	1	0														
<i>data</i>																					

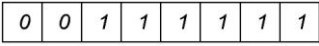
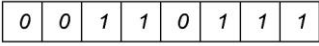
	<i>AC</i> flags are cleared.				
<i>XRA r</i>	The contents of register <i>r</i> is exclusive-or'd with the contents of the accumulator. The result is placed in the accumulator. The <i>CY</i> and <i>AC</i> flags are cleared.		1 (4)	register	S,Z,P ,CY, AC
<i>XRA M</i>	The contents of the memory location whose address is contained in the H and L registers is exclusive-OR'd with the contents of the accumulator. The result is placed in the accumulator. The <i>CY</i> and <i>AC</i> flags are cleared.		2 (7)	register indirect	S,Z,P ,CY, AC
<i>XRI data</i>	The contents of the second byte of the instruction is exclusive-OR'd with the contents of the accumulator. The result is placed <i>in</i> the accumulator.	 	2 (7)	immediate	S,Z,P ,CY, AC

	The <i>CY</i> and <i>AC</i> flags are cleared.																				
<i>ORA r</i>	The contents of register <i>r</i> is inclusive-OR'd with the contents of the accumulator. The result is placed in the accumulator. The <i>CY</i> and <i>AC</i> flags are cleared.	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>S</td><td>S</td><td>S</td> </tr> </table>	1	0	1	1	0	S	S	S	1 (4)	register	S,Z,P ,CY, AC								
1	0	1	1	0	S	S	S														
<i>ORA M</i>	The contents of the memory location whose address is contained in the <i>H</i> and <i>L</i> registers is inclusive-OR'd with the contents of the accumulator. The result is Placed in the accumulator. The <i>CY</i> and <i>AC</i> flags are cleared.	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td> </tr> </table>	1	0	1	1	0	1	1	0	2 (7)	register indirect	S,Z,P ,CY, AC								
1	0	1	1	0	1	1	0														
<i>ORI data</i>	The contents of the second byte of the instruction is inclusive-OR'd with the contents of the accumulator. The result is placed in the	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td> </tr> <tr> <td colspan="8" style="text-align: center;"><i>data</i></td> </tr> </table>	1	1	1	1	0	1	1	0	<i>data</i>								2 (7)	immediate	S,Z,P ,CY, AC
1	1	1	1	0	1	1	0														
<i>data</i>																					

	accumulator. The <i>CY</i> and <i>AC</i> flags are cleared.												
<i>CMP r</i>	The contents of register <i>r</i> is subtracted from the accumulator. The accumulator remains unchanged. The condition flags are set as a result of the subtraction. The Z flag is set to I if $(A) = (r)$. The CY flag is set to I if $(A) < (r)$.	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>S</td><td>S</td><td>S</td> </tr> </table>	1	0	1	1	1	S	S	S	1 (4)	register	S,Z,P, CY, AC
1	0	1	1	1	S	S	S						
<i>CMP M</i>	The contents of the memory location whose address is contained in the H and L registers is subtracted from the accumulator. The accumulator remains unchanged. The condition flags are set as a result of the subtraction. The Z flag is set to I if $(A) = ((H) (L))$. The CY flag	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td> </tr> </table>	1	0	1	1	1	1	1	0	2 (7)	register indirect	S,Z,P, CY, AC
1	0	1	1	1	1	1	0						

	is set to <i>I</i> if $(A) < ((H)(L))$.																				
<i>CPI data</i>	The contents of the second byte of the instruction is subtracted from the accumulator. The condition flags are set by the result of the subtraction. The <i>Z</i> flag is set to <i>I</i> if $(A) = (\text{byte } 2)$. The <i>CY</i> flag is set to <i>I</i> if $(A) < (\text{byte } 2)$.	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">1</td><td style="text-align: center;">1</td><td style="text-align: center;">1</td><td style="text-align: center;">1</td><td style="text-align: center;">1</td><td style="text-align: center;">1</td><td style="text-align: center;">1</td><td style="text-align: center;">0</td> </tr> <tr> <td colspan="8" style="text-align: center;"><i>data</i></td> </tr> </table>	1	1	1	1	1	1	1	0	<i>data</i>								2 (7)	immediate	S,Z,P,CY,AC
1	1	1	1	1	1	1	0														
<i>data</i>																					
<i>RLC</i>	The contents of the accumulator is rotated left one position. The low order bit and the <i>CY</i> flag are both set to the value shifted out of the high order bit position	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">0</td><td style="text-align: center;">0</td><td style="text-align: center;">0</td><td style="text-align: center;">0</td><td style="text-align: center;">0</td><td style="text-align: center;">1</td><td style="text-align: center;">1</td><td style="text-align: center;">1</td> </tr> </table>	0	0	0	0	0	1	1	1	1 (4)	Implied	CY								
0	0	0	0	0	1	1	1														
<i>RRC</i>	The contents of the accumulator is rotated right one position. The high order bit and the <i>CY</i> flag are both set	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">0</td><td style="text-align: center;">0</td><td style="text-align: center;">0</td><td style="text-align: center;">0</td><td style="text-align: center;">1</td><td style="text-align: center;">1</td><td style="text-align: center;">1</td><td style="text-align: center;">1</td> </tr> </table>	0	0	0	0	1	1	1	1	1 (4)	Implied	CY								
0	0	0	0	1	1	1	1														

	to the value shifted out of the low order bit position.												
RAL	The contents of the accumulator is rotated left one position through the CY flag. The low order bit is set equal to the CY flag and the CY flag is set to the value shifted out of the high order bit.	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td> </tr> </table>	0	0	0	1	0	1	1	1	1 (4)	Implied	CY
0	0	0	1	0	1	1	1						
RAR	The contents of the accumulator is rotated right one position through the CY flag. The high order bit is set to the CY flag and the CY flag is set to the value shifted out of the low order bit.	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td> </tr> </table>	0	0	0	1	1	1	1	1	1 (4)	Implied	CY
0	0	0	1	1	1	1	1						
CMA	The contents of the accumulator are complemented (zero bits become 1 , one bits become 0).	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td> </tr> </table>	0	0	1	0	1	1	1	1	1 (4)	Implied	none
0	0	1	0	1	1	1	1						

CMC	The <i>CY</i> flag is complemented.		1 (4)	Implied	CY
STC	The <i>CY</i> flag is set to 1.		1 (4)	Implied	CY

Example 4.16: Using *ANA r*, *ANA M* and *ANI data* commands.

Mnemonic	Operands	Machine code	Comment
MVI	A,05h	3E 05	loading of <i>A</i> register: $A \leftarrow 05h$
LXI	H, 0100h	21 00 01	loading of <i>HL</i> register pair: $HL \leftarrow 0100h$
MOV	M,A	77	$(HL) \leftarrow A$
ANA	A	A7	$A \leftarrow A \wedge A = 05h$
ANA	M	A6	$A \leftarrow A \wedge (M) = 05h$
ANI	04h	E6 04	$A \leftarrow A \wedge 04h = 04h$

Example 4.17: Using *ORA r*, *ORA M* and *ORI data* commands.

Mnemonic	Operands	Machine code	Comment
MVI	A,05h	3E 05	loading of <i>A</i> register: $A \leftarrow 05h$
LXI	H, 0100h	21 00 01	loading of <i>HL</i> register pair: $HL \leftarrow 0100h$
MOV	M,A	77	$(HL) \leftarrow A$
ORA	A	B7	$A \leftarrow A \vee A = 05h$
ORA	M	B6	$A \leftarrow A \vee (M) = 05h$
ORI	08h	F6 08	$A \leftarrow A \vee 04h = 0Dh$

Example 4.18: Using *XRA r*, *XRA M* and *XRI data* commands.

Mnemonic	Operands	Machine code	Comment
MVI	A,0AAh	3E AA	loading of <i>A</i> register: $A \leftarrow AAh$
LXI	H, 0100h	21 00 01	loading of <i>HL</i> register pair: $HL \leftarrow 0100h$
MOV	M,A	77	$(HL) \leftarrow A$
ORA	A	AF	$A \leftarrow A \vee A = 00h$
ORA	M	AE	$A \leftarrow A \vee (M) = AAh$
ORI	0FFh	EE FF	$A \leftarrow A \vee 04h = 55h$

Example 4.19: Using *RAL*, *RAR* and *CMA* commands.

<i>Mnemonic</i>	<i>Operands</i>	<i>Machine code</i>	<i>Comment</i>
MVI	A,0Ch	3E 0C	loading of A register: $A \leftarrow 0Ch$
CMC		1F	$A \leftarrow 06h$
RAR		1F	$A \leftarrow 03h$
MVI	A,0Ch	3E 0C	loading of A register: $A \leftarrow 0Ch$
RAL		17	$A \leftarrow 18h$
RAL		17	$A \leftarrow 30h$
CMA		2F	$A \leftarrow CFh$

Example 4.20: Using *RRC*, *RLC*, *CMC* and *STC* commands.

<i>Mnemonic</i>	<i>Operands</i>	<i>Machine code</i>	<i>Comment</i>
MVI	A,0Ch	3E 0C	loading of A register: $A \leftarrow 0Ch$
CMC		3F	Clear <i>carry</i> flag
RRC		0F	$A \leftarrow 06h$, <i>carry</i> =0
STC		37	Set <i>carry</i> flag
RRC		0F	$A \leftarrow 83h$, <i>carry</i> =0
MVI	A,0Ch	3E 0C	loading of A register: $A \leftarrow 0Ch$
RLC		07	$A \leftarrow 18h$, <i>carry</i> =0
STC		37	Set <i>carry</i> flag
RLC		07	$A \leftarrow 31h$, <i>carry</i> =0

- **Branch Group** — conditional and unconditional jump instructions, subroutine call instructions and return instructions [7, 8];

The two types of branch instructions are unconditional and conditional. Unconditional transfers simply perform the specified operation on register *PC* (the program counter). Conditional transfers examine the status of one of the four processor flags to determine if the specified branch is to be executed. The conditions that may be specified are as follows:

Condition	CCC	Instruction
NZ – not zero (Z=0)	000	<i>JNZ</i>
Z – zero (Z=1)	001	<i>JZ</i>
NC – not carry (C=0)	010	<i>JNC</i>
C – carry (C=1)	011	<i>JC</i>
P – parity odd (P=0)	100	<i>JP</i>
PE – parity even (P=1)	101	<i>JPE</i>

P – plus (S=0)	110	JP
M – minus (S=1)	111	JM

Table 4.5. Branch group instructions

Instruction	Action	Instruction format	Cycles (states)	Addressing	Flags																								
JMP addr	Control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction.	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td> </tr> <tr> <td colspan="8" style="text-align: center;"><i>low-order address</i></td> </tr> <tr> <td colspan="8" style="text-align: center;"><i>high-order address</i></td> </tr> </table>	1	1	0	0	0	0	1	1	<i>low-order address</i>								<i>high-order address</i>								3 (10)	immediate	none
1	1	0	0	0	0	1	1																						
<i>low-order address</i>																													
<i>high-order address</i>																													
Jcondition addr	If the specified condition is true, control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction; otherwise, control continues sequentially.	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>1</td><td>C</td><td>C</td><td>C</td><td>0</td><td>1</td><td>0</td> </tr> <tr> <td colspan="8" style="text-align: center;"><i>low-order address</i></td> </tr> <tr> <td colspan="8" style="text-align: center;"><i>high-order address</i></td> </tr> </table>	1	1	C	C	C	0	1	0	<i>low-order address</i>								<i>high-order address</i>								3 (10)	immediate	none
1	1	C	C	C	0	1	0																						
<i>low-order address</i>																													
<i>high-order address</i>																													
CALL addr	The high-order eight bits of the next instruction address are moved to the memory location whose address is one less than	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td> </tr> <tr> <td colspan="8" style="text-align: center;"><i>low-order address</i></td> </tr> <tr> <td colspan="8" style="text-align: center;"><i>high-order address</i></td> </tr> </table>	1	1	0	0	1	1	0	1	<i>low-order address</i>								<i>high-order address</i>								5 (17)	Immediate/register indirect	none
1	1	0	0	1	1	0	1																						
<i>low-order address</i>																													
<i>high-order address</i>																													

	<p>the contents of register SP. The low-order eight bits of the next instruction address are moved to the memory location whose address is two less than the contents of register SP. The contents of register SP is decremented by 2. Control is transferred to the instruction whose address is specified in <i>byte 3</i> and <i>byte 2</i> of the current instruction.</p>												
RET	<p>The contents of the memory location whose address is specified in register SP is moved to the low-order eight bits of register PC. The contents of the memory location whose address is one more</p>	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">1</td> </tr> </table>	1	1	0	0	1	0	0	1	3 (10)	Register indirect	none
1	1	0	0	1	0	0	1						

	than the contents of register <i>SP</i> is moved to the high-order eight bits of register <i>PC</i> . The contents of register <i>SP</i> is incremented by 2.												
Recondition	If the specified condition is true, the actions specified in the RET instruction (see above) are performed; otherwise, control continues sequentially.	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>1</td><td>C</td><td>C</td><td>C</td><td>0</td><td>0</td><td>0</td> </tr> </table>	1	1	C	C	C	0	0	0	3 (11)	Register indirect	none
1	1	C	C	C	0	0	0						
RST <i>n</i>	The high-order eight bits of the next instruction address are moved to the memory location whose address is one less than the contents of register <i>SP</i> . The low-order eight bits of the next instruction address are moved to the memory location whose ad-	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>1</td><td>N</td><td>N</td><td>N</td><td>1</td><td>1</td><td>1</td> </tr> </table> <p><i>n</i> - The restart number 0 through 7. <i>NNN</i> - The binary representation 000 through 111 for restart number 0 through 7 respectively.</p>	1	1	N	N	N	1	1	1	3 (11)	Register indirect	none
1	1	N	N	N	1	1	1						

	dress is two less than the contents of register <i>SP</i> . The contents of register <i>SP</i> is decremented by two. Control is transferred to the instruction whose address is eight times the contents of <i>NNN</i> .												
PCHL	The contents of register <i>H</i> is moved to the high-order eight bits of register <i>PC</i> . The contents of register <i>L</i> is moved to the low-order eight bits of register <i>PC</i> .	<table border="1" style="margin: auto;"> <tr> <td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td> </tr> </table>	1	1	1	0	1	0	0	1	1 (5)	Register	none
1	1	1	0	1	0	0	1						

Example 4.21: Using *JMP addr* and *Jcondition addr* commands.

Addr	Mnemonic	Operands	Machine code	Comment
0000h	MVI	A,0Ch	3E 0C	loading of <i>A</i> register: $A \leftarrow 0Ch$
0002h	JMP	0100h	C3 00 01	Jump to address <i>0100h</i>
0100h	ADI	A,0FFh	C6 FF	$A \leftarrow A + 0FFh = 0Bh$, <i>Carry</i> =1
0102h	JC	0200h	D2 00 02	Jmp if flag <i>carry</i> is set 1
0200h	MVI	A,00h	3E 00	loading of <i>A</i> register: $A \leftarrow 00h$

The result of executing this program:

$A = 00h$

Example 4.22: Using *CALL addr* and *RET* commands.

Addr	Mnemonic	Operands	Machine code	Comment
0000h	MVI	A,0Ch	3E 0C	loading of A register: $A \leftarrow 0Ch$
0002h	CALL	0100h	CD 00 01	Subprogram call
Subprogram				
0100h	ADI	A,0FFh	C6 FF	$A \leftarrow A + 0FFh = 0Bh$, Carry=1
0102h	RET		C9	Return to the main program

The result of executing this program:

$A = 0Bh$

Example 4.23: Using *PCHL* command.

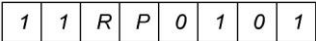
Addr	Mnemonic	Operands	Machine code	Comment
0000h	MVI	A,0Ch	3E 0C	loading of A register: $A \leftarrow 0Ch$
0002h	LXI	H, 0100h	21 00 01	$HL \leftarrow 0100h$
0005h	PCHL			$PC \leftarrow HL$
0100h	ADI	A,0FFh	C6 FF	$A \leftarrow A + 0FFh = 0Bh$, Carry=1

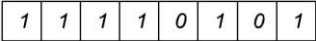
The result of executing this program:

$A = 0Bh$

- **Stack, I/O and Machine Control Group** – includes I/O instructions, as well as instructions for maintaining the stack and internal control flags [7, 8].

Table 4.5. Stack group instructions

Instruction	Action	Instruction format	Cycles (states)	Addressing	Flags
<i>PUSH rp</i>	The contents of the high-order register of register pair <i>rp</i> is moved to the memory location whose address is one less than		3 (11)	Register indirect	none

	<p>the contents of register SP. The contents of the low-order register of register pair rp is moved to the memory location whose address is two less than the contents of register SP. The contents of register SP is decremented by 2. Note: Register pair rp = SP may not be specified.</p>				
PUSH PSW	<p>The contents of register A is moved to the memory location whose address is one less than register SP. The contents of the condition flags are assembled into a processor status word and the word is moved to the memory location whose address is two less than the</p>		3 (11)	Register indirect	none

	<p>contents of register <i>SP</i>. The contents of register <i>SP</i> is decremented by two.</p>												
POP <i>rp</i>	<p>The contents of the memory location, whose address is specified by the contents of register <i>SP</i>, is moved to the low-order register of register pair <i>rp</i>. The contents of the memory location, whose address is one more than the contents of register <i>SP</i>, is moved to the high-order register of register pair <i>rp</i>. The contents of register <i>SP</i> is incremented by 2. Note: Register pair <i>rp</i> = <i>SP</i> may not be specified.</p>	<div style="border: 1px solid black; padding: 2px; display: inline-block;"> <table style="border-collapse: collapse; text-align: center;"> <tr> <td style="border: 1px solid black; padding: 2px 5px;">1</td> <td style="border: 1px solid black; padding: 2px 5px;">1</td> <td style="border: 1px solid black; padding: 2px 5px;"><i>R</i></td> <td style="border: 1px solid black; padding: 2px 5px;"><i>P</i></td> <td style="border: 1px solid black; padding: 2px 5px;">0</td> <td style="border: 1px solid black; padding: 2px 5px;">0</td> <td style="border: 1px solid black; padding: 2px 5px;">0</td> <td style="border: 1px solid black; padding: 2px 5px;">1</td> </tr> </table> </div>	1	1	<i>R</i>	<i>P</i>	0	0	0	1	3 (10)	Register indirect	none
1	1	<i>R</i>	<i>P</i>	0	0	0	1						

POP PSW	The contents of the memory location whose address is specified by the contents of register SP is used to restore the condition flags. The contents of the memory location whose address is one more than the contents of register SP is moved to register A. The contents of register SP is incremented by 2.	<table border="1" style="margin: auto;"> <tr> <td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td> </tr> </table>	1	1	1	1	0	0	0	1	3 (10)	Register indirect	Z,S,P,AC,CY
1	1	1	1	0	0	0	1						

Example 4.24: Using of *PUSH rp* and *POP rp* command.

Addr	Mnemonic	Operands	Machine code	Comment
0000h	LXI	H, 1122h	21 22 11	<i>HL←1122h</i>
0002h	LXI	SP,0202h	31 02 02	<i>SP←0202h</i>
0005h	PUSH	H	E5	<i>(SP-1)=0201h←H</i> <i>(SP-1)=0200h←L</i>
0006h	POP	D	D1	<i>E←(0200h),</i> <i>SP←SP+1=0201h</i> <i>D←(0201h),</i> <i>SP←SP+1=0202h</i>

The result of executing this program:

H=11h

L=22h

D=11h

$$E=22h$$

PART 5. 8051 MICROCONTROLLERS

Microcontroller usually consists of microprocessor unit, memory, usually read-only program memory (*ROM*) and random-access data memory (*RAM*), memory decoders, an oscillator, and a number of input/output (*I/O*) devices, such as parallel and serial data ports, analog to digital and digital to analog converters. Additionally, special-purpose devices, such as interrupt handlers, or counters, may be added to relieve the *CPU* from time-consuming counting or timing chores. Equipping the microcomputer with a mass storage device, commonly a floppy disk drive, and *I/O* peripherals, such as a keyboard and a *CRT* display, yields a small computer that can be applied to a range of general-purpose software applications.

The prime use of a microprocessor is to fetch data, perform extensive calculations on that data, and store those calculations in a mass storage device or display the results for human use. The programs used by the microprocessor are stored in the mass storage device and loaded into *RAM* as the user directs. A few microprocessor programs are stored in *ROM*. The *ROM*-based programs are primarily small fixed programs that operate peripherals and other fixed devices that are connected to the system.

Figure 5.1 shows the block diagram of a typical *microcontroller*. The design incorporates all the features found in a microprocessor *CPU*: *ALU*, *PC*, *SP*, and registers. It also has other features: *ROM*, *RAM*, parallel *I/O*, serial *I/O*, counters, and a clock circuit [9].

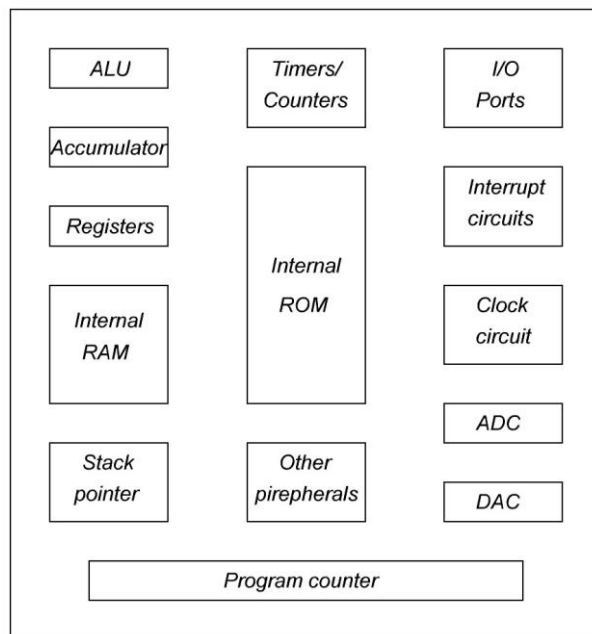


Fig. 5.1. A block diagram of microcontroller

Like the microprocessor, a microcontroller is a general-purpose device, but one which is meant to fetch data, perform limited calculations on that data, and control its environment based on those calculations. The prime use of a microcontroller is to control the operation of a machine using a fixed program that is stored in *ROM* and that does not change over the lifetime of the system.

The design approach of the microcontroller mirrors that of the microprocessor: make a single design that can be used in as many applications as possible in order to sell, hopefully, as many as possible. The microprocessor design accomplishes this goal by having a very flexible and extensive repertoire of multi-byte instructions. These instructions work in a hardware configuration that enables large amounts of memory and *I/O* to be connected to address and data bus pins on the integrated circuit package. Much of the activity in the microprocessor has to do with moving code and data words to and from *external* memory to the *CPU*. The architecture features working registers that can be programmed to take part in the memory access process, and the instruction set is aimed at expediting this activity in order to improve throughput. The pins that connect the microprocessor to external memory are unique, each having a single function. Data is handled in byte, or larger, sizes.

The microcontroller design uses a much more limited set of single- and double-byte instructions that are used to move code and data from *internal* memory to the *ALU*. Many instructions are coupled with pins on the integrated circuit package; the pins are “programmable”—that is, capable of having several different functions depending upon the wishes of a programmer.

The microcontroller is concerned with getting data from and to its own pins; the architecture and instruction set are optimized to handle data in bit and byte size.

The contrast between a microcontroller and a microprocessor is best exemplified by the fact that most microprocessors have many operational codes (operational codes) for moving data from external memory to the *CPU*; microcontrollers may have one, or two. Microprocessors may have one or two types of bit-handling instructions; microcontrollers will have many.

To summarize, the microprocessor is concerned with rapid movement of code and data from external addresses to the chip; the microcontroller is concerned with rapid movement of bits within the chip. The microcontroller can function as a computer with the addition of no external digital parts; the microprocessor must have many additional parts to be operational.

In this part we will describe the principles of operation of microcontrollers by the example of 8-bit 8051 microcontroller [9].

5.1. THE 8051 ARCHITECTURE

Figure 5.2 shows a functional block of the internal operation of an 8051 microcomputer [9].

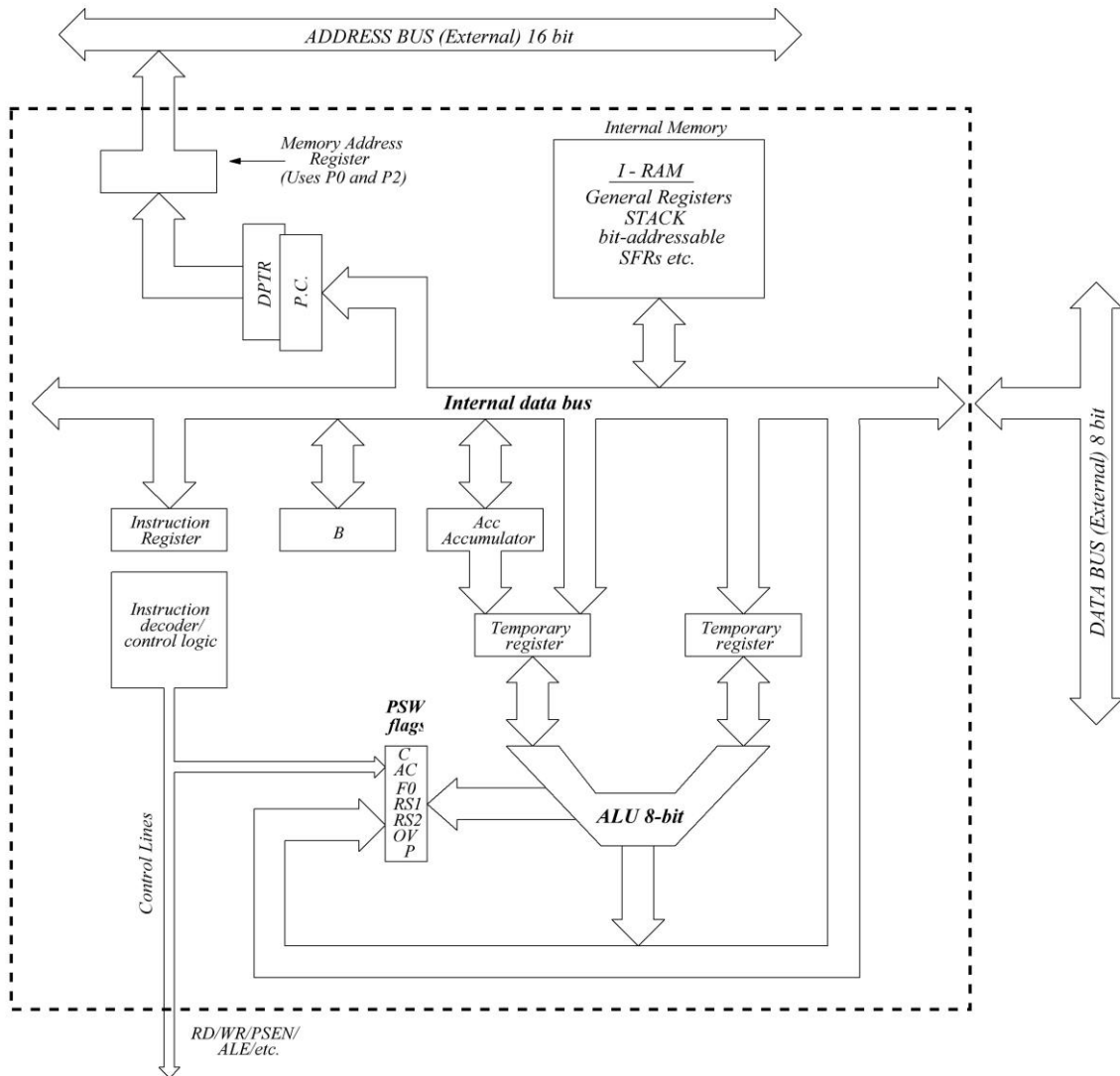


Fig. 5.2. 8051 functional block diagram

The 8051 has a separate memory space for code (programs) and data. There can be two types of memory on-chip memory and external memory as shown in Fig. 5.3. In an actual implementation the external memory may, be contained within the microcomputer chip. However, we will use the definitions of internal and external memory to be consistent with 8051 instructions which operate on memory. Note, the separation of the code and data memory in the 8051 architecture is a little unusual. The separated memory architecture

is referred to as Harvard architecture whereas von Neumann architecture defines a system where code and data can share common memory.

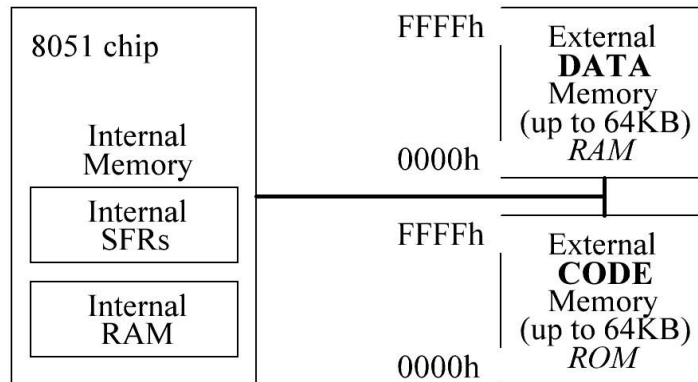


Fig. 5.3. 8051 memory representation

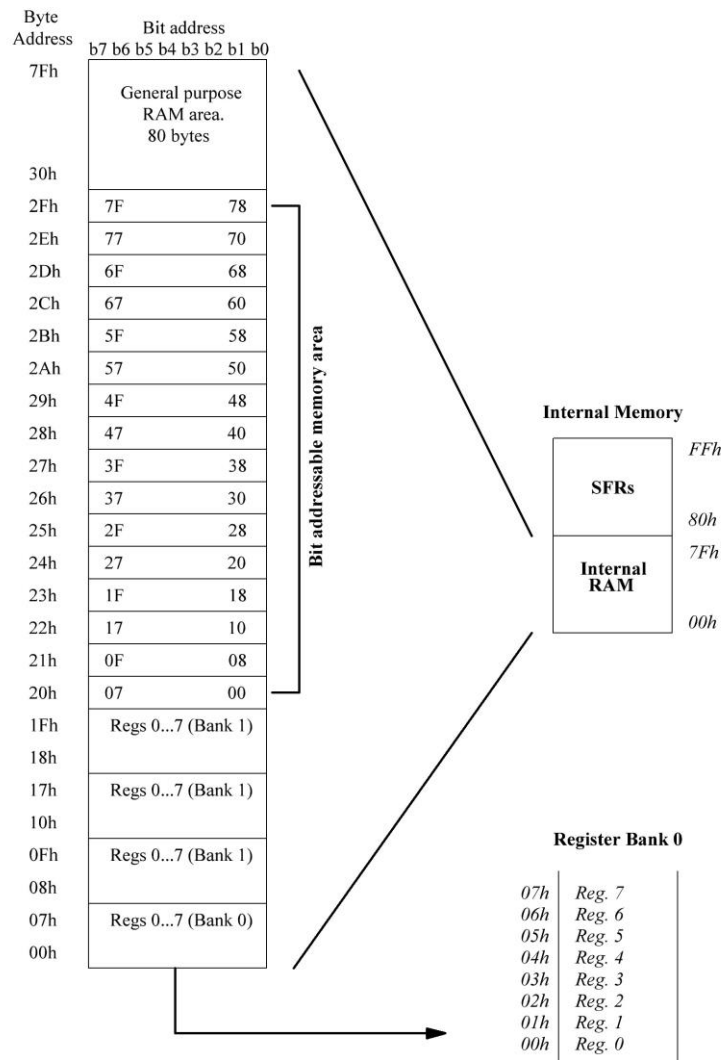


Fig. 5.4. Internal RAM of 8051 microcontroller

The executable program code is stored in this code memory. The code memory size is limited to 64 *KBytes* (in a standard 8051). The code memory is read-only in normal operation and is programmed under special conditions, e.g. it is a *PROM* or a Flash *RAM* types of memory.

External RAM Data Memory

This is read-write memory and is available for storage of data. Up to 64KBytes of external RAM data memory is supported (in a standard 8051).

Internal Memory

The 8051’s on-chip memory consists of 256 memory bytes organized as follows:

- First 128 bytes: 00h to 1Fh Register Banks
 20h to 2Fh Bit Addressable *RAM*
 30 to 7Fh General Purpose *RAM*
- Next 128 bytes: 80h to FFh Special Function Registers

The first 128 bytes of internal memory is organized as shown in Fig 5.4, and is referred to as Internal RAM.

Register Banks

The 8051 uses 8 general-purpose registers R0 through **R7** (**R0**, **R1**, **R2**, **R3**, **R4**, **R5**, **R6**, and **R7**). These registers are used in instructions such as:

- ADD A, R2**; adds the value contained in R2 to the accumulator
- MOV R1,A**; the contents of accumulator moves into R1

Note that the following instructions have the same effect as the above instruction.

ADD A,02h
MOV 01h,A

The picture becomes more complicated when we see that there are four banks of these general-purpose registers defined within the Internal *RAM*. For the moment we will consider register bank 0 only. Register banks 1 to 3 can be ignored when writing introductory level assembly language programs. The bank number is selected in *PSW* register. In table 5 the contents of *PSW* register is given [9].

Table 5.1. *PSW* register.

Symbol	Number of bit	Description
C	PSW.7	Carry flag. This is a conventional carry, or borrow, flag used in arithmetic operations. The carry flag is also used as the ‘Boolean accumulator’ for Boolean instruction operating at the bit level. This flag is sometimes referenced

		as the CY flag.
AC	PSW.6	Auxiliary carry flag. This is a conventional auxiliary carry (half carry) for use in BCD arithmetic.
F0	PSW.5	Zero flag (flag 0). This is a general-purpose flag for user programming.
RS1	PSW.4	Register bank select 1.
RS0	PSW.3	Register bank select 0. These bits define the active register bank (bank 0 is the default register bank).
OV	PSW.2	Overflow flag. This is a conventional overflow bit for signed arithmetic to determine if the result of a signed arithmetic operation is out of range.
-	PSW.1	Reserved.
P	PSW.0	Even parity flag. The parity flag is the accumulator parity flag, set to a value, 1 or 0, such that the number of '1' bits in the accumulator plus the parity bit add up to an even number.

Table 5.2. Register banks.

RS1	RS0	Bank	Addresses
0	0	0	00H – 07H
0	1	1	08H – 0FH
1	0	2	10H – 17H
1	1	3	18H – 1FH

Stack Pointer

The Stack Pointer (*SP*) is an 8-bit *SFR* register. The small address field (8 bits) and the limited space available in the Internal *RAM* confines the stack size and this is sometimes a limitation for 8051 programmes. The *SP* contains the address of the data byte currently on the top of the stack. The *SP* pointer is initialized to a defined address. A new data item is 'pushed' onto the stack using a *PUSH* instruction which will cause the data item to be written to address *SP* + 1. Typical instructions, which cause modification to the stack are: *PUSH*, *POP*, *LCALL*, *RET*, *RETI* etc.. The *SP SFR*, on start-up, is loaded by number 07h and this means the stack will start at 08h and ex-

pand upwards in Internal RAM. If register banks 1 to 3 are to be used the SP SFR should be initialized to start higher up in Internal RAM. The following instruction is used to initialize the stack:

MOV SP, #2Fh

When a subroutine is called, the current contents of the Program Counter (**PC**) is saved into the stack, the low byte of the PC is saved first followed by the high byte. Thus the Stack Pointer (**SP**) is incremented by 2. When a **RET** (return from subroutine) instruction is executed, the stored **PC** value on the stack is restored to the PC, thus decrementing the **SP** by 2.

When a byte is **PUSHed** to the stack, the **SP** is incremented by one so as to point to the next available stack location. Conversely, when a byte is **POPped** from the stack, the **SP** is decremented by one. Fig. 5.5 shows the organization of the stack area within the **I-RAM** memory space.

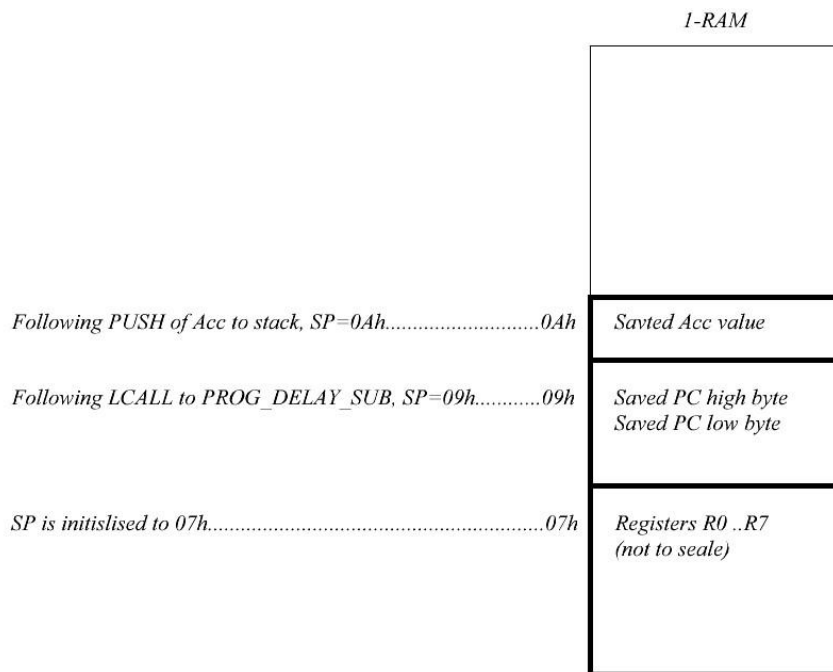


Fig. 5.5. The stack operation

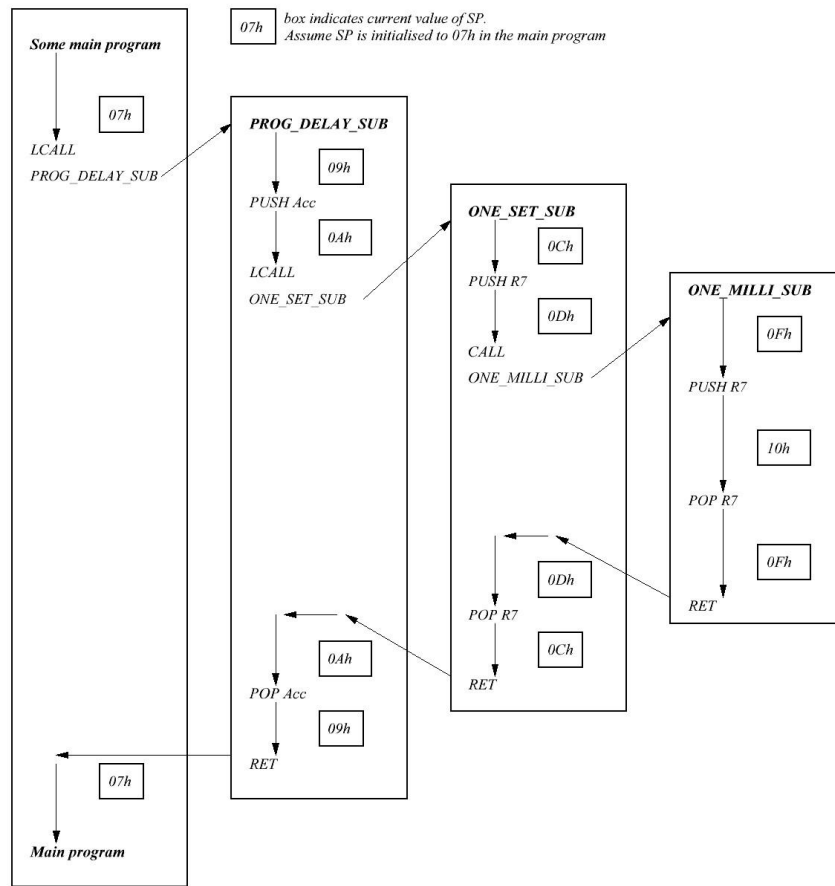


Fig. 5.6. The stack operation

The stack values during the operation of the nested subroutine example are shown in Fig. 5.6. Here it is assumed that the *SP* is initialized to 07h. This is possible when the alternative register banks are not used in the program. The stack then has a maximum value of 20h, if we want to preserve the ‘bit addressable’ *RAM* area. It is more common to initialize the *SP* higher up in the internal *RAM* at location 2Fh. The diagram shows how data is saved into the stack [9, 10].

Data Pointer

The Data Pointer register (*DPTR*) is a special 16-bit register used to address the external code or external data memory. Since the *SFR* registers are just 8-bits wide the *DPTR* is stored in two *SFR* registers, where *DPL* holds the low byte of the *DPTR* and *DPH* holds the high byte of the *DPTR*. For example, for writing down the value 0FFh to external data memory location 1020h, one might use the following instructions:

MOV A, #0FFh; move immediate 8 bit data 46h to accumulator

MOV DPTR, #1020h ; Move immediate 16 bit address value 1020h to A. Now DPL holds 20h and **DPH** holds 10h.

MOVX @DPTR, A ; Move the value in A to external *RAM* location 1020h. Note the **MOVX** instruction is used to access external memory.

Accumulator

This is a conventional accumulator that one expects to find in any computer, which is used to hold the results of various arithmetic and logic operations. Since the 8051 microcontroller is just an 8-bit device, the accumulator is, as expected, an 8 bit register. The accumulator, referred to as **ACC** (in assembler **ACC** means address of accumulator) or A, is usually accessed explicitly using instructions such as:

INC A ; increment the accumulator

However, the accumulator is defined as an **SFR** register at address E0h. So the following two instructions have the same effect:

MOV A, #0FFh ; move immediate the value 0FFh to the accumulator

MOV ACC, #0FFh ; move immediate the value 0FFhh to Internal *RAM* location E0h (it is address of accumulator), which is, in fact, the accumulator **SFR** register.

Usually the first method, **MOV A, #0FFh**, is used as it requires less space - 2 bytes.

B Register

The **B** register is an **SFR** register at addresses **F0h** which is bit-addressable. The **B** register is used in two instructions only: i.e. **MUL** (multiply) and **DIV** (divide). The **B** register can also be used as a general-purpose register.

Program Counter

The **PC** (Program Counter) is a 2 byte (16 bit) register which always contains the memory address of the next instruction to be executed. When the 8051 is reset the **PC** is always initialized to **0000h**. If a 2 byte instruction is executed the **PC** is incremented by 2 and if a 3 byte instruction is executed the **PC** is incremented by three so as to correctly point to the next instruction to be executed. A jump instruction (e.g. **LJMP**) has the effect of causing the program to branch to a newly specified location, so the jump instruction causes the **PC** contents to change to the new address value. Jump instructions cause the program to flow in a non-sequential fashion, as will be described later.

SFR Registers for the Internal Timer

The set up and operation of the on-chip hardware timers will be described later, but the associated registers are briefly described here: **TCON**, the Timer Control register is an **SFR** at address 88h, which is bit-addressable.

TCON is used to configure and monitor the 8051 timers. The *TCON SFR* also contains some interrupt control bits, described later. *TMOD*, the Timer Mode register is an *SFR* at address 89h and is used to define the operational modes for the timers, as will be discussed below.

TLO (Timer 0 Low) and *TH0* (Timer 0 High) are two *SFR* registers addressed at 8Ah and 8Bh respectively. The two registers are associated with Timer 0.

T1L (Timer 1 Low) and *TH1* (Timer 1 High) are two *SFR* registers addressed at 8Ch and 8Dh respectively. These two registers are associated with Timer 1.

Power Control Register

PCON (Power Control) register is an *SFR* at address 87h. It contains various control bits including a control bit, which allows the 8051 to go to 'sleep' so as to save power when not in immediate use.

Serial Port Registers

Programming of the on-chip serial communications port will be described later in the text. The associated *SFR* registers, *SBUF* and *SCON*, are briefly introduced here, as follows:

The *SCON* (Serial Control) is an *SFR* register located at addresses 98h, and it is bit-addressable. *SCON* configures the behavior of the on-chip serial port, setting up parameters such as the baud rate of the serial port, activating send and/or receive data, and setting up some specific control flags.

The *SBUF* (Serial Buffer) is an *SFR* register located at address 99h. *SBUF* is just a single byte deep buffer used for sending and receiving data via the on-chip serial port.

Interrupt Registers

Interrupts will be discussed in more detail in other sections. The associated *SFR* registers are:

IE (Interrupt Enable) is an *SFR* register at addresses A8h and is used to enable and disable specific interrupts. The MSB bit (bit 7) is used to disable all interrupts.

IP (Interrupt Priority) is an *SFR* register at addresses B8h and it is bit addressable.

The *IP* register specifies the relative priority (high or low priority) of each interrupt. On the 8051, an interrupt may either be of low (0) priority or high (1) priority [9, 10, 11].

Control system and synchronization

Quartz crystal connected to the external terminals *X1* and *X2* to the 8051 microcontroller controls the internal oscillator, which in turn generates clock signals.

Control system of 8051 based on synchronization signals generates machine cycle of fixed duration equal to 12 periods of the resonator or six primary controlling machine states (*S1-S6*). Each state of control contains two phases P1, P2 of a resonator. In phase P1 is typically performed in an *ALU* operation, and the phase P2 is interregister transfer. The entire machine cycle consists of 12 phases, starting with phase and ending phase of the *S1P1-S6P2* (Fig. 5.7). This timing diagram illustrates the operation of the control system of 8051 while fetching and executing commands of various degrees of difficulty. ALE signal is generated twice in a single clock cycle (*S1P2-S2P1* and *S4P2-S5P1*) and is used to manage the access to external memory [9].

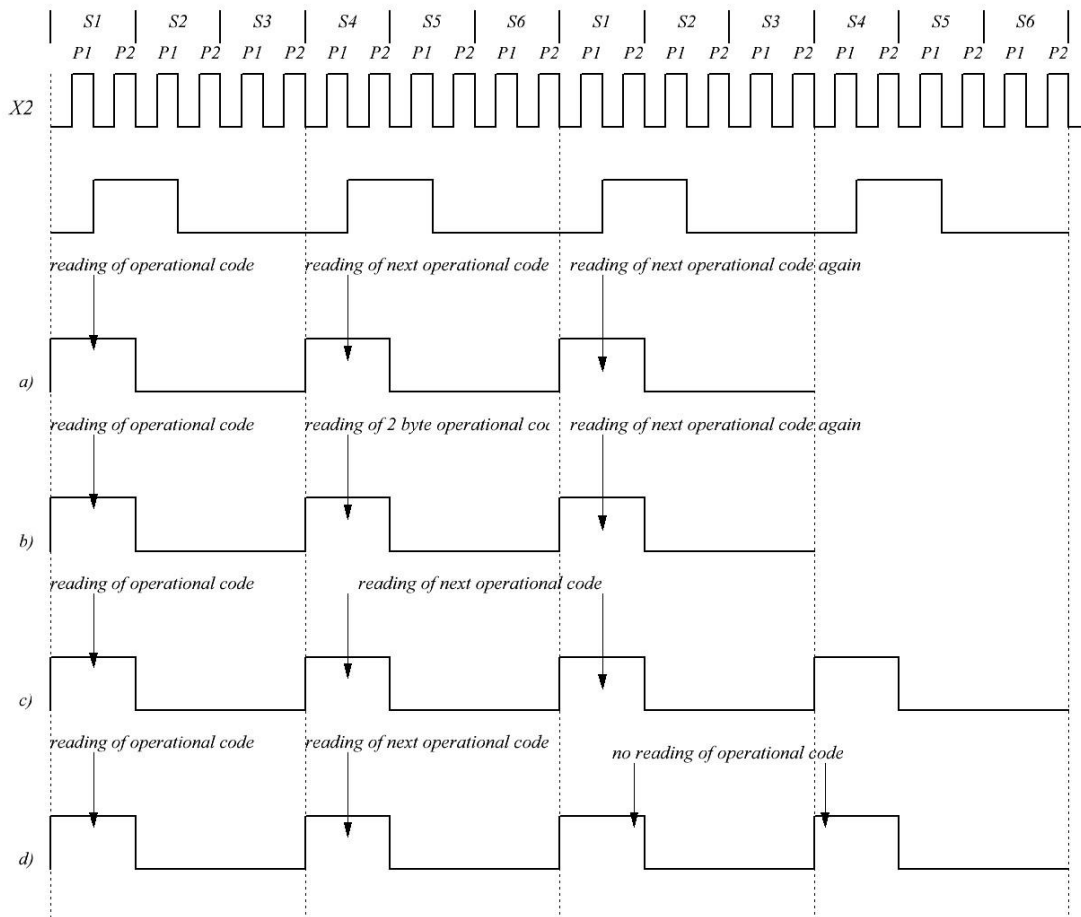


Fig. 5.7. Clock diagrams: a – instruction 1 byte/1 cycle, for example INC A; b – instruction 2 bytes/1 cycle, for example ADD A, #d; c – instruction 1 byte/2 cycle, for example INC DPTR; d – instruction 1 byte/2 cycle, for example MOVX;

Most commands are executed in a single machine cycle. Some of the commands that operate on a 2-byte words or related to the treatment of exter-

nal memory accesses are performed in two machine cycles. Only the operations of division and multiplication require four machine cycles.

Input/output ports

One major feature of a microcontroller is the versatility built into the input/output (*I/O*) circuits that connect the 8051 to the outside world. Microprocessor designs must add additional chips to interface with external circuitry; this ability is built into the microcontroller.

24 pins of the 8051 microcontroller may each be used for two entirely different functions. The function of a pin performed at any given instant depends, first, upon what is physically connected to it and, then, upon what software commands are used to “program” the pin.

Given this pin flexibility, the 8051 may be applied simply as a single component with *I/O* only, or it may be expanded to include additional memory, parallel ports, and serial data communication by using the alternate pin assignments. The key to programming an alternate pin function is the port pin circuitry shown in Fig. 5.8.

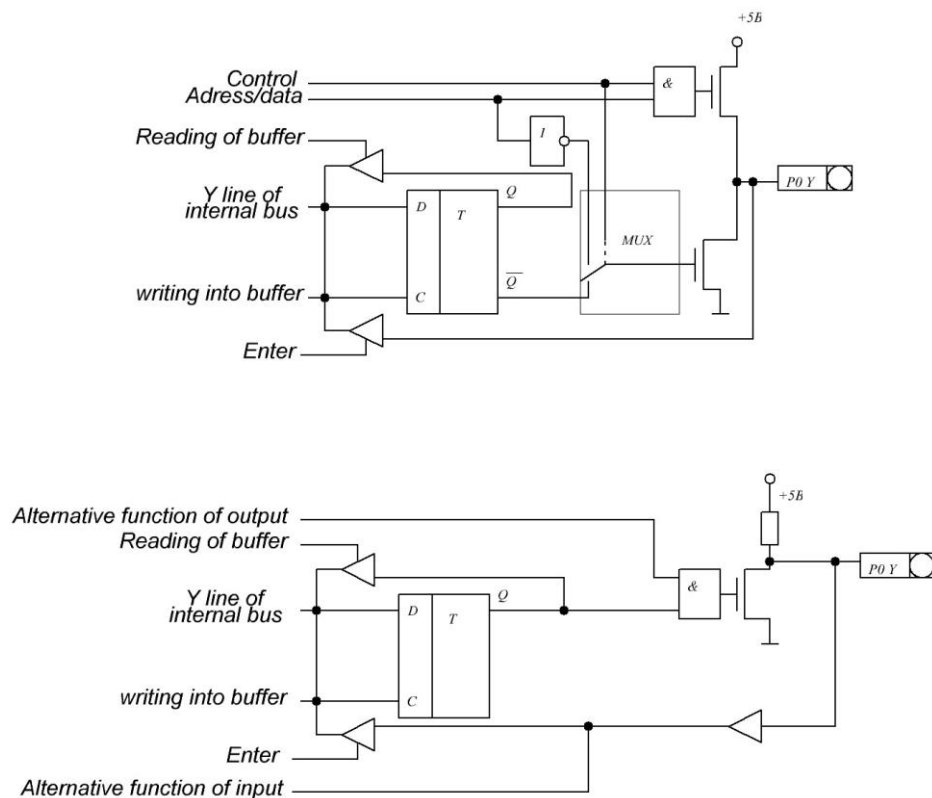


Fig. 5.8. Input/output ports

Each port has a *D*-type output latch for each pin. The *SFR* for each port is made up of these eight latches, which can be addressed at the *SFR* address

for that port. The port latches should not be confused with the port pins; the data on the latches does *not* have to be the same as that on the pins.

The two data paths are shown in Fig. 5.8 by the circuits that read the latch or pin data using two entirely separate buffers. The top buffer is enabled when latch data is read, and the lower buffer, when the pin state is read. The status of each latch may be read from a latch buffer, while an input buffer is connected directly to each pin so that the pin status may be read independently of the latch state.

Different operational codes access the latch or pin states as appropriate. Port operations are determined by the manner in which the 8051 is connected to external circuitry.

Programmable port pins have completely different alternate functions. The configuration of the control circuitry between the output latch and the port pin determines the nature of any particular port pin function.

Port 0

Port 0 pins may serve as inputs, outputs, or, when used together, as a bi-directional low-order address and data bus for external memory. For example, when a pin is to be used as an input, a 1 *must be* written to the corresponding port 0 latch by the program, thus turning both of the output transistors off, which in turn causes the pin to “float” in a high-impedance state, and the pin is essentially connected to the input buffer.

When used as an output, the pin latches that are programmed to a 0 will turn on the lower *FET*, grounding the pin. All latches that are programmed to a 1 still float; thus, external pull-up resistors will be needed to supply a logic high when using port 0 as an output.

When port 0 is used as an address bus to external memory, internal control signals switch the address lines to the gates of the Field Effect Transistors (*FETs*). A logic 1 on an address bit will turn the upper *FET* on and the lower *FET* off to provide a logic high at the pin. When the address bit is a zero, the lower *FET* is on and the upper *FET* off to provide a logic low at the pin. After the address has been formed and latched into external circuits by the Address Latch Enable (*ALE*) pulse, the bus is turned around to become a data bus. Port 0 now reads data from the external memory and must be configured as an input, so a logic 1 is automatically written by internal control logic to all port 0 latches [9, 10].

Port 1

Port 1 pins have no dual functions. Therefore, the output latch is connected directly to the gate of the lower *FET*, which has an *FET* circuit labeled “Internal *FET* Pull-up” as an active pull-up load.

Used as an input, a 1 is written to the latch, turning the lower *FET* off; the pin and the input to the pin buffer are pulled high by the *FET* load. An external circuit can overcome the high impedance pull-up and drive the pin low to input a 0 or leave the input high for a 1.

If used as an output, the latches containing a 1 can drive the input of an external circuit high through the pull-up. If a 0 is written to the latch, the lower *FET* is on, the pull-up is off, and the pin can drive the input of the external circuit low.

To aid in speeding up switching times when the pin is used as an output, the internal *FET* pull-up has another *FET* in parallel with it. The second *FET* is turned on for two oscillator time periods during a low-to-high transition on the pin, as shown in Fig. 5.8. This arrangement provides a low impedance path to the positive voltage supply to help reduce rise times in charging any parasitic capacitances in the external circuitry [9, 10].

Port 2

Port 2 may be used as an input/output port similar in operation to port 1. The alternate use of port 2 is to supply a high-order address byte in conjunction with the port 0 low-order byte to address external memory.

Port 2 pins are momentarily changed by the address control signals when supplying the high byte of a 16-bit address. Port 2 latches remain stable when external memory is addressed, as they do not have to be turned around (set to 1) for data input as is the case for port 0 [9, 10].

Port 3

Port 3 is an input/output port similar to port 1. The input and output functions can be programmed under the control of the P3 latches or under the control of various other special function registers. The port 3 alternate uses are shown in the following table [9, 10]:

Table 5.3. Alternative functions of Port 3

Symbol	Number of bit	Description
$\overline{\text{RD}}$	P3.7	External memory read pulse
$\overline{\text{WR}}$	P3.6	External memory write pulse
T1	P3.5	External timer 1 input
T0	P3.4	External timer 0 input
$\overline{\text{INT1}}$	P3.3	External interrupt 1

$\overline{INT0}$	P3.2	External interrupt 0
TXD	P3.1	Serial data output
RXD	P3.0	Serial data input

5.2. INTERRUPTS OF 8051

An interrupt causes a temporary diversion of program execution in a similar sense to a program subroutine call, but an interrupt is triggered by some event, external to the currently operating program. We say the interrupt event occurs asynchronously to the currently operating program as it is not necessary to know in advance when the interrupt event is going to occur.

There are five interrupt sources for the classical architecture of 8051. But in modern 8051 microcontrollers there are more than five interrupts. Since the main **RESET** input can also be considered as an interrupt, six interrupts are shown on Fig. 5.9:

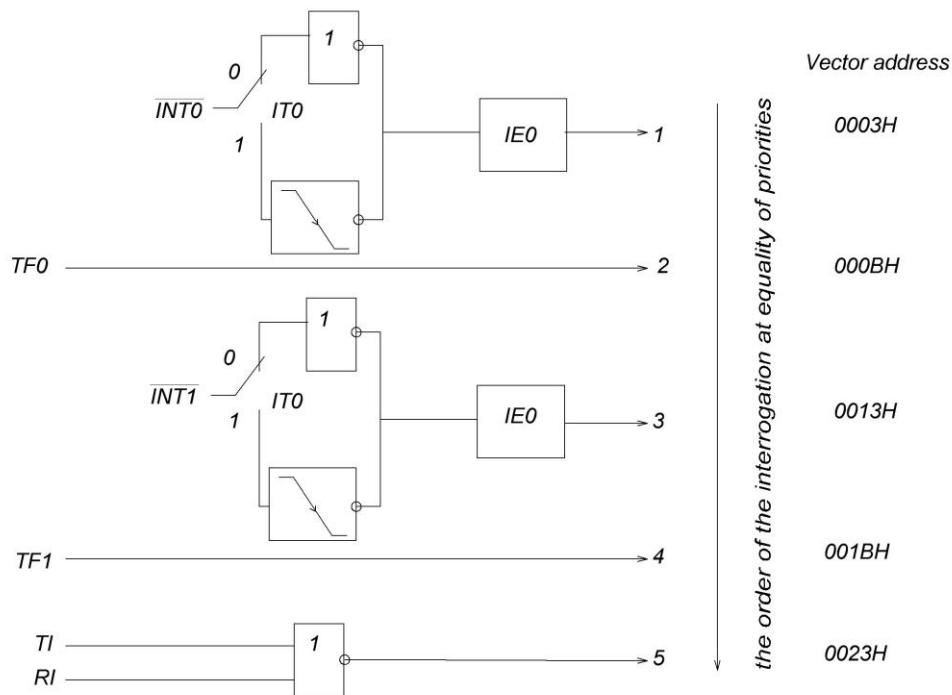


Fig. 5.9. Interrupts of 8051

Now, we will focus on the external interrupts for now, and later we will examine the other interrupt sources. Here is a brief look at some of the register bits which will be used to set up the interrupts in the example programs.

The Interrupt Enable, *IE*, register is an *SFR* register at location A8h in Internal RAM. The EA bit will enable all interrupts (when set to 1) and the individual interrupts must also be enabled.

Table 5.4. Interrupt enable register

Symbol	Number of bit	Description
EA	IE.7	Unlocking interrupts. Cleared by software to disable all interrupts regardless of the states of IE4 - IE0. 1 - enable, 0 - disable
-	IE.6	Reserved
-	IE.5	
ES	IE.4	UART interrupt enable bit. Set/reset by software to enable/disable interrupt flag TI or RI.
ET1	IE.3	Timer 1 interrupt enable bit. Set/reset by software to enable/disable timer 1 interrupt. 1 - enable, 0 - disable
EX1	IE.2	External interrupt 1 enable bit. Set/reset by software to enable/disable external interrupt 1. 1 - enable, 0 - disable
ET0	IE.1	Timer 0 interrupt enable bit. Set/reset by software to enable/disable timer 0 interrupt. 1 - enable, 0 - disable
EX0	IE.0	External interrupt 0 enable bit. Set/reset by software to enable/disable external interrupt 0. 1 - enable, 0 - disable

For example, if we want to enable the two external interrupts we would use the instruction:

MOV IE, #10000101B

Each of the two external interrupt sources can be defined to trigger on the external signal, either on a negative going edge or on a logic low level state. The negative edge trigger is usually preferred as the interrupt flag is automatically cleared by hardware, in this mode. Two bits in the *TCON* register are used to define the trigger operation. The *TCON* register is another *SFR* register and is located at location 88h in Internal RAM. The other bits in the *TCON* register will be described later in the context of the hardware Timer/Counters [10].

To define negative edge triggering for the two external interrupts one uses the following instructions:

SETB IT0 ; negative edge trigger for interrupt 0

SETB IT1 ; negative edge trigger for interrupt 1

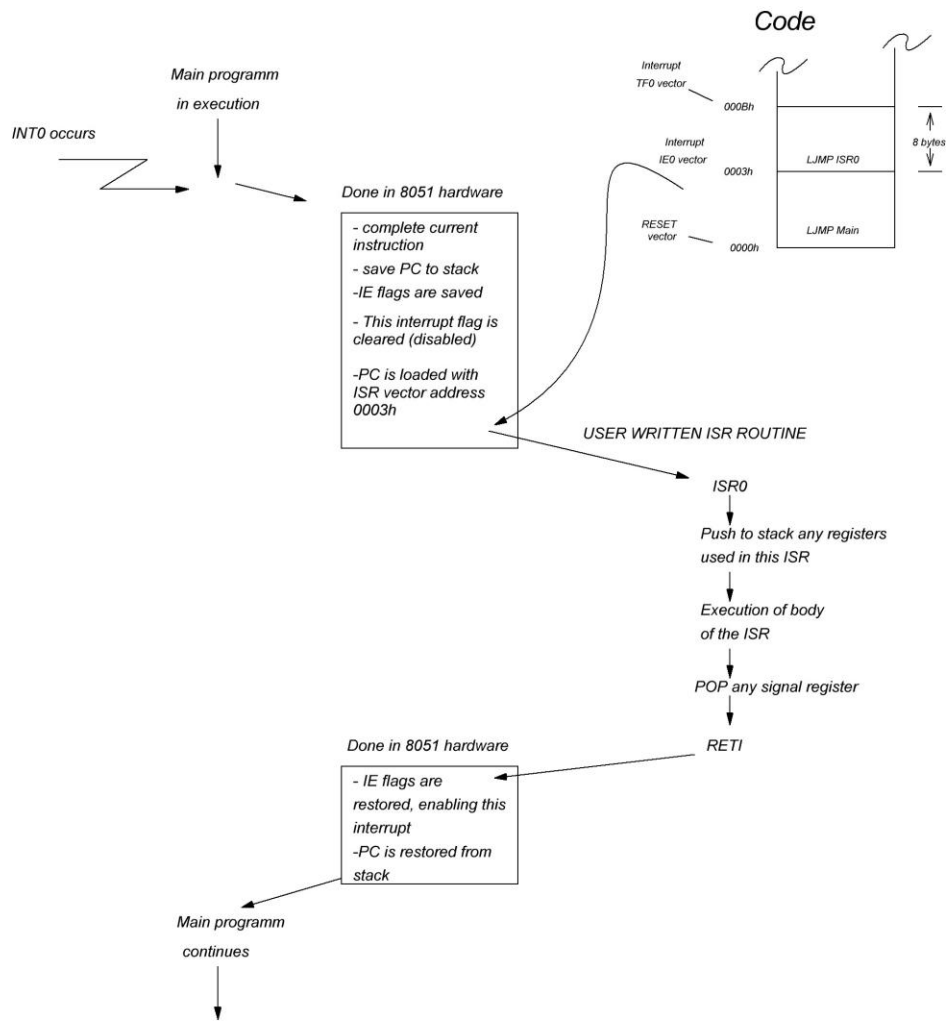


Fig. 5.10. Interrupt operation example

Fig. 5.10 shows the flow of operation when a system is interrupted. In the example it is assumed that some program, say the main program, is executing when the external interrupt *INT0* occurs. The 8051 hardware will automatically complete the current machine level (assembler level) instruction and save the Program Counter to the stack. The *IE* register is also saved to the stack. The *IE0* flag is disabled (cleared) so that another *INT0* interrupt will be inhibited while the current interrupt is being serviced. The Program Counter is now loaded with the vector location 0003h. This vector address is a predefined address for interrupt *INT0* so that the program execution will always trap to this address when an *INT0* interrupt occurs. Other interrupt sources have uniquely defined vector addresses for this purpose. The set of these vector addresses is referred to as the interrupt vector table.

Program execution is now transferred to address location 0003h. In the example a *LJMP* instruction is programmed at this address to cause the pro-

gram to jump to a predefined start address location for the relevant *ISR* (Interrupt Service Routine) routine. The *ISR* routine is a user written routine, which defines what action is to occur following the interrupt event. It is good practice to save (*PUSH*) to the stack any registers used during the *ISR* routine and to restore (*POP*) these registers at the end of the *ISR* routine, thus preserving the registers' contents, justlike a register is preserved within a subroutine program. The last instruction in the *ISR* routine is a *RETI* (RE-Turn from Interrupt) instruction and this instruction causes the 8051 to restore the *IE* register values, enable the *INT0* flag, and restore the Program Counter contents from the stack [10].

Since the Program Counter now contains the address of the next instruction which was to be executed before the *INT0* interrupt occurred, the main program continues as if it had never being interrupted. Thus only the temporal behavior of the interrupted program has been affected by the interrupt; the logic of the program has not been otherwise affected.

An individual interrupt source can be assigned to one of two priority levels. The Interrupt Priority, *IP*, register is an *SFR* register used to program the priority level for each interrupt source. A logic 1 specifies the high priority level while a logic 0 specifies the low priority level.

Table 5.5. Interrupt priority register

Symbol	Number of bit	Description
-	IP.7	Reserved
-	IP.6	
-	IP.5	
PS	IP.4	UART priority bit. Set/reset by software to set highest/lowest priority of UART interrupt.
PT1	IP.3	Timer 1 priority bit. Set/reset by software to set highest/lowest priority of Timer 1 interrupt.
PX1	IP.2	External interrupt 1 priority bit. Set/reset by software to set highest/lowest priority of external interrupt 1.
PT0	IP.1	Timer 0 priority bit. Set/reset by software to set highest/lowest priority of Timer 0 interrupt.
PX0	IP.0	External interrupt 0 priority bit. Set/reset by software to set highest/lowest priority of external interrupt 0.

If two interrupt requests, at different priority levels, arrive at the same time then the high priority interrupt is serviced first. If two, or more, interrupt requests at the same priority level arrive at the same time then the interrupt to

be serviced is selected in the order shown below. Note, this order is used only to resolve simultaneous requests. Once an interrupt service begins it cannot be interrupted by another interrupt at the same priority level.

8051 microcontroller is reset by supplying the input signal **RST** 1. For reliable resetting of 8051 this signal should be deducted in **RESET** pin at least two machine cycles (24 periods of resonator). Quasi-bidirectional Buffers output pins **ALE** and **PSEN** are thus in input mode. Under the influence of the **RST** the contents of the registers **PC**, **ACC**, **B**, **PSW**, **DPTR**, **TMOD**, **TCON**, **T/C0**, **T/C1**, **IE**, **IP** and **SCON** are reset, in **PCON** register only significant bit is reset and in stack pointer register is loaded code 07h, and the ports **P0** - **P3** - codes 0FFh. Status register **SBUF** indefinitely. **RST** has no effect on the cells of internal data memory. When the power is turned on Vcc, the contents of internal data memory is uncertain, except for the return operation from low energy mode.

Fig. 5.11 shows an automatic signal **RST** at power switch-on. The time required to fully charge the capacitance of the resonator provides a confident start and his work for more than two machine cycles.

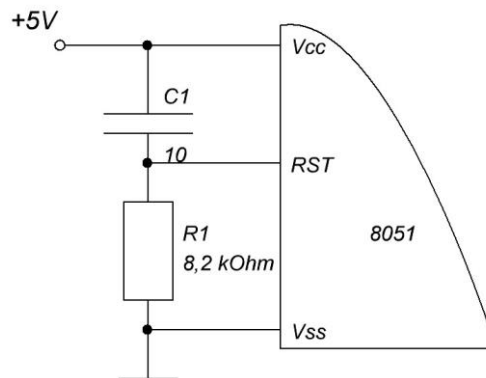


Fig. 5.11. RST signal

5.3. TIMERS AND COUNTERS

The 8051 has two internal sixteen bit hardware Timer/Counters. Each Timer/Counter can be configured in various modes, typically based on 8-bit or 16-bit operation. The 8052 product has an additional (third) Timer/Counter.

Figure 5.12 provides us with a brief refresher on what a hardware counter looks like. This is a circuit for a simple 3-bit counter which counts from 0 to 7 and then overflows, setting the overflow flag. A 3-bit counter would not be very useful in a microcomputer so it is more typical to find 8-bit and 16-bit counter circuits.

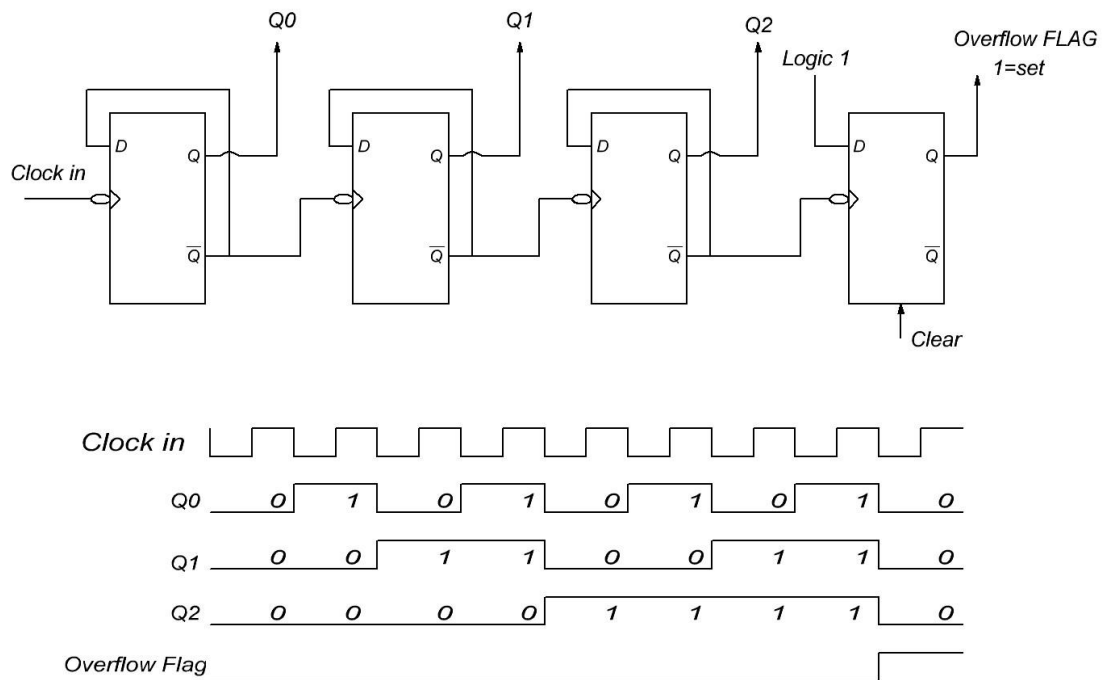


Fig. 5.11. Timer operation

There are two registers for control of timers in 8051 microcontroller - ***TMOD*** and ***TCON*** (Tables 5.6 and 5.7) [9, 10, 11].

Table 5.6. ***TMOD*** register

Symbol	Number of bit	Description
GATE	TMOD.7 for T/C1	Lock Control. If the bit is set, the timer / counter "x" is allowed as long as the input <i>INTx</i> high and <i>TRx</i> control bit is set. If the bit is clear that <i>T/C</i> is permitted, as <i>TRx</i> control bit is set.
	TMOD.3 for T/C0	
C/ \overline{T}	TMOD.6 for T/C1	Mode Select bit timer or an event counter. If the bit is cleared, the timer works by an internal clock signals. If 1, the counter works on the external signal input <i>Tx</i> .
	TMOD.2 for T/C0	
M1	TMOD.5 for T/C1	Operation modes
	TMOD.1 for T/C0	
M0	TMOD.4 for T/C1	
	TMOD.0 for	

		T/C0	
Timers operation modes			
M1	M0	Режим работы	
0	0	13-bit mode	
0	1	16-bit mode. <i>THx</i> and <i>TLx</i> connected in series.	
1	0	8-bit mode with auto reload feature. THx store auto reload value and TLx is a timer register.	
1	1	Ignore for now	

Table 5.7. *TCON* register

Symbol	Number of bit	Description
TF1	TCON.7	Timer 1 overflow flag. Set by hardware when timer/counter overflow. Reset when the interrupt service based apparatus.
TR1	TCON.6	Control bit timer 1. Set/cleared by software to start/stop.
TF0	TCON.5	Timer 0 overflow flag. Set by hardware when timer/counter overflow. Reset when the interrupt service based apparatus.
TR0	TCON.4	Control bit timer 0. Set/cleared by software to start/stop.
IE1	TCON.3	Interrupt 1 edge flag. Set by hardware, on the falling edge of external signal <i>INT1</i> . Reset when the interrupt service in the detection on the falling edge, the interrupt level flag is cleared by removing the <i>INT1</i> .
IT1	TCON.2	Bit of control of the type of interrupt 1. Set / cleared by software to specify the query <i>INT1</i> .
IE0	TCON.1	Interrupt 0 edge flag. Set by hardware, on the falling edge of external signal <i>INT0</i> . Reset when the interrupt service in the detection on the falling edge, the interrupt level flag is cleared by removing the <i>INT0</i> .
IT0	TCON.0	Bit of control of the type of interrupt 0. Set / cleared by software to specify the query <i>INT0</i> .

8-bit counter operation

First let us consider a simple 8-bit counter. Since this is a modulo-8 set up we are concerned with 256 numbers in the range 0 to 255 ($2^8=256$). The counter will count in a continuous sequence as follows [10]:

<i>Hex</i>	<i>Binary</i>	<i>Decimal</i>	
00h	00000000	0	
01h	00000001	1	
02h	00000010	2	
.	.	.	
.	.	.	
FEh	11111110	254	
FFh	11111111	255	
00h	00000000	0	→ here the counter overflows to zero
01h	00000001	1	
etc.			
etc.			

Fig. 5.12. The 8-bit counter operation

We will use Timer/Counter 1 in our examples below.

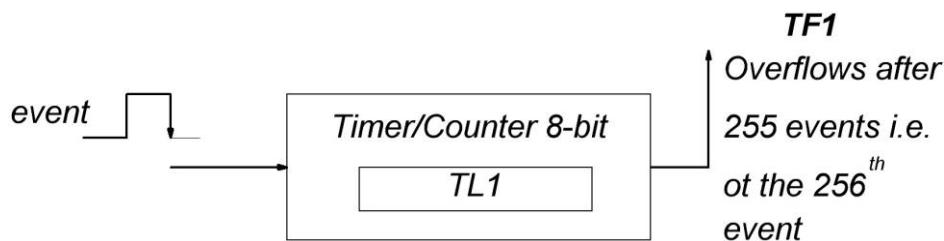


Fig. 5.13. The 8-bit counter operation

Supposing we were to initialize this Timer/Counter with a number, say 252, then the counter would overflow after just four event pulses, i.e.:

FC	11111100	252	counter is initialised at 252
FD	11111101	253	
FE	11111110	254	
FF	11111111	255	
00	00000000	0	→ here the counter overflows

Fig. 5.14. The 8-bit counter operation

An 8-bit counter can count 255 events before overflow, and overflows on the 256 event. When initialized with a predefined value of say 252 it overflows after counting just four events. Thus the number of events to be counted can be programmed by pre-loading the counter with a given number value.

8-bit timer operation

The 8051 internally divides the processor clock by 12. If a 12 MHz. processor clock is used then a 1 MHz. instruction rate clock, or a pulse once every microsecond, is realized internally within the chip. If this 1 microsecond pulse is connected to a Timer/Counter input, in place of an event input, then the Timer/Counter becomes a timer which can delay by up to 255 microseconds. There is a clear difference between a timer and a counter. The counter will count events, up to 255 events before overflow, and the timer will count time pulses, thus creating delays up to 255 microseconds in our example [10].

To be precise we would refer to the counter as an event counter and we would refer to the timer as an interval timer.

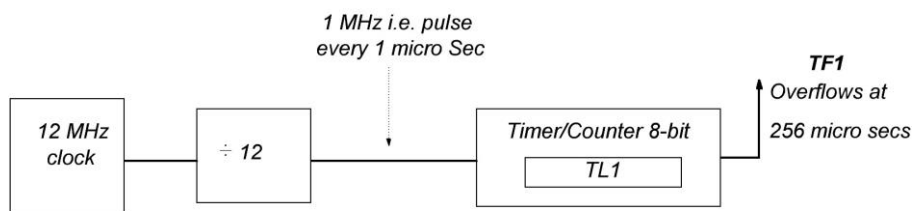


Fig. 5.15. The 8-bit timer operation

If the timer is initialized to zero it will count 256 microseconds before overflow. If the timer is initialized to a value of 252, for example, it will count just 4 microseconds before overflow. Thus this timer is programmable between 1 microsecond and 256 microseconds.

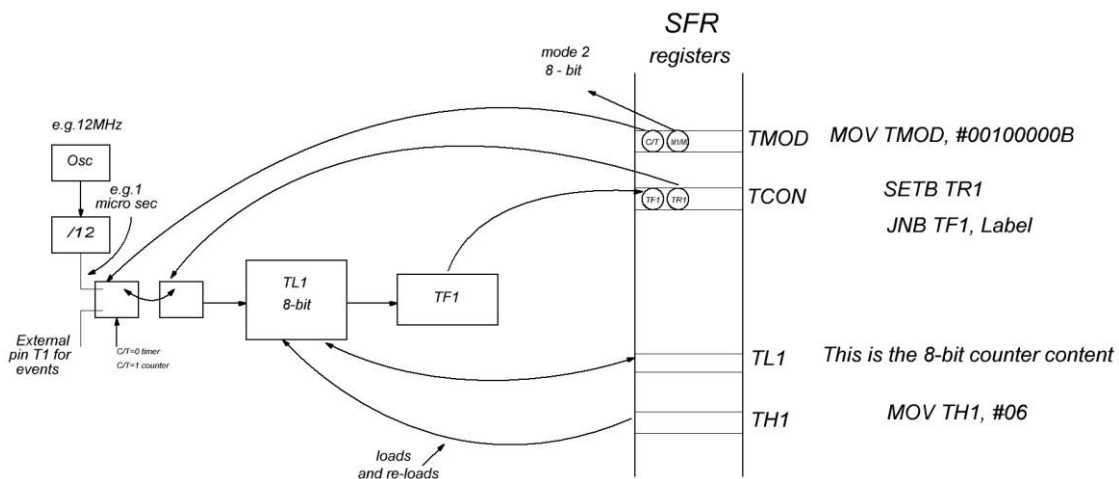


Fig. 5.16. The programmer's view of Timer 1, Mode 2

The 16-bit timer operation

When the Timer/Counter is configured for mode 1 operation it operates in 16 bit mode. Since this is a modulo-16 set up we are concerned with

65,536 numbers in the range 0 to 65535 ($2^{16} = 65536$). Consider a 16 bit Timer/Counter as shown below, which will count in the sequence as follows:

Hex	Binary	Decimal
0000h	0000000000000000	0
0001h	0000000000000001	1
0002h	0000000000000010	2
.	.	.
FFFEh	1111111111111110	65,534
FFFFh	1111111111111111	65,535
0000h	0000000000000000	0

————— here it overflows to zero

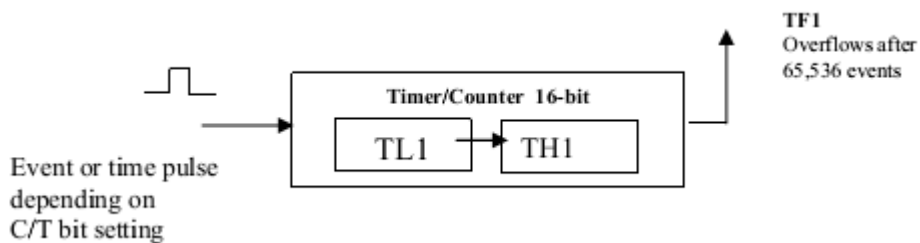


Fig. 5.17. 16-bit Timer operation

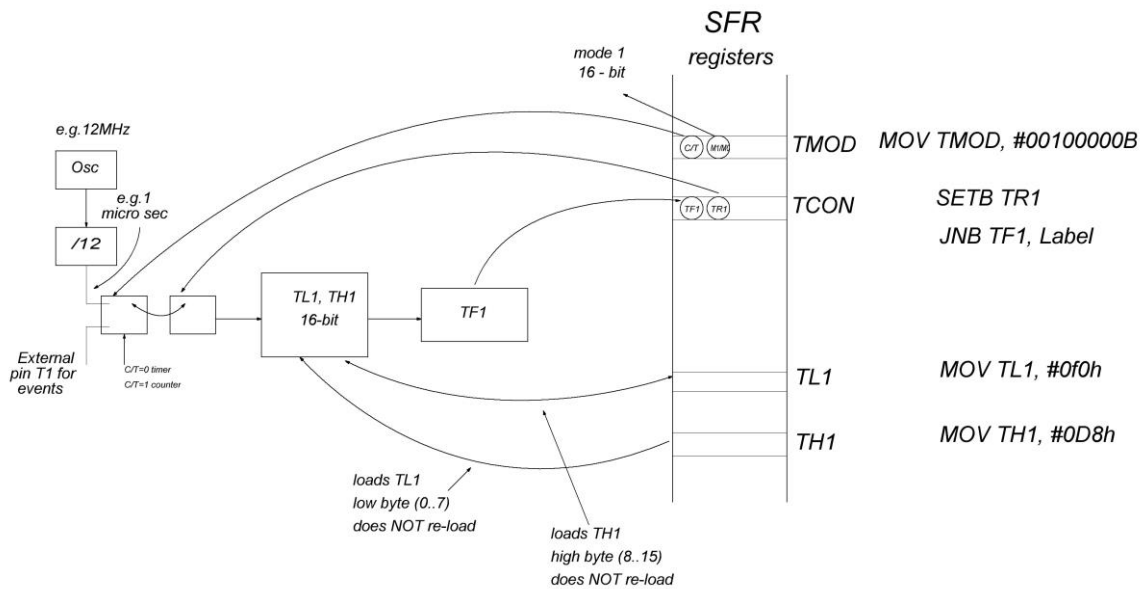


Fig. 5.18. The programmer's view of Timer 1, Mode 1, 16-bit

Now we have a 16-bit Timer/Counter and we can preload it with a sixteen bit number so as to cause a delay from between 1 to 65535 microseconds

(65.535 milliseconds), or in counter mode it can count between 1 and 65535 events. To preload the Timer/Counter value simply write the most significant byte into the TH1 register and the least significant byte into the TL1 register. The 16-bit counter is not automatically reloaded following an overflow and such reloading must be explicitly programmed. We will see this in some examples below [9, 10, 11].

CONCLUSION

This textbook focuses on the working principles of microprocessors using Intel 8080 as an example. This kind of processor is rather simple; however, it allows to describe the basic principles used in all modern processing units. The differences between the microprocessors and microcontrollers are demonstrated with a typical 8051 microcontroller architecture. 8051 microcontroller peripherals, their operation modes and tuning capabilities are also considered.

The author is grateful to Galina Vorobyova and Tatiana Evtushenko for their help.

BIBLIOGRAPHY

1. Gashkov S.B. Systemy schisleniya I ih primeneniya. –M: Izdatelstvo Moskovskogo tsentra nepreryvnogo matematicheskogo obrazovaniya, 2004. – 52 p.
2. Liventsov S.N., Vilnin A.D., Goryunov A.G. Osnovy microprocessor-noi tehniki. –Tomsk: Izdatelstvo TPU, 2007. – 118 p.
3. Emmanuel C. Ifeakor, Barrie W. Jervis. Digital Signal processing. – US: Addison-Wesley Publishers Ltd, 1993. – 760 p.
4. Novikov Yu.V., Ocnovy microprocessornoi tehniki. – M.: Internet-universitet informatsionih tehnologiy, BINOM, 2009. – 357 p.
5. Naresh K. Sinha, Microprocessor-based control systems. Netherlands: Kluwer academic publishers, 1986. – 77 p.
6. Sunil Mathur Microprocessor 8085 and its interfacing. – New delhi: PHI Learning Private Limited, 2011. – 868 p.
7. Intel 8080/8085 Assembly language programming. – USA: Intel Corporation, 1975. – 224 p.
8. Kenneth J. Ayala. The 8051 Microcontroller. Architecture, programming and applications. – USA: West publishing company, 1991. – 255 p.
9. Donal Heffernan, 8051 Tutorial. – USA: University of Limerick, 2002. – 116 p.
10. Silicon Laboratories [Electronic resource] / Silicon laboratories Corporation –Electronic data. – September 2013 – Regime: <http://www.silabs.com/Support%20Documents/TechnicalDocs>, free. – — English language.
11. Silicon Laboratories [Electronic resource] / Silicon laboratories Corporation –8051 Instruction set. – September 2013 – Regime: https://www.silabs.com/Support%20Documents/Software/8051_Instruction_Set.pdf, free. – — English language.

CONTENTSS

Introduction	3
Part 1. Positional notations	4
1.5. Conversion from base-10 to others positional notations	6
1.6. Signed binary numbers	6
1.7. Fixed-point numbers	3
1.8. Floating point numbers	8
Part 2. Microprocessors and microprocessor systems	11
2.1. Basic concepts	11
2.2. Classification of microprocessors	13
2.3. The characteristics of microprocessors	15
2.4. The microprocessor architecture	17
2.5. Bus organized structure	19
2.6. The structure of microprocessor system	23
Part 3. Intel 8080 microprocessor	26
3.1. Arithmetic logic unit (ALU)	29
3.2. Microprocessor registers	29
3.3. Accumulator	30
3.4. Program counter (PC)	30
3.5. Address register	32
3.6. Instruction register (IR)	32
3.7. Flag register (F). Status register.	34
3.8. Buffer register of ALU	37
3.9. General purpose registers, register pairs	37
3.10. Stack pointer (SP)	38
3.11. Control circuit	41
3.12. Status word register	41
3.13. Timing and synchronization of the microprocessor system	43
3.14. Operation of microprocessor	45
3.15. Reactions of microprocessor on signal <i>READY</i>	51
3.16. Reactions of microprocessor on signal <i>HOLD</i>	52

3.17. Reactions of microprocessor on command <i>HLT</i>	54
3.18. Reactions of microprocessor on signal <i>INT</i>	55
Part 4. Assembly language of <i>Intel</i> 8080	58
4.1. Data and instructions formats	60
4.2. Memory addressing	61
4.3. <i>Intel</i> 8080 instructions: description and application	63
Part 5. 8051 microcontrollers	98
5.1. The 8051 Architecture	100
5.2. Interrupts of 8051	112
5.3. Timers and counters	116
Conclusion	125
Bibliography	126

Educational Edition

Национальный исследовательский
Томский политехнический университет

ТОРГАЕВ СТАНИСЛАВ НИКОЛАЕВИЧ

**МИКРОПРОЦЕССОРНЫЕ СИСТЕМЫ УПРАВЛЕНИЯ И
КОНТРОЛЯ. ЧАСТЬ 1. МИКРОПРОЦЕССОР INTEL 8080
И МИКРОКОНТРОЛЛЕР 8051**

Учебное пособие

Издательство Томского политехнического университета, 2013

На английском языке

Published in author's version

Science Editor Doctor of ...,

Professor *Name*

Typesetting *Name*

Cover design *Name*

**Printed in the TPU Publishing House in full accordance
with the quality of the given make up page**

Signed for the press 00.00.2012. Format 60x84/16. Paper "Snegurochka".

Print XEROX. Arbitrary printer's sheet 000. Publisher's signature 000.

Order XXX. Size of print run XXX.



Tomsk Polytechnic University
Quality management system
of Tomsk Polytechnic University was certified by
NATIONAL QUALITY ASSURANCE on BS EN ISO 9001:2008



TPU PUBLISHING HOUSE. 30, Lenina Ave, Tomsk, 634050, Russia

Tel/fax: +7 (3822) 56-35-35, www.tpu.ru

