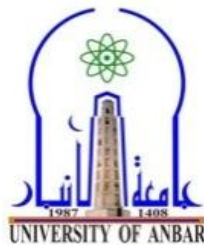




Course Weekly Outline

Course Name: Microprocessor

Course Instructor					
E-mail					
Title					
Course Coordinator					
Course Objective					
Course Description					
Textbook					
References					
Course Assessments	TermTests	Laboratory	Quizzes	Project	Final Exam
General Notes					



First Course Weekly Outline

Week	Date	Topics Covered				Notes
1		Introduction Computer organization				
2		Historical development for computers				
3		Computer Levels				
4		Data Representation in Computer Systems.				
5		Signed Integer Representation				
6		Floating Point Representation				
7		Introduction to a Simple Computer				
8		CPU Functions				
9		Mid Examination				
10		Registers, Buses				
11		simple model computer design, Marie				
12		Instruction Processing				
13		Assembler				
14		Control Unit, Real World Architecture				
15		Final Examination.				
	Term Tests		Laboratory	Quizzes	Project	Final Exam
	(30%)		-----	(10 %)	(%)	(60%)

Instructor Signature:

Dean Signature:



Second Course Weekly Outline

Week	Date	Topics Covered				Notes
1		Instruction Set Architecture				
2		Instruction Format and types				
3		Addressing modes:1-3				
4		Addressing modes:3-7				
5		Memory system, Introduction				
6		Components of memory system				
7		The memory Hierarchy				
8		Cache Memory				
9		Mid Examination				
10		Cache Organization				
11		Replacements Algorithms				
12		Write Strategies				
13		Virtual Memory				
14		Virtual Memory				
15		Final Examination.				
		Term Tests	Laboratory	Quizzes	Project	Final Exam
		(30%)	-----	(10 %)	(%)	(60%)

Instructor Signature:

Dean Signature:

1.1 Evolution from 8080/8085 to 8086

In 1978, Intel Corporation introduced a 16-bit microprocessor called the 8086. This processor was a major improvement over the previous generation 8080/8085 series Intel microprocessors in several ways:

First, the 8086's capacity of 1 megabyte of memory exceeded the 8080/8085's capability of handling a maximum of 64K bytes of memory.

Second, the 8080/8085 was an 8-bit system, meaning that the microprocessor could work on only 8 bits of data at a time. Data larger than 8 bits had to be broken into 8-bit pieces to be processed by the CPU. In contrast, the 8086 is a 16-bit microprocessor.

Third, the 8086 was a pipelined processor, as opposed to the non pipelined 8080/8085. In a system with pipelining, the data and address buses are busy transferring data while the CPU is processing information, thereby increasing the effective processing power of the microprocessor.

Evolution from 8086 to 8088

The 8086 is a microprocessor with a 16-bit data bus internally and externally, meaning that all registers are 16 bits wide and there is a 16-bit data bus to transfer data in and out of the CPU.

Although the introduction of the 8086 marked a great advancement over the previous generation of microprocessors, there was still some resistance in using the 16-bit external data bus since at that time all peripherals were designed around an 8-bit microprocessor.

In addition, a printed circuit board with a 16-bit data bus was much more expensive. Therefore, Intel came out with the 8088 version. It is identical to the 8086 as far as programming is concerned, but externally it has an 8-bit data bus instead of a 16-bit bus. It has the same memory capacity, 1 megabyte.

Other microprocessors: the 80286, 80386, and 80486

With a major victory behind Intel and a need from PC users for a more powerful microprocessor, Intel introduced the **80286** in 1982. Its features included 16-bit internal and external data buses; 24 address lines, which give 16 megabytes of memory ($2^{24} = 16$ megabytes); and most significantly, virtual memory.

The 80286 can operate in one of two modes: **real mode** or **protected mode**. **Real mode** is simply a faster 8088/8086 with the same maximum of 1 megabyte of memory. **Protected mode** allows for 16M of memory but is also capable of protecting the operating system and programs from accidental or deliberate destruction by a user, a feature that is absent in the single-user 8088/8086.

Virtual memory is a way of fooling the microprocessor into thinking that it has access to an almost unlimited amount of memory by swapping data between disk storage and RAM.

With users demanding even more powerful systems, in 1985 Intel introduced the **80386** (sometimes called 80386DX), internally and externally a 32-bit microprocessor with a 32-bit address bus. It is capable of handling physical memory of up to 4 gigabytes (2^{32}). Virtual memory was increased to 64 terabytes (2^{46}). All microprocessors discussed so far were general-purpose microprocessors and could not handle mathematical calculations rapidly. For this reason, Intel introduced numeric data processing chips, called math coprocessors, such as the 8087, 80287, and 80387.

Later Intel introduced the **386SX**, which is internally identical to the 80386 but has a 16-bit external data bus and a 24-bit address bus which gives a capacity of 16 megabytes (2^{24}) of memory. This makes the 386SX system much cheaper. With the introduction of the **80486** in 1989, Intel put a greatly enhanced version of the 80386 and the math coprocessor on a single chip plus additional features such as cache memory. Cache memory is static RAM with a very fast access time. Table 1-1 summarizes the evolution of Intel's microprocessors. It must be noted that all programs written for the 8086/88 will run on 286, 386, and 486 computers.

Table 1-1: Evolution of Intel's Microprocessors

Product	8080	8085	8086	8088	80286	80386	80486
Year introduced	1974	1976	1978	1979	1982	1985	1989
Clock rate (MHz)	2 - 3	3 - 8	5 - 10	5 - 8	6 - 16	16 - 33	25 - 50
No. transistors	4500	6500	29,000	29,000	130,000	275,000	1.2 million
Physical memory	64K	64K	1M	1M	16M	4G	4G
Internal data bus	8	8	16	16	16	32	32
External data bus	8	8	16	8	16	32	32
Address bus	16	16	20	20	24	32	32
Data type (bits)	8	8	8, 16	8, 16	8, 16	8, 16, 32	8, 16, 32

Notes:

1. The 80386SX architecture is the same as the 80386 except that the external data bus is 16 bits in the SX as opposed to 32 bits, and the address bus is 24 bits instead of 32; therefore, physical memory is 16MB.
2. Clock rates range from the rates when the product was introduced to current rates; some rates have risen during this time.

1.2 INSIDE THE 8088/8086

In this section we explore concepts important to the internal operation of the 8088/86, such as pipelining and registers. See the block diagram in Figure 1.1.

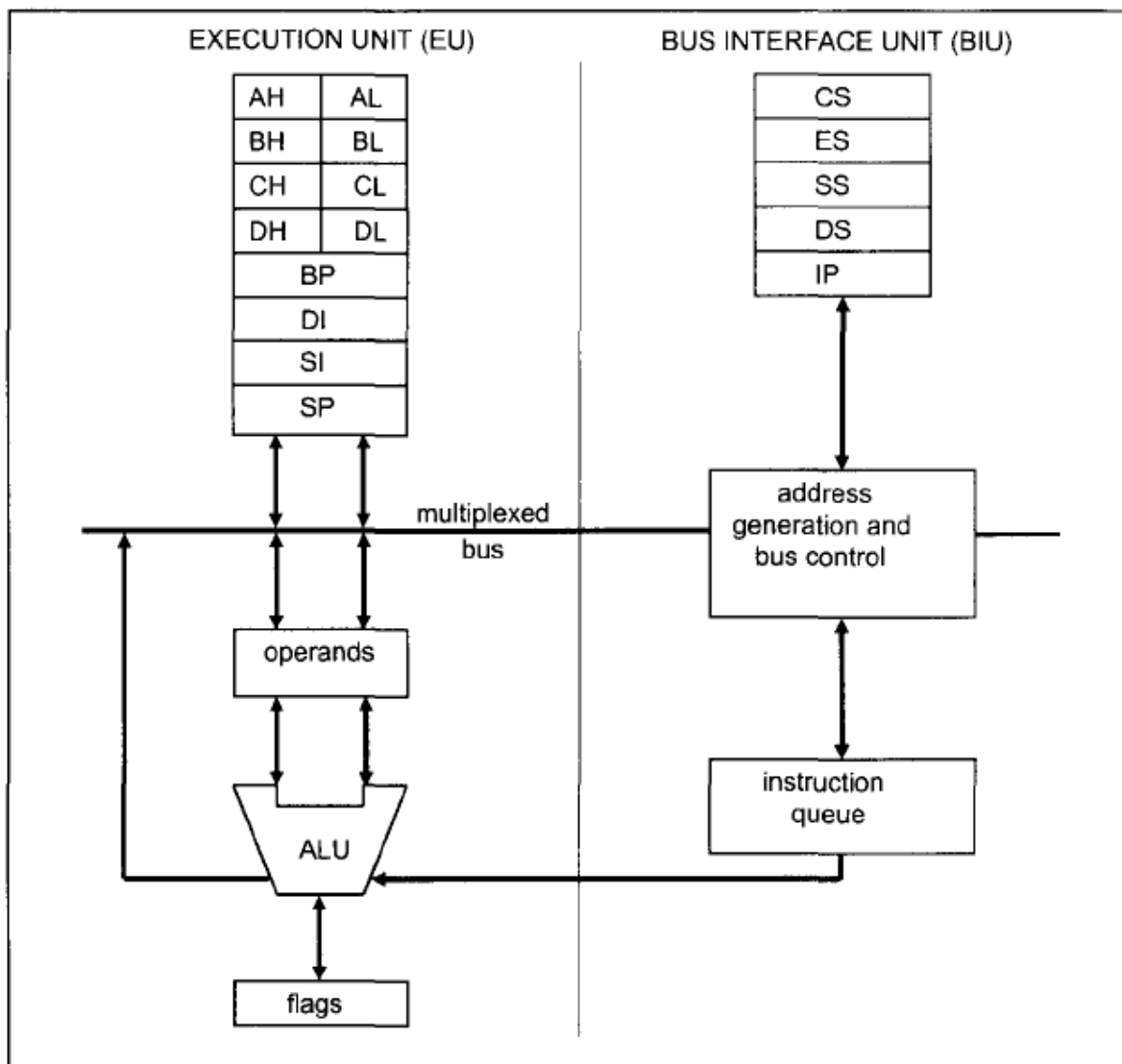


Figure (1.1) Internal block diagram of the 8088/86 CPU

Pipelining

There are two ways to make the CPU process information faster: increase the working frequency or change the internal architecture of the CPU.

The first option is technology dependent, meaning that the designer must use whatever technology is available at the time, with consideration for cost. The technology and materials used in making ICs (integrated circuits) determine the working frequency, power consumption, and the number of transistors packed into a single-chip microprocessor.

The second option for improving the processing power of the CPU has to do with the internal working of the CPU. In the 8085 microprocessor, the CPU could either fetch or execute at a given time. In other words, the CPU had to fetch an instruction from memory, then execute it and then fetch again, execute it, and so on. The idea of pipelining in its simplest form is to allow the CPU to fetch and execute at the same time as shown in Figure 1-2. It is important to point out that Figure 1.2 is not meant to imply that the amount of time for fetch and execute are equal.

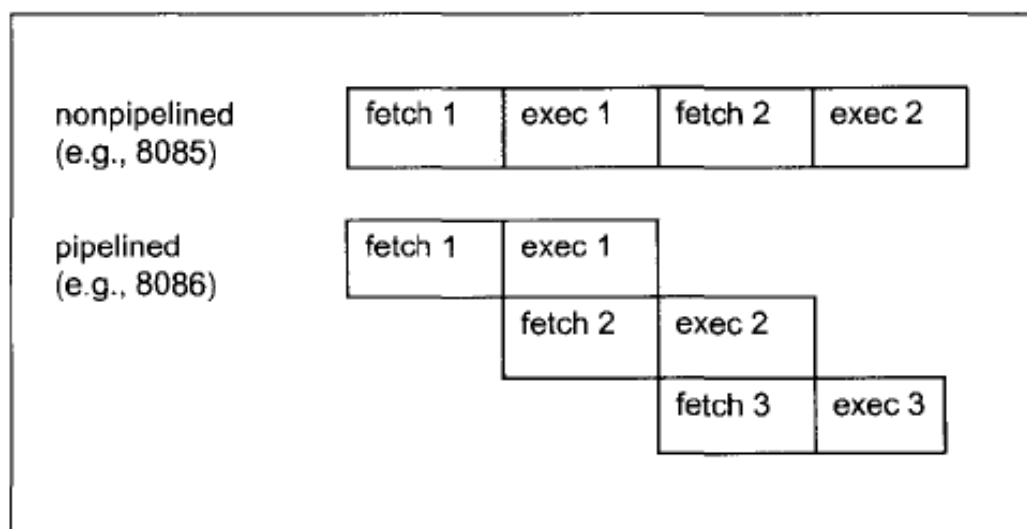


Figure (1.2) Pipelined vs. non pipelined Execution

Intel implemented the concept of pipelining in the 8088/86 by splitting the internal structure of the microprocessor into two sections: the **execution unit (EU)** and the **bus interface unit (BIU)**.

These two sections work simultaneously. The BIU accesses memory and peripherals while the EU executes instructions previously fetched.

This works only if the BIU keeps ahead of the EU; thus the BIU of the 8088/86 has a buffer, or queue (see Figure 1.1). The buffer is 4 bytes long in the 8088 and 6 bytes in the 8086. If any instruction takes too long to execute, the queue is filled to its maximum capacity and the buses will sit idle. The BIU fetches a new instruction whenever the queue has room for 2 bytes in the 6-byte 8086 queue, and for 1 byte in the 4-byte 8088 queue. In some circumstances, the microprocessor must flush out the queue. For example, when a jump instruction is executed, the BIU starts to fetch information from the new location in memory and information in the queue that was fetched previously is discarded. In this situation the EU must wait until the BIU fetches the new instruction. This is referred to in computer

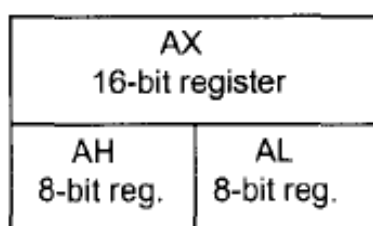
science terminology as a branch penalty. In a pipelined CPU, this means that too much jumping around reduces the efficiency of a program.

Pipelining in the 8088/86 has two stages: fetch and execute, but in more powerful computers pipelining can have many stages. The concept of pipelining combined with an increased number of data bus pins has, in recent years, led to the design of very powerful microprocessors.

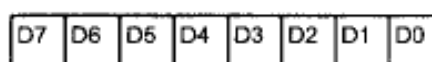
Registers

In the CPU, registers are used to store information temporarily. That information could be one or two bytes of data to be processed or the address of data. The registers of the 8088/86 fall into the six categories outlined in Table 1.2.

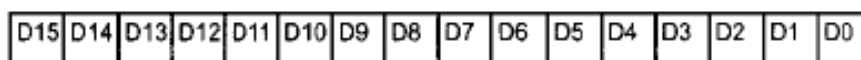
The general-purpose registers in 8088/86 microprocessors can be accessed as either 16-bit or 8-bit registers. All other registers can be accessed only as the full 16 bits. In the 8088/86, data types are either 8 or 16 bits. To access 12-bit data, for example, a 16-bit register must be used with the highest 4 bits set to 0. The bits of a register are numbered in descending order, as shown below.



8-bit register:



16-bit register:



Different registers in the 8088/86 are used for different functions, and since some instructions use only specific registers to perform their tasks, the use of registers will be described in the context of instructions and their application in a given program. The first letter of each general register indicates its use. AX is used for the accumulator, BX as a base addressing register, CX is used as a counter in loop operations, and DX is used to point to data in I/O operations.

Table 1-2: Registers of the 8086/286 by Category

Category	Bits	Register Names
General	16	AX, BX, CX, DX
	8	AH, AL, BH, BL, CH, CL, DH, DL
Pointer	16	SP (stack pointer), BP (base pointer)
Index	16	SI (source index), DI (destination index)
Segment	16	CS (code segment), DS (data segment), SS (stack segment), ES (extra segment)
Instruction	16	IP (instruction pointer)
Flag	16	FR (flag register)

Note:

The general registers can be accessed as the full 16 bits (such as AX), or as the high byte only (AH) or low byte only (AL).

1.3 Assembly language programming

An Assembly language program consists of, among other things, a series of lines of Assembly language instructions. An Assembly language instruction consists of a mnemonic, optionally followed by one or two operands. The **operands** are the data items being manipulated, and the **mnemonics** are the commands to the CPU, telling it what to do with those items.

We introduce Assembly language programming with two widely used instructions: the *move* and *add* instructions.

MOV instruction

Simply stated, the MOV instruction copies data from one location to another. It has the following format:

MOV destination, source ; copy source operand to destination

This instruction tells the CPU to move (in reality, copy) the source operand to the destination operand.

For example, the instruction "MOV DX,CX" copies the contents of register CX to register DX. After this instruction is executed, register DX will have the same value as register CX. The MOV instruction does not affect the source operand.

The following program first loads CL with value 55H, then moves this value around to various registers inside the CPU.

```
MOV    CL,55H      ;move 55H into register CL
MOV    DL,CL       ;copy the contents of CL into DL (now DL=CL=55H)
MOV    AH,DL       ;copy the contents of DL into AH (now AH=DL=55H)
MOV    AL,AH       ;copy the contents of AH into AL (now AL=AH=55H)
MOV    BH,CL       ;copy the contents of CL into BH (now BH=CL=55H)
MOV    CH,BH       ;copy the contents of BH into CH (now CH=BH=55H)
```

The use of 16-bit registers is demonstrated below.

```
MOV    CX,468FH    ;move 468FH into CX (now CH=46,CL=8F)
MOV    AX,CX       ;copy contents of CX to AX (now AX=CX=468FH)
MOV    DX,AX       ;copy contents of AX to DX (now DX=AX=468FH)
MOV    BX,DX       ;copy contents of DX to BX (now BX=DX=468FH)
MOV    DI,BX       ;now DI=BX=468FH
MOV    SI,DI       ;now SI=DI=468FH
MOV    DS,SI       ;now DS=SI=468FH
MOV    BP,DI       ;now BP=DI=468FH
```

In the 8086 CPU, data can be moved among all the registers shown in Table (1-2) (except the flag register) as long as the source and destination registers match in size. such as "MOV AL,DX" will cause an error, since one cannot move the contents of a 16-bit register into an 8-bit register. The exception of the flag register means that there is no such instruction as "MOV FR,AX". Loading the flag register is done through other means, discussed in later chapters.

If data can be moved among all registers including the segment registers, can data be moved directly into all registers?

The answer is no. Data can be moved directly into non segment registers only, using the May instruction. For example, look at the following instructions to see which are legal and which are illegal.

```
MOV    AX,58FCH    ;move 58FCH into AX    (LEGAL)
MOV    DX,6678H    ;move 6678H into DX    (LEGAL)
MOV    SI,924BH    ;move 924B into SI     (LEGAL)
MOV    BP,2459H    ;move 2459H into BP    (LEGAL)
MOV    DS,2341H    ;move 2341H into DS    (ILLEGAL)
MOV    CX,8876H    ;move 8876H into CX    (LEGAL)
MOV    CS,3F47H    ;move 3F47H into CS    (ILLEGAL)
MOV    BH,99H      ;move 99H into BH      (LEGAL)
```

From the discussion above, note the following three points:

1. Values cannot be loaded directly into any segment register (CS, OS, ES, or SS). To load a value into a segment register, first load it to a non segment register and then move it to the segment register, as shown next.

```
MOV  AX,2345H    ;load 2345H into AX
MOV  DS,AX        ;then load the value of AX into DS

MOV  DI,1400H     ;load 1400H into DI
MOV  ES,DI        ;then move it into ES, now ES=DI=1400
```

2. If a value less than FFH is moved into a 16-bit register, the rest of the bits are assumed to be all zeros.

For example, in "MOV BX,5" the result will be BX ~ 0005; that is, BH ~ 00 and BL ~ 05.

3. Moving a value that is too large into a register will cause an error.

```
MOV  BL,7F2H      ;ILLEGAL: 7F2H is larger than 8 bits
MOV  AX,2FE456H   ;ILLEGAL: the value is larger than AX
```

ADD instruction:

The ADD instruction has the following format:

ADD destination, source ; ADD the source operand to the destination

The ADD instruction tells the CPU to add the source and the destination operands and put the result in the destination. To add two numbers such as 25H and 34H, each can be moved to a register and then added together

```
MOV  AL,25H       ;move 25 into AL
MOV  BL,34H       ;move 34 into BL
ADD  AL,BL        ;AL = AL + BL
```

Executing the program above results in AL ~ 59H (25H + 34H ~ 59H) and BL ~ 34H. Notice that the contents of BL do not change. The program above can be written in many ways, depending on the registers used. Another way might be:

Chapter 1: THE 80x86 MICROPROCESSOR

```
MOV  DH,25H      ;move 25 into DH
MOV  CL,34H      ;move 34 into CL
ADD  DH,CL       ;add CL to DH: DH = DH + CL
```

The program above results in DH = 59H and CL = 34H. There are always many ways to write the same program. One question that might come to mind after looking at the program above is whether it is necessary to move both data items into registers before adding them together. The answer is no, it is not necessary.

Look at the following variation of the same program:

```
MOV  DH,25H      ;load one operand into DH
ADD  DH,34H      ;add the second operand to DH
```

In the case above, while one register contained one value, the second value followed the instruction as an operand. This is called an immediate operand. The examples shown so far for the ADD and MOV instructions show that the source operand can be either a register or **immediate data**. In the examples above, the destination operand has always been a register.

The largest number that an 8-bit register can hold is FFH. To use numbers larger than FFH (255 decimal), 16-bit registers such as AX, BX, CX, or DX must be used.

For example, to add two numbers such as 34EH and 6A5H, the following program can be used:

```
MOV  AX,34EH      ;move 34EH into AX
MOV  DX,6A5H      ;move 6A5H into DX
ADD  DX,AX        ;add AX to DX: DX = DX + AX
```

Running the program above gives DX = 9F3H (34E + 6A5 = 9F3) and AX = 34E. Again, any 16-bit non segment registers could have been used to perform the action above:

```
MOV  CX,34EH      ;load 34EH into CX
ADD  CX,6A5H      ;add 6A5H to CX (now CX=9F3H)
```

The general-purpose registers are typically used in arithmetic operations. Register AX is sometimes referred to as the accumulator.

1.4 INTRODUCTION TO PROGRAM SEGMENTS

A typical assembly language program consists of at least three segments: a **code segment**, a **data segment**, and a **stack segment**.

The **code segment** contains the Assembly language instructions that perform the tasks that the program was designed to accomplish. The **data segment** is used to store information (data) that needs to be processed by the instructions in the code segment. The **stack** is used to store information temporarily.

Origin and definition of the segment

A segment is an area of memory that includes up to 64K bytes and begins on an address evenly divisible by 16 (such an address ends in 0H). The segment size of 64K bytes came about because the 8085 microprocessor could address a maximum of 64K bytes of physical memory since it had only 16 pins for the address lines ($2^{16} = 64K$). This limitation was carried into the design of the 8088/86 to ensure compatibility. Whereas in the 8085 there was only 64K bytes of memory for all code, data, and stack information, in the 8088/86 there can be up to 64K bytes of memory assigned to each category. Within an Assembly language program, these categories are called the code segment, data segment, and stack segment. For this reason, the 8088/86 can only handle a maximum of 64K bytes of code and 64K bytes of data and 64K bytes of stack at any given time, although it has a range of 1 Mega byte of memory because of its 20 address pins ($2^{20} = 1$ megabyte). How to move this window of 64K bytes to cover all 1 megabyte of memory is discussed below, after we discuss logical address and physical address

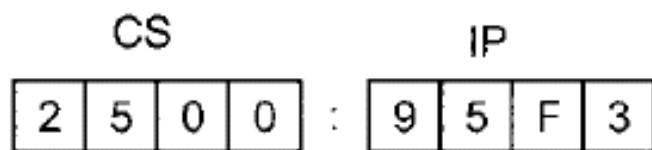
Logical address and physical address

In Intel literature concerning the 8086, there are three types of addresses mentioned frequently: the **physical address**, the **offset address**, and the **logical address**.

The **physical address** is the 20-bit address that is actually put on the address pins of the 8086 microprocessor and decoded by the memory interfacing circuitry. This address can have a range of 00000H to FFFFFH for the 8086 and real-mode 286,386, and 486 CPUs. This is an actual physical location in RAM or ROM within the 1 megabyte memory range.

The **offset address** is a location within a 64K-byte segment range. Therefore, an offset address can range from 0000H to FFFFH. The **logical address** consists of a segment value and an offset address. The differences among these addresses and the process of converting from one to another are best understood in the context of some examples, as shown next.

Code segment



To execute a program, the 8086 fetches the instructions (Op codes and operands) from the code segment. The logical address of an instruction always consists of a CS (code segment) and an IP (instruction pointer), shown in CS:IP format. The physical address for the location of the instruction is generated by shifting the CS left one hex digit and then adding it to the IP. IP contains the offset address. The resulting 20-bit address is called the physical address since it is put on the external physical address bus pins to be decoded by the memory decoding circuitry.

To clarify this important concept, assume values in CS and IP as shown in the diagram. The offset address is contained in IP; in this case it is 95F3H. The logical address is CS:IP, or 2500:95F3H. The physical address will be $25000 + 95F3 = 2E5F3H$. The physical address of an instruction can be calculated as follows:



The microprocessor will retrieve the instruction from memory locations starting at 2E5F3. Since IP can have a minimum value of 0000H and a maximum of FFFFH, the logical address range in this example is 2500:0000 to 2500: FFFF. This means that the lowest memory location of the code segment above will be 25000H ($25000 + 0000$) and the highest memory location will be 34FFFH ($25000 + FFFF$).

What happens if the desired instructions are located beyond these two limits? The answer is that the value of CS must be changed to access those instructions. See Example 1-1.

Example 1-1

If CS = 24F6H and IP = 634AH, show:

- (a) The logical address
- (b) The offset address
- and calculate:
- (c) The physical address
- (d) The lower range
- (e) The upper range of the code segment

Solution:

- (a) 24F6:634A
- (b) 634A
- (c) 2B2AA (24F60 + 634A)
- (d) 24F60 (24F60 + 0000)
- (e) 34F5F (24F60 + FFFF)

Logical vs. physical address in the code segment

In the code segment, CS and IP hold the logical address of the instructions to be executed. The following Assembly language instructions have been assembled (translated into machine code) and stored in memory. The three columns show the logical address of CS:IP the machine code stored at that address and the corresponding Assembly language code. This information can easily be generated by the DEBUG program using the Unassemble command.

Logical address <u>CS:IP</u>	Machine language <u>opcode and operand</u>	Assembly language <u>mnemonics and operand</u>
1132:0100	B057	MOV AL,57
1132:0102	B686	MOV DH,86
1132:0104	B272	MOV DL,72
1132:0106	89D1	MOV CX,DX
1132:0108	88C7	MOV BH,AL
1132:010A	B39F	MOV BL,9F
1132:010C	B420	MOV AH,20
1132:010E	01D0	ADD AX,DX
1132:0110	01D9	ADD CX,BX
1132:0112	05351F	ADD AX,1F35

The program above shows that the byte at address 1132:0 100 contains B0, which is the opcode for moving a value into register AL, and address 1132:0101 contains the operand (in this case 57) to be moved to AL. Therefore, the instruction "MOV

AL, 57" has a machine code of B057, where B0 is the opcode and 57 is the operand. Similarly, the machine code B686 is located in memory locations 1132:0102 and 1132:0103 and represents the opcode and the operand for the instruction "MOV DH,86". The physical address is an actual location within RAM (or even ROM). The following are the physical addresses and the contents of each location for the program above. Remember that it is the physical address that is put on the address bus by the 8086 CPU to be decoded by the memory circuitry.

Data segment

Assume that a program is being written to add 5 bytes of data, such as 25H, 12H, 15H, 1FH, and 2BH, where each byte represents a person's daily overtime pay. One way to add them is as follows:

```
MOV    AL,00H        ;initialize AL
ADD     AL,25H        ;add 25H to AL
ADD     AL,12H        ;add 12H to AL
ADD     AL,15H        ;add 15H to AL
ADD     AL,1FH        ;add 1FH to AL
ADD     AL,2BH        ;add 2BH to AL
```

In the program above, the data and code are mixed together in the instructions. The problem with writing the program this way is that if the data changes, the code must be searched for every place the data is included, and the data retyped. For this reason, the idea arose to set aside an area of memory strictly for data. In 80x86 microprocessors, the area of memory set aside for data is called the data segment. Just as the code segment is associated with CS and IP as its segment register and offset, the data segment uses register DS and an offset value. The following demonstrates how data can be stored in the data segment and the program rewritten so that it can be used for any set of data. Assume that the offset for the data segment begins at 200H. The data is placed in memory locations:

```
DS:0200 = 25
DS:0201 = 12
DS:0202 = 15
DS:0203 = 1F
DS:0204 = 2B
```

and the program can be rewritten as follows:

```
MOV     AL,0          ;clear AL
ADD     AL,[0200]      ;add the contents of DS:200 to AL
ADD     AL,[0201]      ;add the contents of DS:201 to AL
ADD     AL,[0202]      ;add the contents of DS:202 to AL
ADD     AL,[0203]      ;add the contents of DS:203 to AL
ADD     AL,[0204]      ;add the contents of DS:204 to AL
```


Chapter 1: THE 80x86 MICROPROCESSOR

Notice: The 8086/88 allows only the use of registers BX, SI, and DI as offset registers for the data segment. In other words, while CS uses only the IP register as an offset, OS uses only BX, DI, and SI to hold the offset address of the data. The term pointer is often used for a register holding an offset address. In the following example, BX is used as a pointer:

```
MOV    AL,0           ;initialize AL
MOV    BX,0200H        ;BX points to the offset addr of first byte
ADD    AL,[BX]         ;add the first byte to AL
INC    BX              ;increment BX to point to the next byte
ADD    AL,[BX]         ;add the next byte to AL
INC    BX              ;increment the pointer
ADD    AL,[BX]         ;add the next byte to AL
INC    BX              ;increment the pointer
ADD    AL,[BX]         ;add the last byte to AL
```

The "INC" instruction adds 1 to (increments) its operand. "INC BX" achieves the same result as "ADD BX,1". For the program above, if the offset address where data is located is changed, only one instruction will need to be modified and the rest of the program will be unaffected. Examining the program above shows that there is a pattern of two instructions being repeated. This leads to the idea of using a loop to repeat certain instructions. Implementing a loop requires familiarity with the flag register, discussed later in this chapter.

Logical address and physical address in the data segment

The physical address for data is calculated using the same rules as for the code segment. That is, the physical address of data is calculated by shifting OS left one hex digit and adding the offset value, as shown in Examples 1-2, 1-3, and 1-4.

Example 1-2

Assume that DS is 5000 and the offset is 1950. Calculate the physical address of the byte.

Solution:

DS	:	offset
5000	:	1950

The physical address will be $5000 + 1950 = 51950$.

1. Start with DS.

5	0	0	0
---	---	---	---

2. Shift DS left.

5	0	0	0	0
---	---	---	---	---

3. Add the offset.

5	1	9	5	0
---	---	---	---	---

Example 1-3

If DS = 7FA2H and the offset is 438EH,

- (a) Calculate the physical address. (b) Calculate the lower range.
(c) Calculate the upper range of the data segment. (d) Show the logical address.

Solution:

- (a) 83DAE (7FA20 + 438E) (b) 7FA20 (7FA20 + 0000)
(c) 8FA1F (7FA20 + FFFF) (d) 7FA2:438E

Example 1-4

Assume that the DS register is 578C. To access a given byte of data at physical memory location 67F66, does the data segment cover the range where the data is located? If not, what changes need to be made?

Solution:

No, since the range is 578C0 to 678BF, location 67F66 is not included in this range. To access that byte, DS must be changed so that its range will include that byte.

Little endian convention:

Previous examples used 8-bit or 1-byte data. In this case the bytes are stored one after another in memory. What happens when 16-bit data is used?

For example:

```
MOV  AX,35F3H    ;load 35F3H into AX
MOV  [1500],AX   ;copy the contents of AX to offset 1500H
```

In cases like this, the low byte goes to the low memory location and the high byte goes to the high memory address. In the example above, memory location OS: 1500 contains F3H and memory location OS: 1501 contains 35H.

DS:1500 = F3

DS:1501 = 35

This convention is called little endian versus big endian. See Example 1-5.

Example 1-5

Assume memory locations with the following contents: DS:6826 = 48 and DS:6827 = 22. Show the contents of register BX in the instruction "MOV BX,[6826]".

Solution:

According to the little endian convention used in all 80x86 microprocessors, register BL should contain the value from the low offset address 6826 and register BH the value from offset address 6827, giving BL = 48H and BH = 22H.

	BH	BL
DS:6826 = 48		
DS:6827 = 22	22	48

Extra segment (ES)

ES is a segment register used as an extra data segment. Although in many normal programs this segment is not used, its use is absolutely essential for string operations and is discussed in detail in later chapters.

Memory map of the IBM PC

For a program to be executed on the PC, DOS must first load it into RAM. **Where in RAM will it be loaded?**

To answer that question, we must first explain some very important concepts concerning memory in the Pc. The 20-bit address of the 8088/86 allows a total of 1 megabyte (1024K bytes) of memory space with the address range 00000 - FFFFF.

During the design phase of the first IBM PC, engineers had to decide on the allocation of the 1-megabyte memory space to various sections of the PC. This memory allocation is called a memory map. The memory map of the IBM PC is shown in Figure 1-3. Of this 1 megabyte, 640K bytes from addresses 00000 - 9FFFFH were set aside for RAM. The 128K bytes from A0000H to BFFFFH were allocated for video memory. The remaining 256K bytes from C0000H to FFFFFH were set aside for ROM.

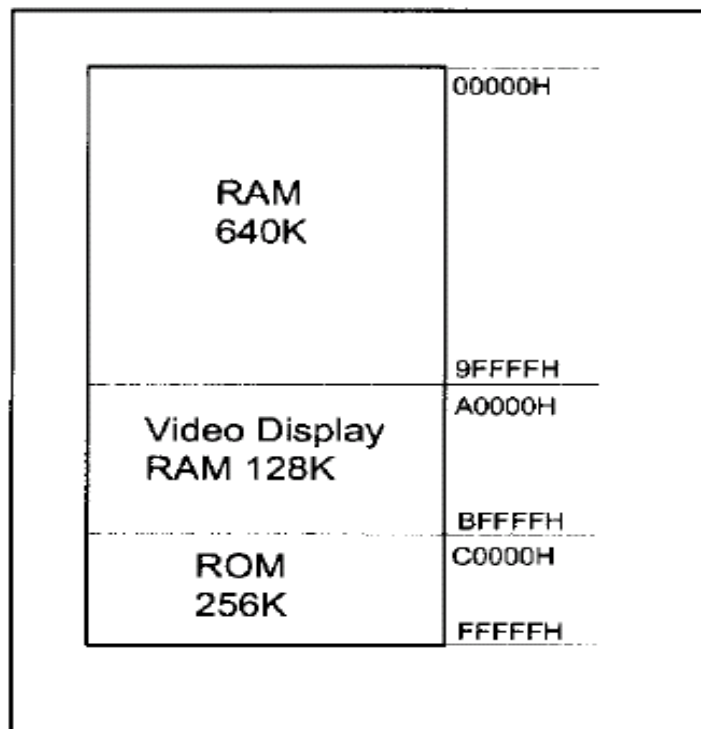


Figure 1-3. Memory Allocation in the PC

1.5 What is a stack, and why is it needed?

The stack is a section of read/write memory (RAM) used by the CPU to store information temporarily. The CPU needs this storage area since there are only a limited number of registers

How stacks are accessed

If the stack is a section of RAM, there must be registers inside the CPU to point to it. The two main registers used to access the stack are the SS (stack segment) register and the SP (stack pointer) register. These registers must be loaded before any instructions accessing the stack are used.

Every register inside the 80x86 (except segment registers and SP) can be stored in the stack and brought back into the CPU from the stack memory. The storing of a CPU register in the stack is called a push, and loading the contents of the stack into the CPU register is called a pop. In other words, a register is pushed onto the stack to store it and popped off the stack to retrieve it. The job of the SP is very critical when push and pop are performed.

In the 80x86, the stack pointer register (SP) points at the current memory location used for the top of the stack and as data is pushed onto the stack it is decremented. It is incremented as data is popped off the stack into the CPU.

Pushing onto the stack

Notice in Example 1-6 that as each PUSH is executed, the contents of the register are saved on the stack and SP is decremented by 2. For every byte of data saved on the stack, SP is decremented once, and since push is saving the contents of a 16-bit register, it is decremented twice. Notice also how the data is stored on the stack.

In the 80x86, the lower byte is always stored in the memory location with the lower address. That is the reason that 24H, the contents of AH, is saved in memory location with address 1235 and AL in location 1234.

Popping the stack

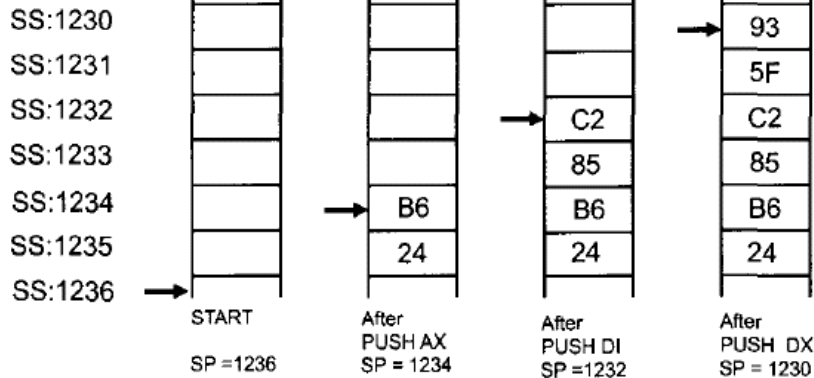
Popping the contents of the stack back into the 80x86 CPU is the opposite process of pushing. With every pop, the top 2 bytes of the stack are copied to the register specified by the instruction and the stack pointer is incremented twice. Although the data actually remains in memory, it is not accessible since the stack pointer is beyond that point. **Example 1-7** demonstrates the POP instruction

Example 1-6

Assuming that SP = 1236, AX = 24B6, DI = 85C2, and DX = 5F93, show the contents of the stack as each of the following instructions is executed:

PUSH AX
PUSH DI
PUSH DX

Solution:

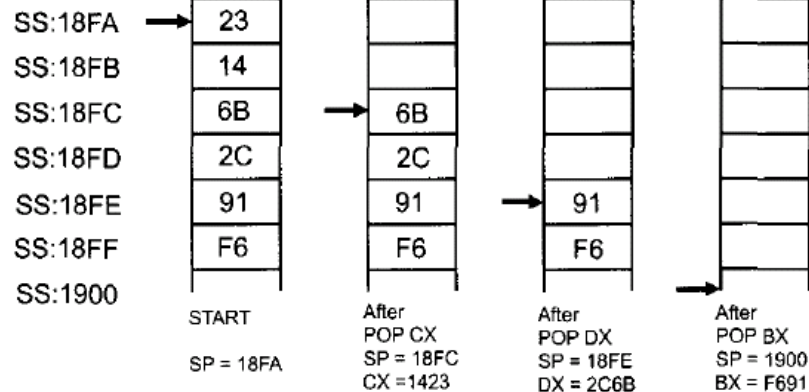


Example 1-7

Assuming that the stack is as shown below, and SP = 18FA, show the contents of the stack and registers as each of the following instructions is executed:

POP CX
POP DX
POP BX

Solution:



Logical address VS, physical address for the stack:

Now one might ask, **what is the exact physical location of the stack?**

That depends on the value of the stack segment (SS) register and SP, the stack pointer. To compute physical addresses for the stack, the same principle is applied as was used for the code and data segments. The method is to shift left SS and then add offset SP, the stack pointer register. This is demonstrated in Example 1-8.

What values are assigned to the SP and SS, and who assigns them?

It is the job of the DOS operating system to assign the values for the SP and SS since memory management is the responsibility of the operating system. Before leaving the discussion of the stack, two points must be made.

First, in the 80x 86 literatures, the top of the stack is the last stack location occupied. This is different from other CPUs.

Second, BP is another register that can be used as an offset into the stack, but it has very special applications and is widely used to access parameters passed between Assembly language programs and high-level language programs such as C.

Example 1-8

If SS = 3500H and the SP is FFFE_H,

- (a) Calculate the physical address of the stack. (b) Calculate the lower range.
(c) Calculate the upper range of the stack segment. (d) Show the logical address of the stack.

Solution:

- (a) 44FFE (35000 + FFFE) (b) 35000 (35000 + 0000)
(c) 44FFF (35000 + FFFF) (d) 3500:FFFE

A few more words about segments in the 80x86

Can a single physical address belong to many different logical addresses?

Yes, look at the case of a physical address value of 15020H. There are many possible logical addresses that represent this single physical address:

Logical address (hex)	Physical address (hex)
1000:5020	15020
1500:0020	15020
1502:0000	15020
1400:1020	15020
1302:2000	15020

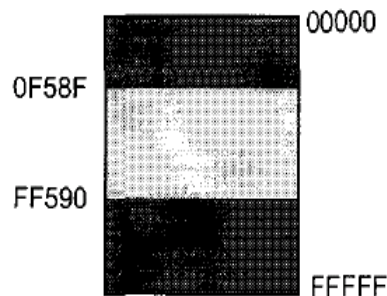
This shows the dynamic behavior of the segment and offset concept in the 8086 CPU. One last point that must be clarified is the case when adding the offset to the shifted segment register results in an address beyond the maximum allowed range of FFFFFH. In that situation, wrap-around will occur. This is shown in Example 1-9.

Example 1-9

What is the range of physical addresses if CS = FF59?

Solution:

The low range is FF590 (FF590 + 0000). The range goes to FFFFF and wraps around, from 00000 to 0F58F (FF590 + FFFF = 0F58F), which is illustrated below.



Overlapping

In calculating the physical address, it is possible that two segments can overlap, which is desirable in some circumstances. Figure 1-4 illustrates overlapping and non overlapping segments.

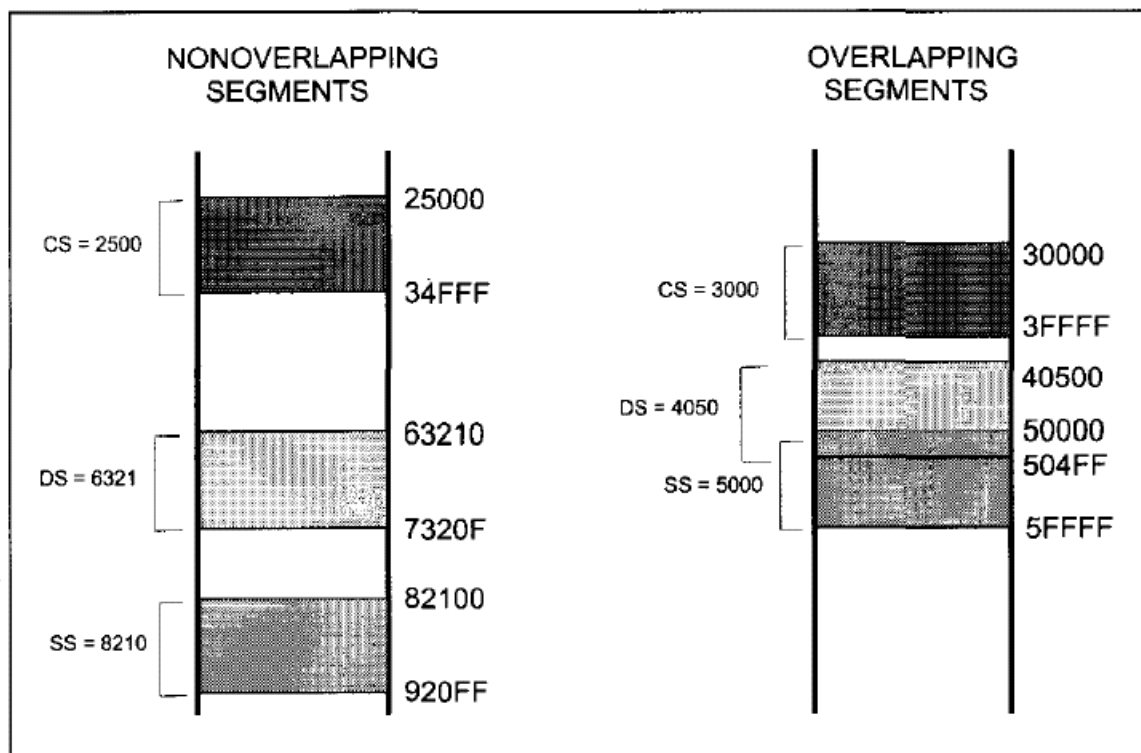


Figure 1-4. Nonoverlapping vs. Overlapping Segments

Flag register

The flag register is a 16-bit register sometimes referred to as the status register. Although the register is 16 bits wide, only some of the bits are used. The rest are either undefined or reserved by Intel. Six of the flags are called conditional flags, meaning that they indicate some condition that resulted after an instruction was executed. These six are CF, PF, AF, ZF, SF, and OF. The three remaining flags are sometimes called control flags since they are used to control the operation of instructions before they are executed. A diagram of the flag register is shown in Figure 1-5.

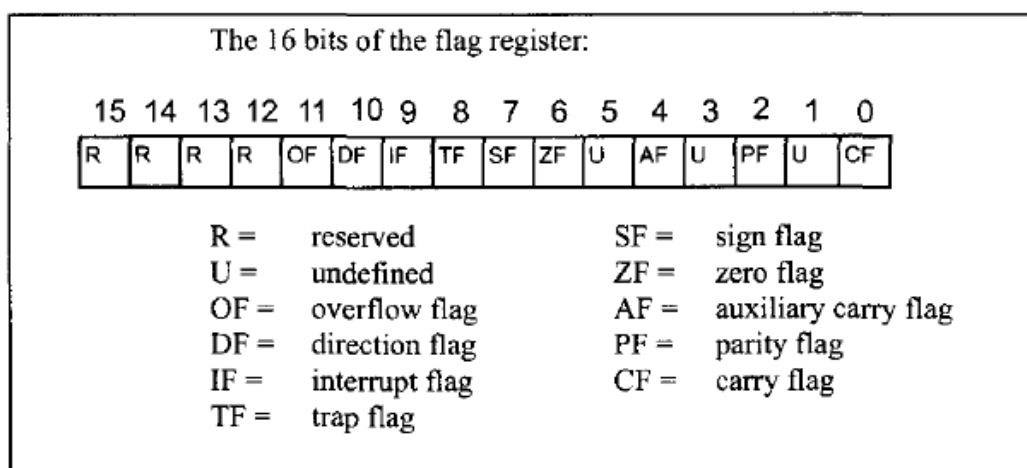


Figure 1-5. Flag Register

(Reprinted by permission of Intel Corporation, Copyright Intel Corp. 1989)

Bits of the flag register

Below are listed the bits of the flag register that are used in 80x86 Assembly language programming. A brief explanation of each bit is given. How these flag bits are used will be seen in programming examples.

CF, the Carry Flag. This flag is set whenever there is a carry out, either from d7 after an 8-bit operation or from d15 after a 16-bit data operation.

PF, the Parity Flag. After certain operations, the parity of the result's low-order byte is checked. If the byte has an even number of 1s, the parity flag is set to 1; otherwise, it is cleared.

AF, Auxiliary Carry Flag. If there is a carry from d3 to d4 of an operation, this bit is set; otherwise, it is cleared (set equal to zero). This flag is used by the instructions that perform BCD (binary coded decimal) arithmetic.

ZF, the Zero Flag. The zero flag is set to 1 if the result of an arithmetic or logical operation is zero; otherwise, it is cleared.

SF, the Sign Flag. Binary representation of signed numbers uses the most significant bit as the sign bit. After arithmetic or logic operations, the status of this sign bit is copied into the SF, thereby indicating the sign of the result.

TF, the Trap Flag. When this flag is set it allows the program to single-step, meaning to execute one instruction at a time. Single-stepping is used for debugging purposes.

IF, Interrupt Enable Flag. This bit is set or cleared to enable or disable only the external mask able interrupt requests.

DF, the Direction Flag. This bit is used to control the direction of string operations.

OF, the Overflow Flag. This flag is set whenever the result of a signed number operation is too large, causing the high-order bit to overflow into the sign bit. In general, the carry flag is used to detect errors in unsigned arithmetic operations. The overflow flag is only used to detect errors in signed arithmetic operations.

Flag register and ADD instruction

In this section we examine the impact of the ADD instruction on the flag register as an example of the use of the flag bits. The flag bits affected by the ADD instruction are CF (carry flag), PF (parity flag), AF (auxiliary carry flag), ZF (zero flag), SF (sign flag), and OF (overflow flag).

Example 1-10

Show how the flag register is affected by the addition of 38H and 2FH.

Solution:

```
MOV  BH,38H      ;BH= 38H
ADD  BH,2FH      ;add 2F to BH, now BH=67H
```

	38	0011	1000
+	<u>2F</u>	<u>0010</u>	<u>1111</u>
	67	0110	0111

CF = 0 since there is no carry beyond d7

PF = 0 since there is an odd number of 1s in the result

AF = 1 since there is a carry from d3 to d4

ZF = 0 since the result is not zero

SF = 0 since d7 of the result is zero

Example 1-11

Show how the flag register is affected by

```
MOV  AL,9CH      ;AL=9CH
MOV  DH,64H      ;DH=64H
ADD  AL,DH        ;now AL=0
```

Solution:

	9C	1001	1100
+	<u>64</u>	0110	<u>0100</u>
	00	0000	0000

CF=1 since there is a carry beyond d7

PF=1 since there is an even number of 1s in the result

AF=1 since there is a carry from d3 to d4

ZF=1 since the result is zero

SF=0 since d7 of the result is zero

The same concepts apply for 16-bit addition, as shown in Examples 1-12 and 1-13. It is important to notice the differences between 8-bit and 16-bit operations in terms of their impact on the flag bits. The parity bit only counts the lower 8-bits of the result and is set accordingly. Also notice the CF bit. The carry flag is set if there is a carry beyond bit d15 instead of bit d7.

Example 1-12

Show how the flag register is affected by

```
MOV  AX,34F5H    ;AX= 34F5H
ADD  AX,95EBH     ;now AX= CAE0H
```

Solution:

	34F5	0011	0100	1111	0101
+	<u>95EB</u>	1001	0101	1110	<u>1011</u>
	CAE0	1100	1010	1110	0000

CF = 0 since there is no carry beyond d15

PF = 0 since there is an odd number of 1s in the lower byte

AF = 1 since there is a carry from d3 to d4

ZF = 0 since the result is not zero

SF = 1 since d15 of the result is one

Example 1-13

Show how the flag register is affected by

```
MOV  BX,AAAAH    ;BX= AAAAH
ADD  BX,5556H     ;now BX= 0000H
```

Solution:

	AAAA	1010	1010	1010	1010
+	<u>5556</u>	0101	0101	0101	<u>0110</u>
	0000	0000	0000	0000	0000

CF = 1 since there is a carry beyond d15

PF = 1 since there is an even number of 1s in the lower byte

AF = 1 since there is a carry from d3 to d4

ZF = 1 since the result is zero

SF = 0 since d15 of the result is zero

Notice the zero flag (ZF) status after the execution of the ADD instruction. Since the result of the entire 16-bit operation is zero (meaning the contents of BX), ZF is set to high. Do all instructions affect the flag bits? The answer is no; some instructions such as data transfers (MOV) affect no flags. As an exercise, run these examples on DEBUG to see the effect of various instructions on the flag register.

Example 1-14

Show how the flag register is affected by

```
MOV  AX,94C2H    ;AX=94C2H
MOV  BX,323EH    ;BX=323EH
ADD  AX,BX        ;now AX=C700H
MOV  DX,AX        ;now DX=C700H
MOV  CX,DX        ;now CX=C700H
```

Solution:

	94C2	1001	0100	1100	0010
+	323E	0011	0010	0011	1110
	C700	1100	0111	0000	0000

After the ADD operation, the following are the flag bits:

CF = 0 since there is no carry beyond d15

PF = 1 since there is an even number of 1s in the lower byte

AF = 1 since there is a carry from d3 to d4

ZF = 0 since the result is not zero

SF = 1 since d15 of the result is 1

Running the instructions in Example 1-14 in DEBUG will verify that MOV instructions have no effect on the flag. How these flag bits are used in programming is discussed in future chapters in the context of many applications.

Use of the zero flag for looping

One of the most widely used applications of the flag register is the use of the zero flag to implement program loops. The term loop refers to a set of instructions that is repeated a number of times. For example, to add 5 bytes of data, a counter can be used to keep track of how many times the loop needs to be repeated. Each time the addition is performed the counter is decremented and the zero flag is checked. When the counter becomes zero, the zero flag is set ($ZF = 1$) and the loop is stopped. The following shows the implementation of the looping concept in the program, which adds 5 bytes of data. Register CX is used to hold the counter and BX is the offset pointer (SI or could have been used instead). AL is initialized before the start of the loop. In each iteration, ZF is checked by the JNZ instruction.

JNZ stands for "Jump Not Zero" meaning that if $ZF = 0$, jump to a new address. If $ZF = 1$, the jump is not performed and the instruction below the jump will be executed. Notice that the JNZ instruction must come immediately after the instruction that decrements CX since JNZ needs to check the affect of "DEC CX" on the zero flag. If any instruction were placed between them, that instruction might affect the zero flag.

```

MOV    CX,05      ;CX holds the loop count
MOV    BX,0200H   ;BX holds the offset data address
MOV    AL,00      ;initialize AL
ADD_LP: ADD    AL,[BX] ;add the next byte to AL
        INC    BX    ;increment the data pointer
        DEC    CX    ;decrement the loop counter
        JNZ    ADD_LP ;jump to next iteration if counter not zero
```

1.6 80x86 Addressing Modes

The CPU can access operands (data) in various ways, called addressing modes. The number of addressing modes is determined when the microprocessor is designed and cannot be changed. The 80x86 provides a total of seven distinct addressing modes:

A. Register addressing mode

The register addressing mode involves the use of registers to hold the data to be manipulated. Memory is not accessed when this addressing mode is executed; therefore, it is relatively fast. **Examples** of register addressing mode follow:

```
MOV BX, DX ;copy contents of DX into BX
MOV ES, AX ;copy contents of AX into ES
ADD AL, BH ;add the contents of BH to the contents of AL and store in AL
```

It should be noted that the source and destination registers must match in size. In other words coding "MOV CL,AX" will give an error, since the source is a 16-bit register and the destination is an 8-bit register.

B. Immediate addressing mode

In the immediate addressing mode, the source operand is a constant. In immediate addressing mode, as the name implies, when the instruction is assembled, the operand comes immediately after the opcode. For this reason, this addressing mode executes quickly. However, in programming it has limited use. Immediate addressing mode can be used to load information into any of the registers except the segment registers and flag registers. **Examples:**

```
MOV AX 3F50H ;move 3F50H into AX
MOV CX, 425 ;load decimal value 425 into CX
MOV BL, 40H ;load 40H into BL
```

To move information to the segment registers, the data must first be moved to a general-purpose registers and then to the segment register. **Example:**

```
MOV AX,2550H
MOV DS,AX
```

In other words, the following would produce an error:

```
MOV DS,0123H ;illegal!!
```

In the first two addressing modes, the operands are either inside the microprocessor or tagged along with the instruction. In most programs, the data to be processed is often in some memory location outside the CPU. There are many ways of accessing the data in the data segment. The following describes those different methods.

C. Direct addressing mode

In the direct addressing mode the data is in some memory location(s) and the address of the data in memory comes immediately after the instruction. Note that in immediate addressing, the operand itself is provided with the instruction, whereas in direct addressing mode, the address of the operand is provided with the instruction. This address is the offset address and one can calculate the physical address by shifting left the DS register and adding it to the offset as follows:

```
MOV DL,[2400] ;move contents of DS:2400H into DL
```

In this case the physical address is calculated by combining the contents of offset location 2400 with DS, the data segment register. Notice the bracket around the address. In the absence of this bracket it will give an error since it is interpreted to move the value 2400 (16-bit data) into register DL, an 8-bit register. **Example 1-15** gives another example of direct addressing.

Example 1-15

Find the physical address of the memory location and its contents after the execution of the following, assuming that DS = 1512H.

```
MOV    AL,99H
MOV    [3518],AL
```

Solution:

First AL is initialized to 99H, then in line two, the contents of AL are moved to logical address DS:3518 which is 1512:3518. Shifting DS left and adding it to the offset gives the physical address of 18638H (15120H + 3518H = 18638H). That means after the execution of the second instruction, the memory location with address 18638H will contain the value 99H.

D. Register indirect addressing

In the register indirect addressing mode, the address of the memory location where the operand resides is held by a register. The registers used for this purpose are SI, DI, and BX. If these three registers are used as pointers, that is, if they hold the offset of the memory location, they must be combined with DS in order to generate the 20-bit physical address. **For example:**

```
MOV    AL,[BX]           ;moves into AL the contents of the memory location
                           ;pointed to by DS:BX.
```

Notice that BX is in brackets. In the absence of brackets, it is interpreted as an instruction moving the contents of register BX to AL (which gives an error because source and destination do not match) instead of the contents of the memory location whose offset address is in BX. The physical address is calculated by shifting DS left one hex position and adding BX to it. The same rules apply when using register SI or DI.

```
MOV    CL,[SI]           ;move contents of DS:SI into CL
MOV    [DI],AH           ;move contents of AH into DS:DI
```

In the examples above, the data moved is byte sized. Example 1-16 shows 16-bit operands.

Example 1-16

Assume that DS = 1120, SI = 2498, and AX = 17FE. Show the contents of memory locations after the execution of

```
MOV [SI],AX
```

Solution:

The contents of AX are moved into memory locations with logical address DS:SI and DS:SI + 1; therefore, the physical address starts at DS (shifted left) + SI = 13698. According to the little endian convention, low address 13698H contains FE, the low byte, and high address 13699H will contain 17, the high byte.

E. Based relative addressing mode

In the based relative addressing mode, base registers BX and BP, as well as a displacement value, are used to calculate what is called the effective address. The default segments used for the calculation of the physical address (PA) are DS for BX and SS for BP. For example:

```
MOV    CX,[BX]+10    ;move DS:BX+10 and DS:BX+10+1 into CX
                        ;PA = DS (shifted left) + BX + 10
```

Alternative coding's are "MOV CX,[BX+10]" or "MOV CX,10[BX]". Again the low address contents will go into CL and the high address contents into CH. In the case of the BP register,

```
MOV    AL,[BP]+5      ;PA = SS (shifted left) + BP + 5
```

Again, alternative codings are "MOV AL,[BP+5]" or "MOV AL,5[BP]". A brief mention should be made of the terminology effective address used in Intel literature. In "MOV AL,[BP]+5", BP+5 is called the effective address since the fifth byte from the beginning of the offset BP is moved to register AL. Similarly in "MOV CX,[BX]+10", BX+10 is called the effective address.

F. Indexed relative addressing modes

The indexed relative addressing mode works the same as the based relative addressing mode, except that registers DI and SI hold the offset address. Examples:

```
MOV    DX,[SI]+5      ;PA = DS (shifted left) + SI + 5
MOV    CL,[DI]+20     ;PA = DS (shifted left) + DI + 20
```

Example 1-17 gives further examples of indexed relative addressing mode.

Example 1-17

Assume that DS = 4500, SS = 2000, BX = 2100, SI = 1486, DI = 8500, BP = 7814, and AX = 2512. Show the exact physical memory location where AX is stored in each of the following. All values are in hex.

- (a) MOV [BX]+20,AX (b) MOV [SI]+10,AX
- (c) MOV [DI]+4,AX (d) MOV [BP]+12,AX

Solution:

In each case PA = segment register (shifted left) + offset register + displacement.

- (a) DS:BX+20 location 47120 = (12) and 47121 = (25)
- (b) DS:SI+10 location 46496 = (12) and 46497 = (25)
- (c) DS:DI+4 location 4D504 = (12) and 4D505 = (25)
- (d) SS:BP+12 location 27826 = (12) and 27827 = (25)

G. Based indexed addressing mode

By combining based and indexed addressing modes, a new addressing mode is derived called the based indexed addressing mode. In this mode, one base register and one index register are used. Examples:

```
MOV CL,[BX][DI]+8      ;PA = DS (shifted left) + BX + DI + 8
MOV CH,[BX][SI]+20     ;PA = DS (shifted left) + BX + SI + 20
MOV AH,[BP][DI]+12     ;PA = SS (shifted left) + BP + DI + 12
MOV AH,[BP][SI]+29     ;PA = SS (shifted left) + BP + SI + 29
```

The coding of the instructions above can vary; for example, the last example could have been written

```
MOV AH,[BP+SI+29]
or
MOV AH,[SI+BP+29] ;the register order does not matter.
```

Note that "MOV AX,[SI][DI]+displacement" is illegal.

In many of the examples above, the MOV instruction was used for the sake of clarity, even though one can use any instruction as long as that instruction supports the addressing mode. For example, the instruction "ADD DL,[BX]" would add the contents of the memory location pointed at by DS:BX to the contents of register DL.

Table 1-3: Offset Registers for Various Segments

Segment register:	CS	DS	ES	SS
Offset register(s):	IP	SI, DI, BX	SI, DI, BX	SP, BP

Segment overrides

Table 1.3 provides a summary of the offset registers that can be used with the four segment registers of the 80x86. The 80x86 CPU allows the program to override the default segment and use any segment register. To do that, specify the segment in the code. For example, in "MOV AL,[BX]", the physical address of the operand to be moved into AL is DS:BX, as was shown earlier since DS is the default segment for pointer BX. To override that default, specify the desired segment in the instruction as "MOV AL,ES:[BX]". Now the address of the operand being moved to AL is ES:BX instead of DS:BX. Extensive use of all these addressing modes is

Chapter 1: THE 80x86 MICROPROCESSOR

shown in future chapters in the context of program examples. Table 1.4 shows more examples of segment overrides shown next to the default address in the absence of the override. Table 1.5 summarizes addressing modes of the 8086/88.

Table 1-4: Sample Segment Overrides

Instruction	Segment Used	Default Segment
MOV AX,CS:[BP]	CS:BP	SS:BP
MOV DX,SS:[SI]	SS:SI	DS:SI
MOV AX,DS:[BP]	DS:BP	SS:BP
MOV CX,ES:[BX]+12	ES:BX+12	DS:BX+12
MOV SS:[BX][DI]+32,AX	SS:BX+DI+32	DS:BX+DI+32

Table 1-5: Summary of 80x86 Addressing Modes

Addressing Mode	Operand	Default Segment
Register	reg	none
Immediate	data	none
Direct	[offset]	DS
Register indirect	[BX]	DS
	[SI]	DS
	[DI]	DS
Based relative	[BX]+disp	DS
	[BP]+disp	SS
Indexed relative	[DI]+disp	DS
	[SI]+disp	DS
Based indexed relative	[BX][SI]+disp	DS
	[BX][DI]+disp	DS
	[BP][SI]+ disp	SS
	[BP][DI]+ disp	SS

2.1 CONTROL TRANSFER INSTRUCTIONS

In the sequence of instructions to be executed, it is often necessary to transfer program control to a different location. There are many instructions in the 80x86 to achieve this. This section covers the control transfer instructions available in the 8086 Assembly language. Before that, however, it is necessary to explain the concept of FAR and NEAR as it applies to jump and call instructions.

FAR and NEAR

If control is transferred to a memory location within the current code segment, it is NEAR. This is sometimes called *intrasegment* (within segment). If control is transferred outside the current code segment, it is a FAR or intersegment (between segments) jump. Since the CS:IP registers always point to the address of the next instruction to be executed, they must be updated when a control transfer instruction is executed. In a NEAR jump, the IP is updated and CS remains the same, since control is still inside the current code segment. In a FAR jump, because control is passing outside the current code segment, both CS and IP have to be updated to the new values. In other words, in any control transfer instruction such as jump or call, the IP must be changed, but only in the FAR case is the CS changed, too.

Conditional jumps

Conditional jumps, summarized in Table 2-1, have mnemonics such as JNZ (Jump not zero) and JC (Jump if carry). In the conditional jump, control is transferred to a new location if a certain condition is met. The flag register is the one that indicates the current condition. For example, with "JNZ label", the processor looks at the zero flag to see if it is raised. If not, the CPU starts to fetch and execute instructions from the address of the label. If $ZF = 1$, it will not jump but will execute the next instruction below the JNZ.

Short jumps

All conditional jumps are short jumps. In a short jump, the address of the target must be within -128 to + 127 bytes of the IP. In other words, the conditional jump is a two-byte instruction: one byte is the opcode of the J condition and the second byte is a value between 00 and FF. An offset range of 00 to FF gives 256 possible addresses; these are split between backward jumps (to -128) and forward jumps (to + 127).

In a jump backward, the second byte is the 2's complement of the displacement value. To calculate the target address, the second byte is added to the IP of the instruction after the jump. To understand this, look at the unassembled code below.

```

1067:0000 B86610    MOV    AX,1066
1067:0003 8ED8      MOV    DS,AX
1067:0005 B90500    MOV    CX,0005
1067:0008 BB0000    MOV    BX,0000
1067:000D 0207      ADD    AL,[BX]
1067:000F 43        INC    BX
1067:0010 49        DEC    CX
1067:0011 75FA      JNZ    000D
1067:0013 A20500    MOV    [0005],AL
1067:0016 B44C      MOV    AH,4C
1067:0018 CD21      INT    21

```

Table 2-1: 8086 Conditional Jump Instructions

Mnemonic	Condition Tested	"Jump IF ..."
JA/JNBE	(CF = 0) and (ZF = 0)	above/not below nor zero
JAЕ/JNB	CF = 0	above or equal/not below
JB/JNAE	CF = 1	below/not above nor equal
JBE/JNA	(CF or ZF) = 1	below or equal/not above
JC	CF = 1	carry
JE/JZ	ZF = 1	equal/zero
JG/JNLE	((SF xor OF) or ZF) = 0	greater/not less nor equal
JGE/JNL	(SF xor OF) = 0	greater or equal/not less
JL/JNGE	(SF xor OF) = 1	less/not greater nor equal
JLE/JNG	((SF xor OF) or ZF) = 1	less or equal/not greater
JNC	CF = 0	not carry
JNE/JNZ	ZF = 0	not equal/not zero
JNO	OF = 0	not overflow
JNP/JPO	PF = 0	not parity/parity odd
JNS	SF = 0	not sign
JO	OF = 1	overflow
JP/JPE	PF = 1	parity/parity equal
JS	SF = 1	sign

The instruction "JNZ AGAIN" was assembled as "JNZ 000D", and 000D is the address of the instruction with the label AGAIN. The instruction "JNZ 000D" has the opcode 75 and the target address FA, which is located at offset addresses 0011 and 0012. This is followed by "MOV SUM,AL", which is located beginning at offset address 0013. The IP value of MOV, 0013, is added to FA to calculate the address of label AGAIN ($0013 + FA = 000D$) and the carry is dropped. In reality, FA is the 2's complement of -6, meaning that the address of the target is -6 bytes from the IP of the next instruction.

Similarly, the target address for a forward jump is calculated by adding the IP of the following instruction to the operand. In that case the displacement value is

positive, as shown next. Below is a portion of a list file showing the opcodes for several conditional jumps.

```
0005 8A 47 02 AGAIN:    MOV  AL,[BX]+2
0008 3C 61              CMP  AL,61H
000A 72 06              JB   NEXT
000C 3C 7A              CMP  AL,7AH
000E 77 02              JA   NEXT
0010 24 DF              AND  AL,ODFH
0012 88 04  NEXT:      MOV  [SI],AL
```

In the program above, "JB NEXT" has the opcode 72 and the target address 06 and is located at IP = 000A and 000B. The jump will be 6 bytes from the next instruction, which is IP = 000C. Adding gives us 000CH + 0006H = 0012H, which is the exact address of the NEXT label. Look also at "JA NEXT", which has 77 and 02 for the opcode and displacement, respectively. The IP of the following instruction, 0010, is added to 02 to get 0012, the address of the target location.

It must be emphasized that regardless of whether the jump is forward or backward, for conditional jumps the address of the target address can never be more than -128 to + 127 bytes away from the IP associated with the instruction following the jump (- for the backward jump and + for the forward jump). If any attempt is made to violate this rule, the assembler will generate a "relative jump out of range" message. These conditional jumps are sometimes referred to as **SHORT jumps**.

Unconditional jumps

"JMP label" is an unconditional jump in which control is transferred unconditionally to the target location label. The unconditional jump can take the following forms:

1. SHORT JUMP, which is specified by the format "JMP SHORT label". This is a jump in which the address of the target location is within -128 to + 127 bytes of memory relative to the address of the current IP. In this case, the opcode is EB and the operand is 1 byte in the range 00 to FF. The operand byte is added to the current IP to calculate the target address. If the jump is backward, the operand is in 2's complement. This is exactly like the J condition case. Coding the directive "short" makes the jump more efficient in that it will be assembled into a 2-byte instruction instead of a 3-byte instruction.

2. NEAR JUMP, which is the default, has the format "JMP label". This is a near jump (within the current code segment) and has the opcode E9. The target address can be any of the addressing modes of direct, register, register indirect, or memory indirect:

(a) **Direct JUMP** is exactly like the short jump explained earlier, except that the target address can be anywhere in the segment within the range +32767 to -32768 of the current IP.

(b) **Register indirect JUMP**; the target address is in a register. For example, in "JMP BX", IP takes the value BX.

(c) **Memory indirect JMP**; the target address is the contents of two memory locations pointed at by the register. Example: "JMP [DI]" will replace the IP with the contents of memory locations pointed at by DI and DI+ 1.

3. FAR JUMP which has the format "JMP FAR PTR label". This is a jump out of the current code segment, meaning that not only the IP but also the CS is replaced with new values.

CALL statements

Another control transfer instruction is the CALL instruction, which is used to call a procedure. CALLs to procedures are used to perform tasks that need to be performed frequently. This makes a program more structured. The target address could be in the current segment, in which case it will be a NEAR call or outside the current CS segment, which is a FAR call. To make sure that after execution of the called subroutine the microprocessor knows where to come back, the microprocessor automatically saves the address of the instruction following the call on the stack. It must be noted that in the NEAR call only the IP is saved on the stack, and in a FAR call both CS and IP are saved. When a subroutine is called, control is transferred to that subroutine and the processor saves the IP (and CS in the case of a FAR call) and begins to fetch instructions from the new location. After finishing execution of the subroutine, for control to be transferred back to the caller, the last instruction in the called subroutine must be RET (return). In the same way that the assembler generates different opcode for FAR and NEAR calls, the opcode for the RET instruction in the case of NEAR and FAR is different, as well. For NEAR calls, the IP is restored; for FAR calls, both CS and IP are restored. This will ensure that control is given back to the caller. As an example, assume that SP = FFFEh and the following code is a portion of the program unassembled in DEBUG:

```
12B0:0200 BB1295 MOV BX,9512
12B0:0203 E8FA00 CALL 0300
12B0:0206 B82F14 MOV AX,142F
```

Since the CALL instruction is a NEAR call, meaning that it is in the same code segment (different IP, same CS), only IP is saved on the stack. In this case, the IP address of the instruction after the call is saved on the stack as shown in Figure 2-5. That IP will be 0206, which belongs to the "MOV AX,142F" instruction.

The last instruction of the called subroutine must be a RET instruction which directs the CPU to POP the top 2 bytes of the stack into the IP and resume executing at offset address 0206. For this reason, the number of PUSH and POP instructions (which alter the SP) must match. In other words, for every PUSH there must be a POP.

```
12B0:0300 53 PUSH BX
12B0:0301 ... ..
.....
12B0:0309 5B POP BX
12B0:030A C3 RET
```

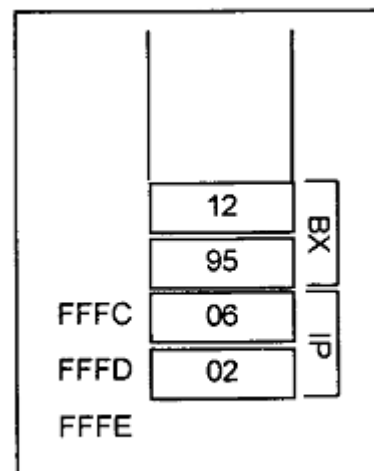


Figure 2-5. IP in the Stack

2.2 Assembly language subroutines

In Assembly language programming it is common to have one main program and many subroutines to be called from the main program. This allows you to make each subroutine into a separate module. Each module can be tested separately and then brought together, as will be shown next chapters. The main program is the entry point from DOS and is FAR, as explained earlier, but the subroutines called within the main program can be FAR or NEAR.

Remember that NEAR routines are in the same code segment, while FAR routines are outside the current code segment. If there is no specific mention of FAR after the directive PROC, it defaults to NEAR, as shown in Figure 2-6. From now on, all code segments will be written in that format.

Rules for names in Assembly language

By choosing label names that are meaningful, a programmer can make a program much easier to read and maintain. There are several rules that names must follow. **First**, each label name must be unique. The names used for labels in Assembly

language programming consist of alphabetic letters in both upper and lower case, the digits 0 through 9, and the special characters question mark (?), period (.), at (@), underline (_), and dollar sign (\$). The first character of the name must be an alphabetic character or special character. It cannot be a digit. The period can only be used as the first character, but this is not recommended since later versions of MASM have several reserved words that begin with a period. Names may be up to 31 characters long.

```

MAIN      .CODE
          PROC FAR           ;THIS IS THE ENTRY POINT FOR DOS
          MOV AX,@DATA
          MOV DS,AX
          CALL SUBR1
          CALL SUBR2
          CALL SUBR3
          MOV AH,4CH
          INT 21H
MAIN      ENDP
;
SUBR1     PROC
          ...
          RET
SUBR1     ENDP
;
SUBR2     PROC
          ...
          RET
SUBR2     ENDP
;
SUBR3     PROC
          ...
          RET
SUBR3     ENDP
;
          END MAIN          ;THIS IS THE EXIT POINT

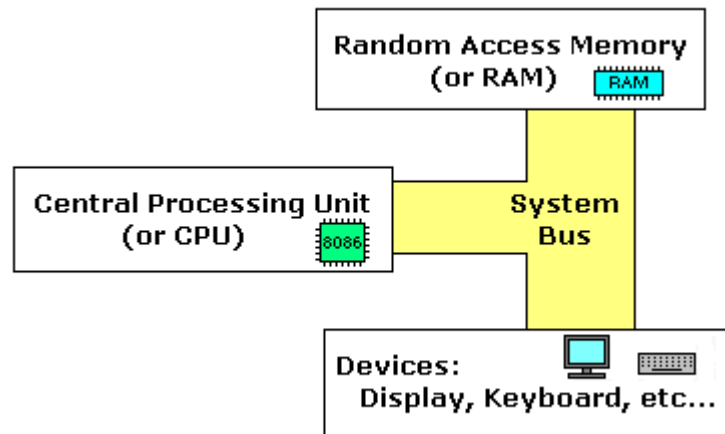
```

Figure 2-6. Shell of Assembly Language Subroutines

Lab 1

Inside the CPU

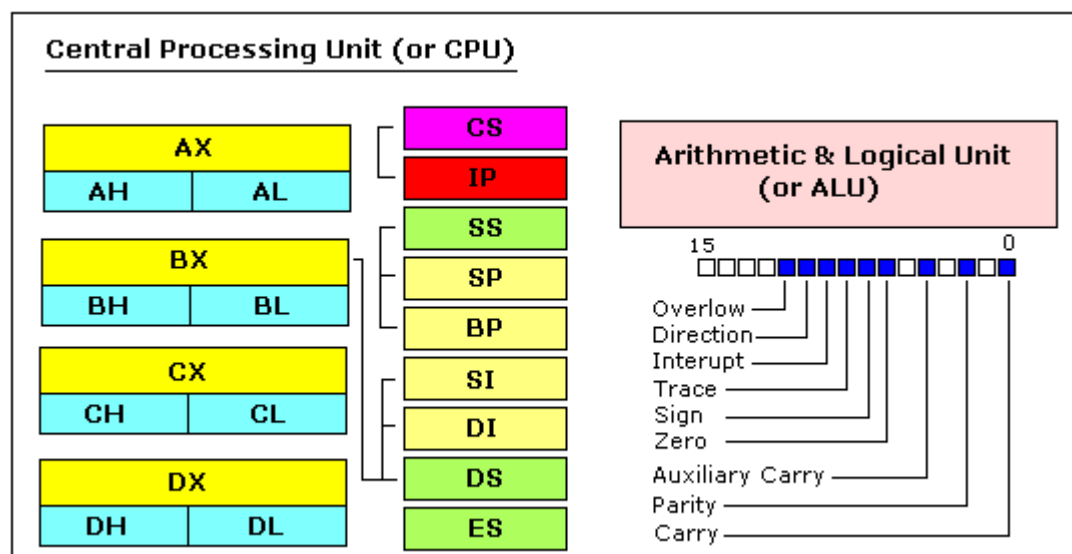
The simple computer model as:



The **system bus** (shown in yellow) connects the various components of a computer.

The **CPU** is the heart of the computer, most of computations occur inside the **CPU**.

RAM is a place to where the programs are loaded in order to be executed.



Assembly Language

Assembly language is a low level programming language. An Assembly language program consists of, among other things, a series of lines of Assembly language instructions. An Assembly language instruction consists of a mnemonic, optionally followed by one or two operands. The operands are the data items being manipulated, and the mnemonics are the commands to the CPU, telling it what to do with those items.

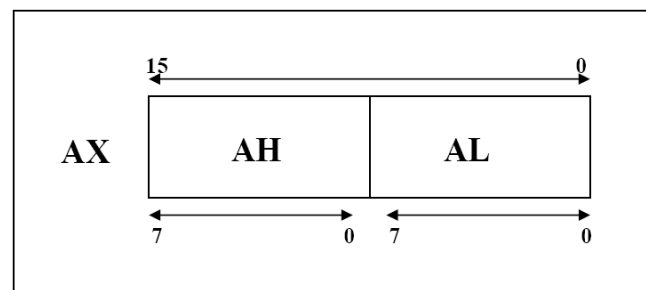
Registers

In the CPU, registers are used to store information temporarily. That information could be one or two bytes of data to be processed or the address of data.

General purpose registers

The general purpose registers, are used for arithmetic and data movement. Each register can be addressed as either 16-bit or 8 bit value. Example, AX register is a 16-bit register, its upper 8-bit is called AH, and its lower 8-bit is called AL. Bit 0 in AL corresponds to bit 0 in AX and bit 8 in AH corresponds to bit 8 in AX.

- **AX** - the accumulator register (divided into **AH** / **AL**).
- **BX** - the base address register (divided into **BH** / **BL**).
- **CX** – the counter register (divided into **CH** / **CL**).
- **DX** - the data register (divided into **DH** / **DL**).



Each **general purpose register** has special attributes:

1- AX (Accumulator): AX is the accumulator register because it is favored by the CPU for arithmetic operations. Other operations are also slightly more efficient when performed using **AX**.

2- BX (Base): the BX register can hold the address of a procedure or variable. Three other registers with this ability are **SI**, **DI** and **BP**. The **BX** register can also perform arithmetic and data movement.

3- CX (Counter): the CX register acts as a counter for repeating or looping instructions. These instructions automatically repeat and decrement **CX**.

4- DX (Data): the DX register has a special role in multiply and divide operation. When multiplying for example **DX** hold the high 16 bit of the product.

Lab 2

Instructions

MOV instruction

Simply stated, the MOV instruction copies data from one location to another. It has the following format:

MOV destination, source ;copy source operand to destination

This instruction tells the CPU to move (in reality, copy) the source operand to the destination operand. For example, the instruction "MOV DX,CX" copies the contents of register CX to register DX. After this instruction is executed, register DX will have the same value as register CX. The MOV instruction does not affect the source operand.

These types of operands are supported:

MOV REG, memory

MOV memory, REG

MOV REG, REG

MOV memory, immediate

MOV REG, immediate

REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

memory: [BX], [BX+SI+7], variable, etc...

immediate: 5, -24, 3Fh, 10001101b, etc...

Example:

MOV AX,2345H ;load 2345H into AX

MOV DS,AX ;then load the value of AX into DS

ADD instruction

The ADD instruction has the following format:

ADD destination, source ;ADD the source operand to the destination

The ADD instruction tells the CPU to add the source and the destination operands and put the result in the destination.

operand1 = operand1 + operand2

These types of operands are supported:

ADD REG, memory

ADD memory, REG

ADD REG, REG

ADD memory, immediate

ADD REG, immediate

Example:

MOV AL, 5 ; AL = 5

ADD AL, -3 ; AL = 2

RET

INT instruction

Interrupts can be seen as a number of functions. These functions make the programming much easier.

To make a software interrupt there is an INT instruction, it has very simple syntax:

INT value

Where value can be a number between 0 to 255 (or 0 to 0FFh),

Each interrupt may have sub-functions.

To specify a sub-function AH register should be set before calling interrupt.

Each interrupt may have up to 256 sub-functions (so we get $256 * 256 = 65536$ functions). In general AH register is used, but sometimes other registers maybe in use. Generally other registers are used to pass parameters and data to sub-function.

These types of operands are supported:

INT immediate byte

Example:

MOV AH, 0Eh ; teletype.

MOV AL, 'A'

INT 10h ; BIOS interrupt.

RET

Eum8086



Lab 3

Assembly language instructions and pseudo-Instructions

An Assembly language program is composed of a series of statements that are either instructions or pseudo-instructions, also called directives. Instructions are translated by the assembler into machine code. Pseudo-instructions are not translated into machine code; They direct the assembler in how to translate the instructions into machine code. The statements of an Assembly language program are grouped into segments.

Directives do not generate any machine code and are used only by the assembler as opposed to instructions, which are translated into machine code for the CPU to execute. In the below figure the commands DB, END, and ENDP are examples of directives.

;THE FORM OF AN ASSEMBLY LANGUAGE PROGRAM			
;NOTE: USING SIMPLIFIED SEGMENT DEFINITION			
	.MODEL	SMALL	
	.STACK	64	
	.DATA		
DATA1	DB	52H	
DATA2	DB	29H	
SUM	DB	?	
	.CODE		
MAIN	PROC	FAR	;this is the program entry point
	MOV	AX,@DATA	;load the data segment address
	MOV	DS,AX	;assign value to DS
	MOV	AL,DATA1	;get the first operand
	MOV	BL,DATA2	;get the second operand
	ADD	AL,BL	;add the operands
	MOV	SUM,AL	;store the result in location SUM
	MOV	AH,4CH	;set up to return to DOS
	INT	21H	;
MAIN	ENDP		
	END	MAIN	;this is the program exit point

Notes:

- A given Assembly language program is a series of statements, or lines, which are either Assembly language instructions such as ADD and MOV, or statements called directives. *Directives* (also called *pseudo-instructions*) give directions to the assembler about how it should translate the Assembly language instructions into machine code.

- What is the purpose of pseudo-instructions? Pseudo-instructions direct the assembler as to how to assemble the program.
- Instructions are translated by the assembler into machine code, whereas pseudo-instructions or directives are not.

Identify the segments of an Assembly language program

An Assembly language instruction consists of four fields:

[label:] mnemonic [operands] [;comment]

In Assembly language statements such as

ADD AL,BL

MOV AX,6764

ADD and MOV are the mnemonic opcodes and "AL,BL" and "AX,6764" are the operands. Instead of a mnemonic and operand, these two fields could contain assembler pseudo-instructions, or directives.

Code simple Assembly language instructions

Code control transfer instructions

In the sequence of instructions to be executed, it is often necessary to transfer program control to a different location. There are many instructions in the 8086 to achieve this.

FAR and NEAR

If control is transferred to a memory location within the current code segment, it is NEAR. This is sometimes called intrasegment (within segment). If control is transferred outside the current code segment, it is a FAR or intersegment(between segments) jump.

Conditional jumps

Conditional jumps, have mnemonics such as JNZ (Jump not zero) and JC (Jump if carry). In the conditional jump, control is transferred to a new location if a certain condition is met.

Mnemonic	Condition Tested	“Jump IF ...”
JA/JNBE	$(CF = 0) \text{ and } (ZF = 0)$	above/not below nor zero
JAЕ/JNB	$CF = 0$	above or equal/not below
JB/JNAE	$CF = 1$	below/not above nor equal
JBE/JNA	$(CF \text{ or } ZF) = 1$	below or equal/not above
JC	$CF = 1$	carry
JE/JZ	$ZF = 1$	equal/zero
JG/JNLE	$((SF \text{ xor } OF) \text{ or } ZF) = 0$	greater/not less nor equal
JGE/JNL	$(SF \text{ xor } OF) = 0$	greater or equal/not less
JL/JNGE	$(SF \text{ xor } OF) = 1$	less/not greater nor equal
JLE/JNG	$((SF \text{ xor } OF) \text{ or } ZF) = 1$	less or equal/not greater
JNC	$CF = 0$	not carry
JNE/JNZ	$ZF = 0$	not equal/not zero
JNO	$OF = 0$	not overflow
JNP/JPO	$PF = 0$	not parity/parity odd
JNS	$SF = 0$	not sign
JO	$OF = 1$	overflow
JP/JPE	$PF = 1$	parity/parity equal
JS	$SF = 1$	sign

Lab 4

Numbering Systems

Decimal System

Most people today use decimal representation to count. In the decimal system there are 10 digits:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

These digits can represent any value, for example:
754.

The value is formed by the sum of each digit, multiplied by the **base** (in this case it is **10** because there are 10 digits in decimal system) in power of digit position (counting from zero):

$$7 \cdot 10^2 + 5 \cdot 10^1 + 4 \cdot 10^0 = 700 + 50 + 4 = 754$$

base
digit position

Position of each digit is very important! for example if you place "7" to the end:

547

it will be another value:

$$5 \cdot 10^2 + 4 \cdot 10^1 + 7 \cdot 10^0 = 500 + 40 + 7 = 547$$

base
digit position

Important note: any number in power of zero is 1, even zero in power of zero is 1:

$10^0 = 1$	$0^0 = 1$	$x^0 = 1$
------------	-----------	-----------

Binary System

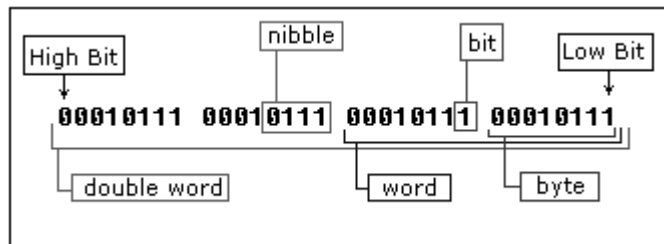
Computers are not as smart as humans are, it's easy to make an electronic machine with two states: **on** and **off**, or **1** and **0**.

Computers use binary system, binary system uses 2 digits:

0, 1

And thus the **base** is **2**.

Each digit in a binary number is called a **BIT**, 4 bits form a **NIBBLE**, 8 bits form a **BYTE**, two bytes form a **WORD**, two words form a **DOUBLE WORD** (rarely used):



There is a convention to add "**b**" in the end of a binary number, this way we can determine that 101b is a binary number with decimal value of 5.

The binary number **10100101b** equals to decimal value of 165:

$$\begin{aligned}
 10100101b &= \\
 &= 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\
 &= 128 + 0 + 32 + 0 + 0 + 4 + 0 + 1 = 165 \quad (\text{decimal value})
 \end{aligned}$$

The diagram also includes labels: 'base' points to the '2' in the powers of 2, and 'digit position' points to the superscripted powers of 2.

Octal System

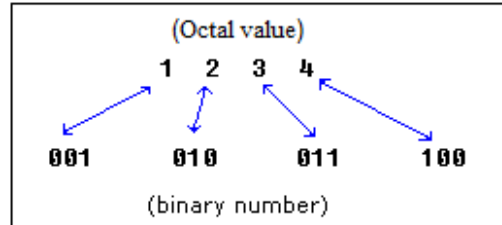
Octal System uses 8 digits:

0, 1, 2, 3, 4, 5, 6, 7

And thus the **base** is **8**.

Octal numbers are compact and easy to read. It is very easy to convert numbers from binary system to Octal system and vice-versa, every 3 bits can be converted to a Octal digit using this table:

Decimal (base 10)	Binary (base 2)	Octal (base 8)
0	000	0
1	001	1
2	010	2
3	011	3
4	100	4
5	101	5
6	110	6
7	111	7



Hexadecimal System

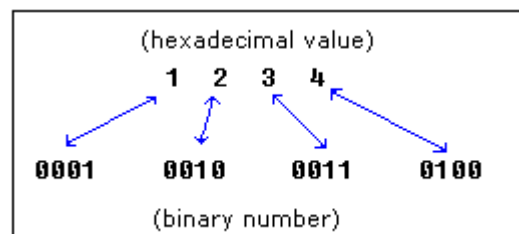
Hexadecimal System uses 16 digits:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

And thus the **base** is **16**.

Hexadecimal numbers are compact and easy to read. It is very easy to convert numbers from binary system to hexadecimal system and vice-versa, every nibble (4 bits) can be converted to a hexadecimal digit using this table:

Decimal (base 10)	Binary (base 2)	Hexadecimal (base 16)
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


There is a convention to add "**h**" in the end of a hexadecimal number, this way we can determine that 5Fh is a hexadecimal number with decimal value of 95.

We also add "**0**" (zero) in the beginning of hexadecimal numbers that begin with a letter (A..F), for example **0E120h**.

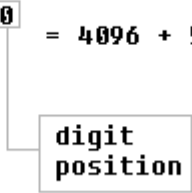
The hexadecimal number **1234h** is equal to decimal value of 4660:

$$1 \cdot 16^3 + 2 \cdot 16^2 + 3 \cdot 16^1 + 4 \cdot 16^0 = 4096 + 512 + 48 + 4 = 4660$$

(decimal value)



base



digit
position

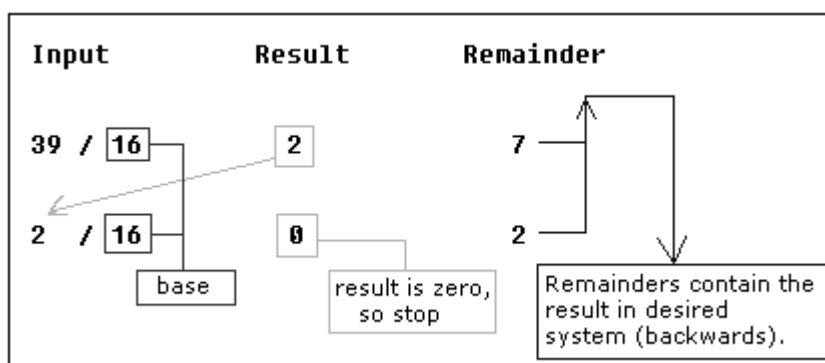
Lab 5

Numbering Systems (2)

Converting from Decimal System to Any Other

In order to convert from decimal system, to any other system, it is required to divide the decimal value by the **base** of the desired system, each time you should remember the **result** and keep the **remainder**, the divide process continues until the **result** is zero.

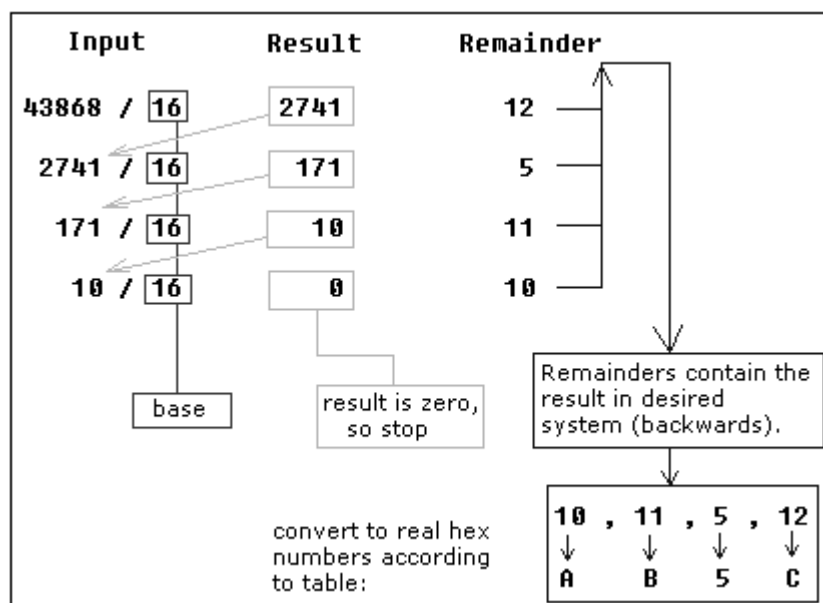
The **remainders** are then used to represent a value in that system. Let's convert the value of **39** (base 10) to *Hexadecimal System* (base 16):



As you see we got this hexadecimal number: **27h**. All remainders were below **10** in the above example, so we do not use any letters.

Here is another more complex example:

let's convert decimal number **43868** to hexadecimal form:



The result is **0AB5Ch**, we are using the table to convert remainders over **9** to corresponding letters.

Using the same principle we can convert to binary form (using **2** as the divider), or convert to hexadecimal number, and then convert it to binary number using the table:

A	B	5	C
↙	↙	↙	↙
1010	1011	0101	1100

As you see we got this binary number: **1010101101011100b**

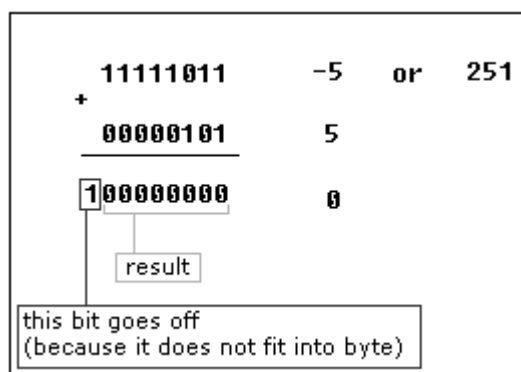
Signed Numbers

There is no way to say for sure whether the hexadecimal byte **0FFh** is positive or negative, it can represent both decimal value "**255**" and "**- 1**".

8 bits can be used to create **256** combinations (including zero), so we simply presume that first **128** combinations (**0..127**) will represent positive numbers and next **128** combinations (**128..256**) will represent negative numbers.

In order to get "**- 5**", we should subtract **5** from the number of combinations (**256**), so it we'll get: **256 - 5 = 251**.

Using this complex way to represent negative numbers has some meaning, in math when you add "**- 5**" to "**5**" you should get zero. This is what happens when processor adds two bytes **5** and **251**, the result gets over **255**, because of the overflow processor gets zero!



When combinations **128..256** are used the high bit is always **1**, so this maybe used to determine the sign of a number.

The same principle is used for **words** (16 bit values), 16 bits create **65536** combinations, first 32768 combinations (**0..32767**) are used to represent positive numbers, and next 32768 combinations (**32767..65535**) represent negative numbers.

Lab 6

Data Types in Assembly Language

In this lab, you will learn how to deal with the types of data.

- Integer
-5,0,4,100
- Character
'A','b','2','#'
- String
'Hello, World'

Variables

Variable is a memory location. For a programmer it is much easier to have some value be kept in a variable named "**var1**" then at the address 5A73:235B, especially when you have 10 or more variables.

Our compiler supports two types of variables: **BYTE** and **WORD**.

name **DB** value

name **DW** value

DB - stays for Define Byte.

DW - stays for Define Word.

name - can be any letter or digit combination, though it should start with a letter. It's possible to declare unnamed variables by not specifying the name (this variable will have an address but no name).

value - can be any numeric value in any supported numbering system (hexadecimal, binary, or decimal), or "?" symbol for variables that are not initialized.

ORG 100h

MOV AL, var1

MOV BX, var2

RET ; stops the program.

var1 DB 7

var2 DW 1234h

Arrays

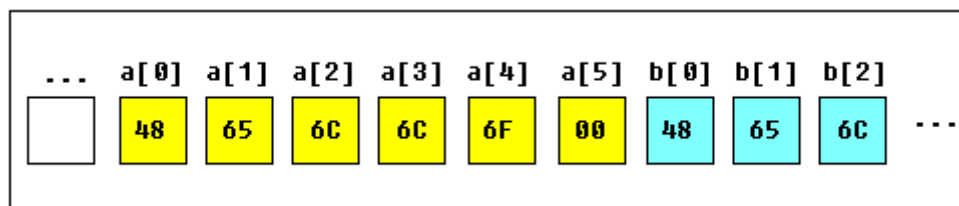
Arrays can be seen as chains of variables. A text string is an example of a byte array, each character is presented as an ASCII code value (0..255).

Here are some array definition examples:

```
a DB 48h, 65h, 6Ch, 6Ch, 6Fh, 00h
```

```
b DB 'Hello', 0
```

b is an exact copy of the *a* array, when compiler sees a string inside quotes it automatically converts it to set of bytes. This chart shows a part of the memory where these arrays are declared:



you can access the value of any element in array using square brackets, for example:

```
MOV AL, a[3]
```

You can also use any of the memory index registers **BX**, **SI**, **DI**, **BP**, for example:

```
MOV SI, 3
```

```
MOV AL, a[SI]
```

If you need to declare a large array you can use **DUP** operator.

The syntax for **DUP**:

number **DUP** (value(s))

number - number of duplicate to make (any constant value).

value - expression that DUP will duplicate.

for example:

```
c DB 5 DUP(9)
```

is an alternative way of declaring:

```
c DB 9, 9, 9, 9, 9
```

one more example:

```
d DB 5 DUP(1, 2)
```

is an alternative way of declaring:

```
d DB 1, 2, 1, 2, 1, 2, 1, 2, 1, 2
```

Constants

Constants are just like variables, but they exist only until your program is compiled (assembled). After definition of a constant its value cannot be changed. To define constants **EQU** directive is used:

name EQU < any expression >

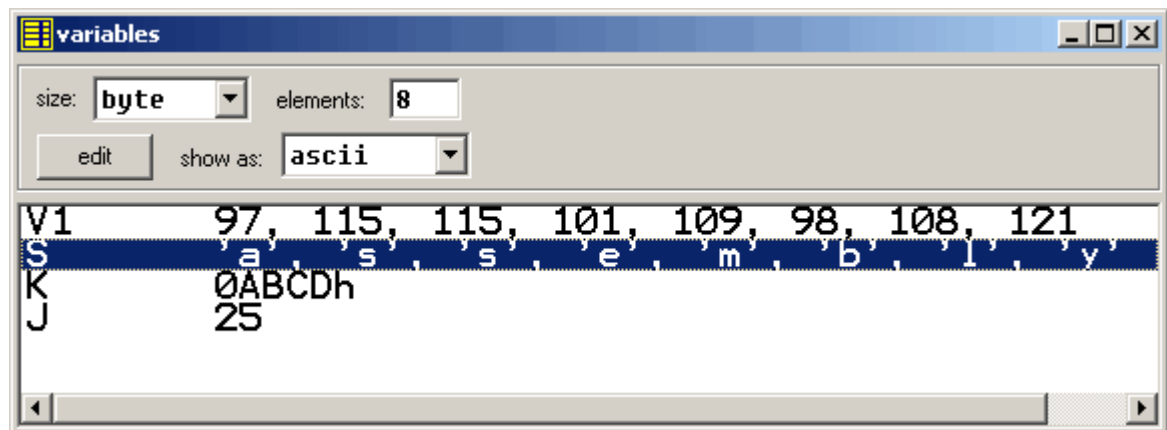
For example:

```
k EQU 5  
MOV AX, k
```

The above example is functionally identical to code:

```
MOV AX, 5
```

You can view variables while your program executes by selecting **"Variables"** from the **"View"** menu of emulator.



To view arrays you should click on a variable and set **Elements** property to array size.

In assembly language there are not strict data types, so any variable can be presented as an array.

Lab 7

Notes

Note:

The Numbering system have:

Numbering system	Stat	Example
decimal	without any letters	12
hexadecimal	h	0 ch
octal	O	14 O
binary	b	1100 b
when start as character	use 0 in start	0ch
0	writ 0 only	0

Loop instruction

Decrease CX, jump to label if CX not zero.

Algorithm:

- CX = CX - 1
- if CX \neq 0 then
 - jump
- else
 - no jump, continue

Example:

```
ORG 100h
    MOV CX, 5
label1: PRINTN 'loop!'
    LOOP label1
RET
```

Previous labs programs in assembly language

Write an assembly language program that put number 5 in Registrar AX?

```
org 100h
    MOV AX,5      ;Copy 5 to AX.
Ret
```

Write an assembly language program that calculates the sum of two numbers, such as: 5 +6?

```
org 100h
    MOV AX,5      ;Copy 5 to AX.
    ADD AX,6      ;Such as AX=AX+6.
ret
```

Assembly language program to find summation 1 ten times?

```
org 100h
    MOV AX,0000
    MOV CX,10
aa: ADD AX,1
    loop aa
ret
```

**Do in Assembly language program: $AX \leftarrow 0ah$
 $BX \leftarrow 5h$**

```
org 100h
    MOV AX,0ah
    MOV BX,5h
ret
```

Assembly language program to find summation the numbers from 1 to 10?

```
org 100h
    MOV AX,0000
    MOV CX,10
aa: ADD AX,CX
    loop aa
ret
```

Do in Assembly language program: $AX \leftarrow 11$ in decimal

$BX \leftarrow 11$ in binary

$CX \leftarrow 11$ in octal

$DX \leftarrow 11$ in hexadecimal

```
org 100h
    MOV AX,11      ;decimal
    MOV BX,01011B  ;binary
    MOV CX,130     ;octal
    MOV DX,0BH     ;hexadecimal
ret
```

Explains four fields of assembly language?

```
org 100h
;My Program
CC: MOV AX,5H      ;Put 5 in AX.
ret
```

Write an assembly language program that calculates the multiplying two numbers, such as $5 * 6$ using instructions (MOV, ADD, LOOP) only?

```
org 100h
    MOV AX,0000
    MOV CX,5
    MOV BX,6
cc:  ADD AX,BX
    loop cc
ret
```

Print the letter 'A' on the screen?

```
org 100h

MOV AL,'A'
MOV AH,0Eh
INT 10h

ret
```

Print the letter 'E' on the screen Ten times?


```
org 100h

MOV AL, 'E'
MOV CX, 10

MOV AH, 0Eh
PP: INT 10h
LOOP PP

ret
```

Write program prints letters in a sequence on the screen?

ABC...XYZ

```
org 100h

MOV AL, 65 ;the ascii code of 'A'
MOV CX, 26 ;number of letters
MOV AH, 0Eh
PP: INT 10h
ADD AL, 1
LOOP PP

Ret
```

Lab 8

Registers of the 80x86 Microprocessor

Category	Bits	Register Names
General	16	AX, BX, CX, DX
	8	AH, AL, BH, BL, CH, CL, DH, DL
Pointer	16	SP (stack pointer), BP (base pointer)
Index	16	SI (source index), DI (destination index)
Segment	16	CS (code segment), DS (data segment), SS (stack segment), ES (extra segment)
Instruction	16	IP (instruction pointer)
Flag	16	FR (flag register)

The 80x86 family, like other processors, has both general and special purpose registers available to the assembly language programmer. These registers can be loosely classified into Data Registers, Address Registers, and Status Registers.

8-bit Registers

- AH, AL, BH, BL, CH, CL, DH, DL
- Any of these registers can be used as an 8-bit operand.

16-bit Registers

- AX, BX, CX, DX, SI, DI, SP, BP
- Any of these registers can be used as a 16-bit operand.

General Data Registers

Special Register Functions

- AX - Accumulator register
 - Generates shortest machine code
 - Arithmetic, logic and data transfer
 - One number must be in AL or AX
 - Multiplication & Division
 - Input & Output
- BX - Base Register Addressing (Pointer)
- CX - Counter Register
 - Iterative code segments using the LOOP instruction
 - Repetitive operations on strings with the REP command

- Count (in CL) of bits to shift and rotate
- DX - Data Register (Arithmetic)
 - DX:AX concatenated into 32-bit register for some MUL and DIV operations
 - Specifying ports in some IN and OUT operations

Segmentation and Segment Registers

- 8086 Address Space
 - The 8086 processor has a 20-bit physical address to directly address 1M byte of memory
 - Word size is only 16 bits
 - 20-bit address is splitted in 16-bit segment address and 16-bit offset address
 - Segment address shifted four bits to the left and added to the offset value to generate a 20-bit effective address
 - Effective address is expressed as a 5 digit hex value (00000h to FFFFFh)
- Memory Segments
 - A segment is a block of 64K consecutive memory bytes
 - Segments are identified by segment numbers 0 - FFFFh
 - A 16 byte block makes up a paragraph
 - Segments always start on paragraph boundaries
 - Least significant nibble of segment address will always be 0
- Segment Registers - CS, DS, SS, ES
 - Always point to low address end of segment
 - CS - Code Segment Register - points to a segment containing code
 - DS - Data Segment Register - points to a segment containing data
 - SS - Stack Segment Register - points to a segment containing stack
 - ES - Extra Segment register - points to a segment containing data

Pointer and Index Registers - SP, BP, SI, DI

- SP - Stack Pointer
 - Always points to top item on the stack

- Offset address relative to SS
 - Always points to word (byte at even address)
 - An empty stack will had SP = FFFEH
- BP - Base pointer
 - Primarily used to access parameters passed via the stack
 - Offset address relative to SS
- SI - Source Index
 - Can be used for pointer addressing of data
 - Used as source in some string processing instructions
 - Offset address relative to DS
- DI - Destination Index
 - Can be used for pointer addressing of data
 - Used as destination in some string processing instructions
 - Offset address relative to ES
- IP - Instruction Pointer
 - Always points to next instruction to be executed
 - Offset address relative to CS

Flag Register

- Status is indicated with individual bits:
 - 0 - CF - Carry Flag
 - 2 - PF - Parity Flag
 - 4 - AF - Auxiliary carry Flag
 - 6 - ZF - Zero Flag
 - 7 - SF - Sign Flag
 - 8 - TF - Trap Flag
 - 9 - IF - Interrupt Flag
 - 10 - DF - Direction Flag
 - 11 - OF - Overflow Flag
- Flag bits are set by instructions
- Flag bits are basis of conditional jump instructions

The 16 bits of the flag register:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	R	R	R	OF	DF	IF	TF	SF	ZF	U	AF	U	PF	U	CF

R = reserved

U = undefined

OF = overflow flag

DF = direction flag

IF = interrupt flag

TF = trap flag

SF = sign flag

ZF = zero flag

AF = auxiliary carry flag

PF = parity flag

CF = carry flag

Write program summation two numbers in the two variables.

```
org 100h
MOV AL,VAR1
ADD AL,VAR2
ret
VAR1 DB 5
VAR2 DB 6
```

Write program summation the numbers in the array: (2,4,6,8,10).

```
org 100h
MOV CX,5
MOV SI,0
MOV AL,0
AA: ADD AL,a[SI]
ADD SI,1
LOOP AA
ret
a DB 2,4,6,8,10
```

**Write program that add 2 to all the elements of the array:
(2,4,6,8,10). Definition number 2 a constant.**

```
org 100h
MOV CX,5
MOV SI,0
AA: ADD a[SI],c
ADD SI,1
LOOP AA
MOV BL,a[0]
ret
a DB 2,4,6,8,10
c EQU 2
```