# 8086 ASSEMBLY LANGUAGE PROGRAMMING

# Chapter One

## 1.1. Introduction

## Why learning Assembly Language?

Assembly language is useful in making efficient code that consumes less number of clock cycles and takes minimum memory footprint as compared to high level language. This makes Assembly Language Programming to be used in designing software for memory and power constrained portable devices. It is also used in firmware design real-time mission critical applications, device drivers etc.

The first reason to work with assembler is that it provides the opportunity of knowing more the operation of your PC, which allows the development of software in a more consistent manner. The second reason is the total control of the PC which you can have with the use of the assembler. Another reason is that the assembly programs are quicker, smaller, and have larger capacities than ones created with other languages. Lastly, the assembler allows an ideal optimization in programs, be it on their size or on their execution.

Assembler or other languages? that is the question. Why should I learn another language, if I already learned other programming languages? The best argument: while you live in France you are able to get through by speaking English, but you will never feel at home then, and life remains complicated. You can get through with this, but it is rather inappropriate. If things need a hurry, you should use the country's language.

Many people that use higher-level languages in their daily work recommend that beginners start with learning assembly language. The reason is that sometimes, namely in the following cases:

- if bugs have to be analyzed,
- if the program executes different than designed and expected,
- if the higher-level language doesn't support the use of certain hardware features,
- if time-critical in line routines require assembly language portions,

it is necessary to understand assembly language, e. g. to understand what the higher-level language compiler produced. Without understanding assembly language you do not have a chance to proceed further in these cases.

### Short and easy

Assembler instructions translate one by one to executed machine instructions. The processor needs only to execute what you want it to do and what is necessary to perform the task. No extra loops and unnecessary features blow up the generated code. If your program storage is short and limited and you have to optimize your program to fit into memory, assembler is choice 1. Shorter programs are easier to debug, every step makes sense.

**Fast and quick**

Because only necessary code steps are executed, assembly programs are as fast as possible. The duration of every step is known. Time critical applications, like time measurements without a hardware timer, that should perform excellent, must be written in assembler. If you have more time and don't mind if your chip remains 99% in a wait state type of operation, you can choose any language you want.

**Assembler is easy to learn**

It is not true that assembly language is more complicated or not as easy to understand than other languages. Learning assembly language for whatever hardware type brings you to understand the basic concepts of any other assembly language dialects. Adding other dialects later is easy. As some features are hardware-dependent optimal code requires some familiarity with the hardware concept and the dialect. What makes assembler sometimes look complicated is that it requires an understanding of the controller's hardware functions. Consider this an advantage: by learning assembly language you simultaneously learn more about the hardware. Higher level languages often do not allow you to use special hardware features and so hide these functions.

The first assembly code does not look very attractive, with every 100 additional lines programmed it looks better. Perfect programs require some thousand lines of code of exercise, and optimization requires lots of work. The first steps are hard in any language. After some weeks of programming you will laugh if you go through your first code. Some assembler instructions need some months of experience.

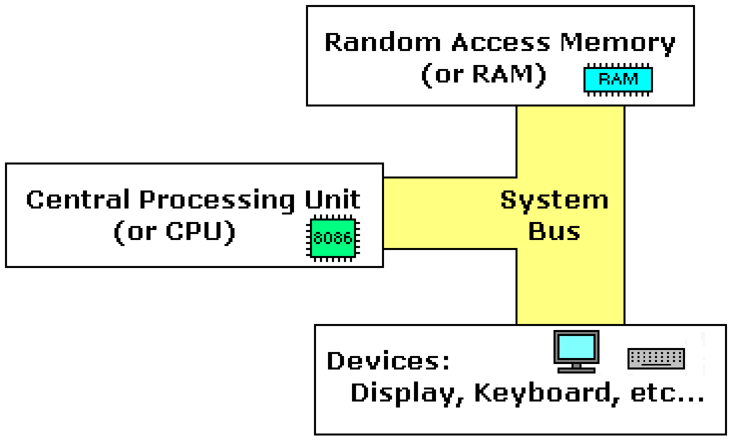
**History**

Intel’s first 16-bit CPU was the 8086. A version of the 8086 that used an 8-bit data bus, the 8088, was released later to permit lower-cost designs. The 8088 was used in the very popular IBM PC and many later compatible machines. Intel’s first 32-bit CPU was the 80386. It was designed to be backwards compatible with the large amount of software, which was available for the 8086. The 80386 extended the data and address registers to 32 bits. The Intel ’386 also included a sophisticated memory management architecture that allowed *virtual memory* and *memory protection* to be implemented. This same basic 80386 architecture is used in the Pentium series and compatible processors.

**What is an assembly language?**
Assembly language is a low level programming language. It is expressed as a combination of letters rather than binary. These letters are called mnemonics, enabling the programmers remember the operation codes.

## 1.2 Basic Computer Architecture
The basic operational design of a computer system is called its architecture. John Von Neumann, a pioneer in computer design, is given credit for the architecture of most computers in use today. For example, the 80x86 family uses the Von Neumann architecture (VNA). A typical Von Neumann system has three major components: the central processing unit (or CPU), memory, and input/output (or I/O). The way a system designer combines these components impacts system performance. In VNA machines, like the 80x86 family, the CPU is where all the action takes place. All computations occur inside the CPU. Data and CPU instructions reside in memory until required by the CPU. To the CPU, most I/O devices look like memory because the

CPU can store data to an output device and read data from an input device. The major difference between memory and I/O locations is the fact that I/O locations are generally associated with external devices in the outside world.



The system bus  connects the various components of a computer. The CPU is the heart of the computer - most computations occur inside the CPU. RAM is a place to where the programs are loaded in order to be executed.

**The System Bus**

The system bus connects the various components of a VNA machine. The 80x86 family has three major buses: the address bus, the data bus, and the control bus. A bus is a collection of wires on which electrical signals pass between components in the system. These buses vary from processor to processor. However, each bus carries comparable information on all processors; e.g., the data bus may have a different implementation on the 80386 than on the 8088, but both carry data between the processor, I/O, and memory. A typical 80x86 system component uses standard TTL logic levels. This means each wire on a bus uses a standard voltage level to represent zero and one1. We will always specify zero and one rather than the electrical levels because these levels vary on different processors (especially laptops).

**The Data Bus**

The 80x86 processors use the data bus to shuffle data between the various components in a computer system. The size of this bus varies widely in the 80x86 family. Indeed, this bus defines the “size” of the processor. On typical 80x86 systems, the data bus contains 8, 16, 32, or 64 lines. The 8088 and 80188 microprocessors have an 8-bit data bus (eight data lines). The 8086, 80186, 80286, and 80386SX processors have a 16-bit data bus. The 80386DX, 80486, and Pentium Overdrive Ô processors have a 32-bit data bus. The Pentium Ô and Pentium Pro processors have a 64-bit data bus. Future versions of the chip (the 80686/80786) may have a larger bus. Having an 8-bit data bus does not limit the processor to 8-bit data types. It simply means that the processor can only access one byte of data per memory cycle. Therefore, the eight bit bus on an 8088 can only transmit half the information per unit time (memory cycle) as the 16 bit bus on the 8086. Therefore, processors with a 16 bit bus are naturally faster than processors with an eight bit bus. Likewise, processors with a 32 bit bus are faster than those with a 16 or eight bit data bus. The size of the data bus affects the performance of the system more than the size of any other bus. Data moves from both, processor to memory and memory to processor, so the data bus is bidirectional.

**80x86 Processor Data Bus Sizes**

| Processor | Data Bus Size |
|---|---|
| 8088 | 8 |
| 80188 | 8 |
| 8086 | 16 |
| 80186 | 16 |
| 80286 | 16 |
| 80386sx | 16 |
| 80386dx | 32 |
| 80486 | 32 |
| 80586 class/ Pentium (Pro) | 64 |

## The Address Bus

The data bus on an 80x86 family processor transfers information between a particular memory location or I/O device and the CPU. The only question is, "*Which memory location or I/O device*. The address bus answers that question. To differentiate memory locations and I/O devices, the system designer assigns a unique memory address to each memory element and I/O device. When the software wants to access some particular memory location or I/O device, it places the corresponding address on the address bus. Circuitry associated with the memory or I/O device recognizes this address and instructs the memory or I/O device to read the data from or place data on the data bus. In either case, all other memory locations ignore the request. Only the device whose address matches the value on the address bus responds. With a single address line, a processor could create exactly two unique addresses: zero and one. With *n* address lines, the processor can provide $2^n$ unique addresses (since there are $2^n$ unique values in an *n*-bit binary number). Therefore, the number of bits on the address bus will determine the *maximum* number of addressable memory and I/O locations. The 8088 and 8086, for example, have 20-bit address bus. Therefore, they can access up to 1,048,576 (or $2^{20}$) memory locations. Larger address buses can access more memory. The 8088 and 8086, for example, suffer from an anemic address space – their address bus is too small. However, later processors have larger address buses. The address bus is unidirectional and address always travels from processor to memory. This is because memory is a dumb device and cannot predict which element the processor at a particular instant of time needs.

## The Control Bus

The control bus is an eclectic collection of signals that control how the processor communicates with the rest of the system. Consider for a moment the data bus. The CPU sends data to memory and receives data from memory on the data bus. This prompts the question, "Is it sending or receiving?" There are two lines on the control bus, read and write, which specify the direction of data flow. The control bus carries the intent of the processor that it wants to read or to write. Memory changes its behavior in response to this signal from the processor. It defines the direction of data flow. If processor wants to read but memory wants to write, there will be no communication or useful flow of information. Both must be synchronized, like a speaker speaks and the listener listens. If both speak simultaneously or both listen there will be no communication. This precise synchronization between the processor and the memory is the responsibility of the control bus. Control bus is special and relatively complex, because different lines comprising it behave differently. It is only the mechanism because the responsibility of sending the appropriate signals on the control bus to the memory is of the processor. Since the memory never wants to listen or to speak of itself. Why then is the control bus bidirectional?

Consider a scenario where we sent a servant to fetch a particular book in a shelf but he found that the room where it is placed has been locked. Now the servant can wait there indefinitely keeping us in surprise or come back and inform us about the situation so that we can act accordingly. The servant, even though he was obedient was unable to fulfill our orders so in all his obedience, he came back to inform us about the problem. Synchronization is still important, as a result of our orders either we got the desired cell or we came to know that the memory is locked for the moment. Such information cannot be transferred via the address or the data bus. For such situations when peripherals want to talk to the processor when the processor wasn't expecting them to speak, special lines in the control bus are used. The information in such signals is usually to indicate the incapability of the peripheral to do something for the moment. For these reasons the control bus is a bidirectional bus and can carry information from processor to memory as well as from memory to processor. Other signals include system clocks, interrupt lines, status lines, and so on. The exact make up of the control bus varies among processors in the 80x86 family.

**CPU Registers**

The basic purpose of a computer is to perform operations, and operations need operands. Operands are the data on which we want to perform a certain operation. Consider the addition operation; it involves adding two numbers to get their sum. We can have precisely one address on the address bus and consequently precisely one element on the data bus. At the very same instant the second operand cannot be brought inside the processor. As soon as the second is selected, the first operand is no longer there. For this reason there are temporary storage places inside the processor called registers. Now one operand can be read in a register and added into the other which is read directly from the memory. Both are made accessible at one instance of time, one from inside the processor and one from outside on the data bus. The result can be written to at a distinct location as the operation has completed and we can access a different memory cell. Sometimes we hold both operands in registers for the sake of efficiency as what we can do inside the processor is undoubtedly faster than if we have to go outside and bring the second operand. Registers are like a scratch pad ram inside the processor and their operation is very much like normal memory cells. They have precise locations and remember what is placed inside them. They are used when we need more than one data element inside the processor at one time.

The 8086 CPU has 14 registers and each register has its own name

| |
|---|
| CS |
| DS |
| SS |
| ES |
| |
| IP |
| |
| FLAGS |

| | |
|---|---|
| SP | |
| BP | |
| SI | |
| DI | |
| AH | AL |
| BH | BL |
| CH | CL |

| | |
|---|---|
| DH | DL |

(AX)
(BX)
(CX)
(DX)

The following are the 8 general-purpose registers

- **AX** - the accumulator register (divided into **AH / AL**).
- **BX** - the base address register (divided into **BH / BL**).
- **CX** - the count register (divided into **CH / CL**).
- **DX** - the data register (divided into **DH / DL**).
- **SI** - source index register.
- **DI** - destination index register.
- **BP** - base pointer.
- **SP** - stack pointer.

Despite the name of a register, it's the programmer who determines the usage for each general-purpose register. The main purpose of a register is to keep a number (variable). The size of the above registers is 16 bits, it's something like: **0011000000111001b** (in binary form), or **12345** in decimal (human) form.

The registers AX, BX, CX, and DX behave as general purpose registers in Intel architecture and do some specific functions in addition to it. X in their names stand for extended meaning 16bit registers. For example AX means we are referring to the extended 16bit “A” register. Its upper and lower byte are separately accessible as AH (A high byte) and AL (A low byte). All general purpose registers can be accessed as one 16bit register or as two 8bit registers. The two registers AH and AL are part of the big whole AX. Any change in AH or AL is reflected in AX as well. AX is a composite or extended register formed by gluing together the two parts AH and AL. The A of AX stands for Accumulator. Even though all general purpose registers can act as accumulator in most instructions there are some specific variations which can only work on AX which is why it is named the accumulator. The B of BX stands for Base because of its role in memory addressing. The C of CX stands for Counter as there are certain instructions that work with an automatic count in the CX register. The D of DX stands for Destination as it acts as the destination in I/O operations. The location of registers inside the CPU makes them much faster than memory which requires a system bus to access it and this takes much longer. Accessing data in a register usually takes no time. Therefore, you should try to keep variables in the registers. Register sets are very small and most registers have special purposes which limit their use as variables, but they are still an excellent place to store temporary data of calculations.

Each register also has a special purpose as shown below:

| Register | Special Purpose |
|---|---|
| AX | Multiply/Divide |
| BX | Index register for MOVE |
| CX | Count register for string operations |

| | |
|---|---|
| DX | Port address for IN and OUT |

**Index Registers (SI and DI)**
SI and DI stand for source index and destination index respectively. These are the index registers of the Intel architecture which hold address of data and used in memory access. Being an open and flexible architecture, Intel allows many mathematical and logical operations on these registers as well like the general registers. The source and destination are named because of their implied functionality as the source or the destination in a special class of instructions called the string instructions. However their use is not at all restricted to string instructions. SI and DI are 16bit and cannot be used as 8bit register pairs like AX, BX, CX, and DX.

**Stack Pointer (SP)**
It is a memory pointer and is used indirectly by a set of instructions. It has a very special purpose – it maintains the *program stack*. Normally, you would not use this register for arithmetic computations. The proper operation of most programs depends upon the careful use of this register.

**Base Pointer (BP)**
It is the bx register. You'll generally use this register to access parameters and local variables in a procedure.

**SEGMENT REGISTERS**
Although it is possible to store any data in the segment registers, this is never a good idea. The segment registers have a very special purpose, i.e. pointing at accessible blocks of memory. Segment registers work together with general purpose registers to access any memory value. For example if we would like to access memory at the physical address **12345h** (hexadecimal), we should set the **DS=1230h** and **SI = 0045h**. This is good, since this way we can access much more memory than with a single register that is limited to 16-bit values.
CPU makes a calculation of physical address by multiplying the segment register by 10h and adding general purpose register to it (1230h* 10h + 45h = 12345h):

$$\begin{array}{r} \mathbf{12300} \\ \underline{\mathbf{+\ 0045}} \\ \underline{\mathbf{12345}} \end{array}$$

The address formed with 2 registers is called an **effective address**. By default **BX, SI** and **DI** registers work with **DS** segment register; **BP** and **SP** work with **SS** segment register. Other general purpose registers cannot form an effective address! Also, although **BX** can form an effective address, **BH** and **BL** cannot!
- CS - points at the segment containing the current program.
- DS - generally points at segment where variables are defined.
- ES - extra segment register, it's up to a coder to define its usage.
- SS - points at the segment containing the stack.

**SPECIAL PURPOSE REGISTERS**
- IP - the instruction pointer. This is the special register containing the address of the next instruction to be executed. No mathematics or memory access can be done through this register. It is out of our direct control and is automatically used. Playing with it is dangerous and needs special care. Program control instructions change the IP register.

- Flags Register - determines the current state of the processor. The flags register is not meaningful as a unit rather it is bit-wise significant and accordingly each bit is named separately. The bits not named are unused. The Intel flags register has its bits organized as follows:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  |  |  |  | O | D | I | T | S | Z |  | A |  | P |  | C |

The individual flags are explained in the following table.

|  |  |  |
|---|---|---|
| C | Carry | When two 16bit numbers are added the answer can be 17 bits long or when two 8bit numbers are added the answer can be 9 bits long. This extra bit that won't fit in the target register is placed in the carry flag where it can be used and tested. |
| P | Parity | Parity is the number of "one" bits in a binary number. Parity is either odd or even. This information is normally used in communications to verify the integrity of data sent from the sender to the receiver. |
| A | Auxiliary Carry | A number in base 16 is called a hex number and can be represented by 4 bits. The collection of 4 bits is called a nibble. During addition or subtraction if a carry goes from one nibble to the next this flag is set. Carry flag is for the carry from the whole addition while auxiliary carry is the carry from the first nibble to the second. |
| Z | Zero Flag | The Zero flag is set if the last mathematical or logical instruction has produced a zero in its destination. |
| S | Sign Flag | A signed number is represented in its two's complement form in the computer. The most significant bit (MSB) of a negative number in this representation is 1 and for a positive number it is zero. The sign bit of the last mathematical or logical operation's destination is copied into the sign flag. |
| T | Trap Flag | The trap flag has a special role in debugging which will be discussed later. |
| I | Interrupt Flag | It tells whether the processor can be interrupted from outside or not. Sometimes the programmer doesn't want a particular task to be interrupted so the Interrupt flag can be zeroed for this time. The programmer rather than the processor sets this flag since the programmer knows when interruption is okay and when it is not. Interruption can be disabled or enabled by making this bit zero or one, respectively, using special instructions. |
| D | Direction Flag | Specifically related to string instructions, this flag tells whether the current operation has to be done from bottom to top of the block (D=0) or from top to bottom of the block (D=1). |
| O | Overflow Flag | The overflow flag is set during signed arithmetic, e.g. addition or subtraction, when the sign of the destination changes unexpectedly. The actual process sets the overflow flag whenever the carry into the MSB is different from the carry out of the MSB |

IP register always works together with CS segment register and it points to currently executing instruction. Flags Register is modified automatically by CPU after mathematical operations, this allows to determine the type of the result, and to determine conditions to transfer control to other parts of the program. Generally you cannot access these registers directly.

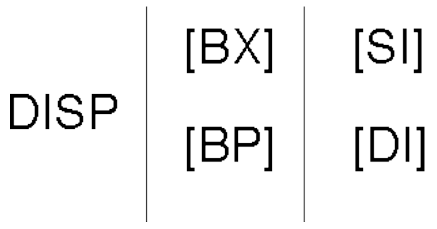
# Chapter Two

## Memory Access

To access memory we can use these four registers: **BX, SI, BP, DI**. Combining these registers inside **[ ]** symbols, we can get 17 different memory locations. These combinations are supported (addressing modes):

| | | |
|---|---|---|
| [BX + SI]<br>[BX + DI]<br>[BP + SI]<br>[BP + DI] | [SI]<br>[DI]<br>d16 (variable offset only)<br>[BX] | [BX + SI] + d8<br>[BX + DI] + d8<br>[BP + SI] + d8<br>[BP + DI] + d8 |
| [SI] + d8<br>[DI] + d8<br>[BP] + d8<br>[BX] + d8 | [BX + SI] + d16<br>[BX + DI] + d16<br>[BP + SI] + d16<br>[BP + DI] + d16 | [SI] + d16<br>[DI] + d16<br>[BP] + d16<br>[BX] + d16 |

**d8** - stands for 8 bit displacement.
**d16** - stands for 16 bit displacement.
Displacement can be an immediate value or offset of a variable, or even both. It's up to compiler to calculate a single immediate value.
- Displacement can be inside or outside of **[ ]** symbols, compiler generates the same machine code for both ways.
- Displacement is a **signed** value, so it can be either positive or negative.
Generally the compiler takes care about difference between **d8** and **d16**, and generates the required machine code.
For example, let's assume that **DS = 100**, **BX = 30**, **SI = 70**.
The following addressing mode: **[BX + SI] + 25** is calculated by processor to this physical address: **100 * 16 + 30 + 70 + 25 = 1725**.
By default **DS** segment register is used for all modes except those with **BP** register, for these **SS** segment register is used. There is an easy way to remember all those possible combinations using this chart:



There are a total of 17 different legal memory addressing modes on the 8086: disp, [bx], [bp], [si], [di], disp[bx], disp[bp], disp[si], disp[di], [bx][si], [bx][di], [bp][si], [bp][di], disp[bx][si], disp [bx][di], disp[bp][si], and disp[bp][di]. You could memorize all these forms so that you know which are valid (and, by omission, which forms are invalid). However, there is an easier way besides memorizing these 17 forms. If you choose zero or one item from each of the columns and wind up with at least one item, you've got a valid 8086 memory addressing mode. For example:

- Choose disp from column one, nothing from column two, [di] from column 3, you get disp[di].
- Choose disp, [bx], and [di]. You get disp[bx][di].
- Skip column one & two, choose [si]. You get [si]

- Skip column one, choose [bx], then choose [di]. You get [bx][di]

Likewise, if you have an addressing mode that you cannot construct from this table, then it is not legal. For example, disp[dx][si] is illegal because you cannot obtain [dx] from any of the columns above. As you see **BX** and **BP** never go together. **SI** and **DI** also don't go together. Here is an example of a valid addressing mode: **[BX+5]**.

The value in segment register (CS, DS, SS, ES) is called a "**segment**", and the value in purpose register (BX, SI, DI, BP) is called an "**offset**". When DS contains value **1234h** and SI contains the value **7890h** it can be also recorded as **1234:7890**. The physical address will be 1234h * 10h + 7890h = 19BD0h.

The *effective address* is the final offset produced by an addressing mode computation. For example, if bx contains 10h, the effective address for 10h[bx] is 20h. You will see the term effective address in almost any discussion of the 8086's addressing mode. There is even a special instruction *load effective address* (lea) that computes effective addresses. Not all addressing modes are created equal! Different addressing modes may take differing amounts of time to compute the effective address. The exact difference varies from processor to processor. Generally, though, the more complex an addressing mode is, the longer it takes to compute the effective address. Complexity of an addressing mode is directly related to the number of terms in the addressing mode. For example, disp[bx][si] is more complex than [bx].

The displacement field in all addressing modes *except* displacement-only can be a signed eight bit constant or a signed 16 bit constant. If your offset is in the range -128…+127 the instruction will be shorter (and therefore faster) than an instruction with a displacement outside that range. The size of the value in the register does not affect the execution time or size. So if you can arrange to put a large number in the register(s) and use a small displacement, that is preferable over a large constant and small values in the register(s). If the effective address calculation produces a value greater than 0FFFFh, the CPU ignores the overflow and the result *wraps around* back to zero. For example, if bx contains 10h, then the instruction mov al, 0FFFFh[bx] will load the al register from location ds:0Fh, not from location ds:1000Fh.

## 80x86 Physical Memory Organization

Chapter One discussed the basic organization of a Von Neumann Architecture (VNA) computer system. In a typical VNA machine, the CPU connects to memory via the bus. The 80x86 selects some particular memory element using a binary number on the address bus. Another way to view memory is as an array of bytes. A Pascal data structure that roughly corresponds to memory would be:

Memory : array [0..MaxRAM] of byte;

The value on the address bus corresponds to the index supplied to this array. E.g., writing data to memory is equivalent to

Memory [address] := Value_to_Write;

Reading data from memory is equivalent to

Value_Read := Memory [address];

Different 80x86 CPUs have different address busses that control the maximum number of elements in the memory array). However, regardless of the number of address lines on the bus, most computer systems do *not* have one byte of memory for each addressable location. For example, 80386 processors have 32 address lines allowing up to four gigabytes of memory. Very few 80386 systems actually have four gigabytes. Usually, you'll find one to 256 megabytes in an 80x86 based system. The first megabyte of memory, from address zero to 0FFFFFh is special on the 80x86. This corresponds to the entire address space of the 8088, 8086, 80186, and 80188 microprocessors. Most DOS programs limit their program and data addresses to locations in this range. Addresses limited to this range are named *real addresses* after the 80x86 *real mode*.

**Segments on the 80x86**

You cannot discuss memory addressing on the 80x86 processor family without first discussing segmentation. Among other things, segmentation provides a powerful memory management mechanism. It allows programmers to partition their programs into modules that operate independently of one another. Segments provide a way to easily implement object-oriented programs. Segments allow two processes to easily share data. All in all, segmentation is a really neat feature. On the other hand, if you ask ten programmers what they think of segmentation, at least nine of the ten will claim it's terrible. Why such a response? Well, it turns out that segmentation provides one other nifty feature: it allows you to extend the addressability of a processor. In the case of the 8086, segmentation let Intel's designers extend the maximum addressable memory from 64K to one megabyte. Gee, that sounds good. Why is everyone complaining? Well, a little history lesson is in order to understand what went wrong.

In 1976, when Intel began designing the 8086 processor, memory was very expensive. Personal computers, such that they were at the time, typically had four thousand bytes of memory. Even when IBM introduced the PC five years later, 64K was still quite a bit of memory, one megabyte was a tremendous amount. Intel's designers felt that 64K memory would remain a large amount throughout the lifetime of the 8086. The only mistake they made was completely underestimating the lifetime of the 8086. They figured it would last about five years, like their earlier 8080 processor. They had plans for lots of other processors at the time, and "86" was not a suffix on the names of any of those. Intel figured they were set. Surely one megabyte would be more than enough to last until they came out with something better.

Unfortunately, Intel didn't count on the IBM PC and the massive amount of software to appear for it. By 1983, it was very clear that Intel could not abandon the 80x86 architecture. They were stuck with it, but by then people were running up against the one megabyte limit of 8086. So Intel gave us the 80286. This processor could address up to 16 megabytes of memory. Surely more than enough. The only problem was that all that wonderful software written for the IBM PC was written in such a way that it couldn't take advantage of any memory beyond one megabyte. It turns out that the maximum amount of addressable memory is not everyone's main complaint. The real problem is that the 8086 was a 16 bit processor, with 16 bit registers and 16 bit addresses. This limited the processor to addressing 64K chunks of memory.

Intel's clever use of segmentation extended this to one megabyte, but addressing more than 64K at one time takes some effort. Addressing more than 256K at one time takes a *lot* of effort.

Despite what you might have heard, segmentation is not bad. In fact, it is a really great memory management scheme. What is bad is Intel's 1976 implementation of segmentation still in use today. You can't blame Intel for this – they fixed the problem in the 80's with the release of the 80386. The real culprit is MS-DOS that forces programmers to continue to use 1976 style segmentation. Fortunately, newer operating systems such as Linux, UNIX, Windows 9x, Windows NT, and OS/2 don't suffer from the same problems as MS-DOS. Furthermore, users finally seem to be more willing to switch to these newer operating systems so programmers can take advantage of the new features of the 80x86 family.

With the history lesson aside, it's probably a good idea to figure out what segmentation is all about. Consider the current view of memory: it looks like a linear array of bytes.

A single index (address) selects some particular byte from that array. Let's call this type of addressing *linear* or *flat* addressing. Segmented addressing uses two components to specify a memory location: a segment value and an offset within that segment. Ideally, the segment and offset values are independent of one another. The best way to describe segmented addressing is with a two-dimensional array. The segment provides one of the indices into the array, the offset provides the other.

## The 80x86 Memory Addressing Modes

80x86 processors let you access memory in many different ways (In fact, the 8086 provides 17 different ways to access memory). The 80x86 memory addressing modes provide flexible access to memory, allowing you to easily access variables, arrays, records, pointers, and other complex data types. Mastery of the 80x86 addressing modes is the first step towards mastering 80x86 assembly language.

When Intel designed the original 8086 processor, they provided it with a flexible, though limited, set of memory addressing modes. Intel added several new addressing modes when it introduced the 80386 microprocessor. Note that the 80386 retained all the modes of the previous processors; the new modes are just an added bonus. If you need to write code that works on 80286 and earlier processors, you will not be able to take advantage of these new modes. However, if you intend to run your code on 80386sx or higher processors, you can use these new modes. The basic addressing modes are discussed below:

## Register Addressing Modes

Most 8086 instructions can operate on the 8086's general purpose register set. By specifying the name of the register as an operand to the instruction, you may access the contents of that register. Consider the 8086 mov (move) instruction:

```
                    mov     destination, source
```

This instruction copies the data from the source operand to the destination operand. The eight and 16 bit registers are certainly valid operands for this instruction. The only restriction is that both operands must be the same size. Now let's look at some actual 8086 mov instructions:

```
          mov   ax, bx  ;Copies the value from BX into AX
          mov   dl, al  ;Copies the value from AL into DL
          mov   si, dx  ;Copies the value from DX into SI
          mov   sp, bp  ;Copies the value from BP into SP
          mov   dh, cl  ;Copies the value from CL into DH
          mov   ax, ax  ;Yes, this is legal!
```

Remember, the registers are the best place to keep often used variables.

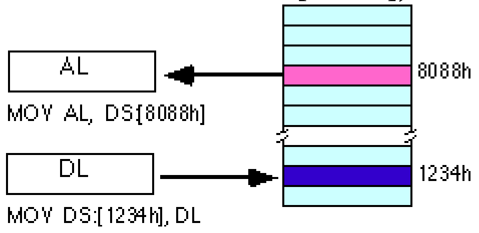
In addition to the general purpose registers, many 8086 instructions (including the mov instruction) allow you to specify one of the segment registers as an operand. There are two restrictions on the use of the segment registers with the mov instruction. First of all, you may not specify cs as the destination operand, second, only one of the operands can be a segment register. You cannot move data from one segment register to another with a single mov instruction. To copy the value of cs to ds, you'd have to use some sequence like:

```
          mov   ax, cs
          mov   ds, ax
```

You should never use the segment registers as data registers to hold arbitrary values. They should only contain segment addresses.

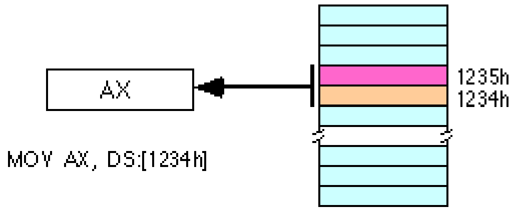
## The Displacement Only Addressing Mode

The most common addressing mode, and the one that's easiest to understand, is the displacement-only (or direct) addressing mode. The displacement-only addressing mode consists of a 16 bit constant that specifies the address of the target location. The instruction mov al,ds:[8088h] loads the al register with a copy of the byte at memory location 8088h. Likewise, the instruction mov ds:[1234h],dl stores the value in the dl register to memory location 1234h:



The displacement-only addressing mode is perfect for accessing simple variables. Of course, you'd probably prefer using names like "I" or "J" rather than "DS:[1234h]" or "DS:[8088h]". Well, fear not, you'll soon see it's possible to do just that.

Intel named this the displacement-only addressing mode because a 16 bit constant (displacement) follows the mov opcode in memory. In that respect it is quite similar to the direct addressing mode on the x86 processors (see the previous chapter). There are some minor differences, however. First of all, a displacement is exactly that- some distance from some other point. On the x86, a direct address can be thought of as a displacement from address zero. On the 80x86 processors, this displacement is an offset from the beginning of a segment (the data segment in this example). Don't worry if this doesn't make a lot of sense right now. You'll get an opportunity to study segments to your heart's content a little later in this chapter. For now, you can think of the displacement-only addressing mode as a direct addressing mode. The examples in this chapter will typically access bytes in memory. Don't forget, however, that you can also access words on the 8086 processors :



By default, all displacement-only values provide offsets into the data segment. If you want to provide an offset into a different segment, you must use a segment override prefix before your address. For example, to access location 1234h in the extra segment (es) you would use an instruction of the form mov ax,es:[1234h]. Likewise, to access this location in the code segment you would use the instruction mov ax, cs:[1234h]. The ds: prefix in the previous examples is not a segment override. The CPU uses the data segment register by default. These specific examples require ds: because of MASM's syntactical limitations.

## The Register Indirect Addressing Modes

The 80x86 CPUs let you access memory indirectly through a register using the register indirect addressing modes. There are four forms of this addressing mode on the 8086, best demonstrated by the following instructions:

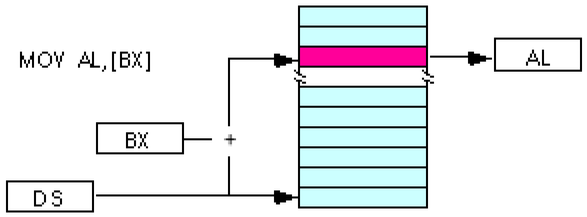
```
mov     al, [bx]
mov     al, [bp]
mov     al, [si]
mov     al, [di]
```

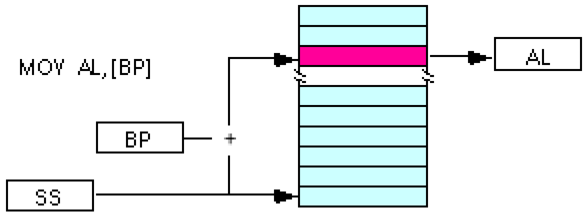
As with the x86 [bx] addressing mode, these four addressing modes reference the byte at the offset found in the bx, bp, si, or di register, respectively. The [bx], [si], and [di] modes use the ds segment by default. The [bp] addressing mode uses the stack segment (ss) by default.

You can use the segment override prefix symbols if you wish to access data in different segments. The following instructions demonstrate the use of these overrides:

```
mov     al, cs:[bx]
mov     al, ds:[bp]
mov     al, ss:[si]
mov     al, es:[di]
```

Intel refers to [bx] and [bp] as base addressing modes and bx and bp as base registers (in fact, bp stands for base pointer). Intel refers to the [si] and [di] addressing modes as indexed addressing modes (si stands for source index, di stands for destination index). However, these addressing modes are functionally equivalent. This text will call these forms register indirect modes to be consistent.





Note: the [si] and [di] addressing modes work exactly the same way, just substitute si and di for bx above.

## Indexed Addressing Modes

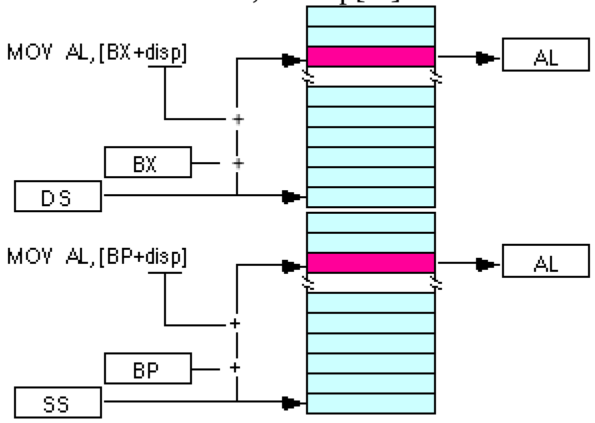
The indexed addressing modes use the following syntax:

```
mov     al, disp[bx]
mov     al, disp[bp]
mov     al, disp[si]
mov     al, disp[di]
```

If bx contains 1000h, then the instruction mov cl,20h[bx] will load cl from memory location ds:1020h. Likewise, if bp contains 2020h, mov dh,1000h[bp] will load dh from location ss:3020.

The offsets generated by these addressing modes are the sum of the constant and the specified register. The addressing modes involving bx, si, and di all use the data segment, the disp[bp] addressing mode uses the stack segment by default. As with the register indirect addressing modes, you can use the segment override prefixes to specify a different segment:

```
mov     al, ss:disp[bx]
mov     al, es:disp[bp]
mov     al, cs:disp[si]
mov     al, ss:disp[di]
```



You may substitute si or di in the figure above to obtain the [si+disp] and [di+disp] addressing modes.

Note that Intel still refers to these addressing modes as based addressing and indexed addressing. Intel's literature does not differentiate between these modes with or without the constant. If you look at how the hardware works, this is a reasonable definition. From the programmer's point of view, however, these addressing modes are useful for entirely different things. Which is why this text uses different terms to describe them. Unfortunately, there is very little consensus on the use of these terms in the 80x86 world.

**Based Indexed Addressing Modes**

The based indexed addressing modes are simply combinations of the register indirect addressing modes. These addressing modes form the offset by adding together a base register (bx or bp) and an index register (si or di). The allowable forms for these addressing modes are

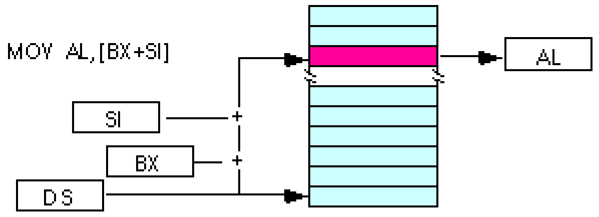
```
mov     al, [bx][si]
mov     al, [bx][di]
mov     al, [bp][si]
mov     al, [bp][di]
```

Suppose that bx contains 1000h and si contains 880h. Then the instruction

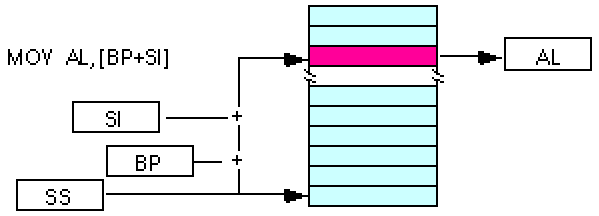
```
mov       al,[bx][si]
```

would load al from location DS:1880h. Likewise, if bp contains 1598h and di contains 1004, mov ax,[bp+di] will load the 16 bits in ax from locations SS:259C and SS:259D.

The addressing modes that do not involve bp use the data segment by default. Those that have bp as an operand use the stack segment by default.



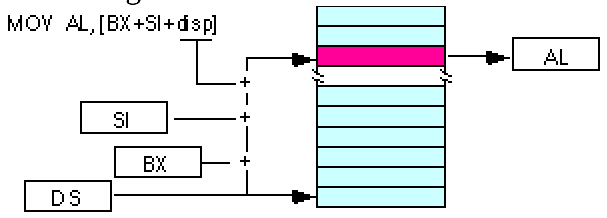
You substitute di in the figure above to obtain the [bx+di] addressing mode.



You substitute di in the figure above for the [bp+di] addressing mode.

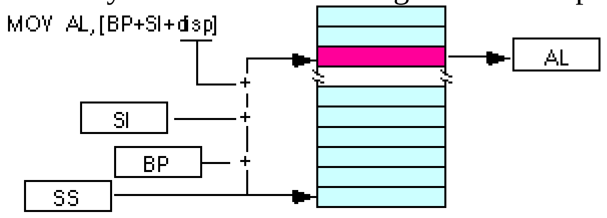
**Based Indexed Plus Displacement Addressing Mode**

These addressing modes are a slight modification of the base/indexed addressing modes with the addition of an eight bit or sixteen bit constant. The following are some examples of these addressing modes:



```
mov     al, disp[bx][si]
mov     al, disp[bx+di]
mov     al, [bp+si+disp]
mov     al, [bp][di][disp]
```

You may substitute di in the figure above to produce the [bx+di+disp] addressing mode.



You may substitute di in the figure above to produce the [bp+di+disp] addressing mode.

Suppose bp contains 1000h, bx contains 2000h, si contains 120h, and di contains 5. Then mov al,10h[bx+si] loads al from address DS:2130; mov ch,125h[bp+di] loads ch from location SS:112A; and mov bx,cs:2[bx][di] loads bx from location CS:2007.

# Chapter Three
# Instruction Groups

An assembly language, like many other computer languages, consists of a series of instruction statements that tell the CPU what operations to perform. An assembly language has three types of statements: *Assembler Directives*, *Instructions*, and *Comments*. Assembler directives are interpreted by the assembler at build time and are commonly used for defining program constants and reserving space for dynamic variables, among other things. Assembly instructions are divided into two fields: Operation and Operand fields. The Operation field consists of the mnemonic names for the target machine instructions. The Operand field contains the operand(s) and assembler directive arguments (if necessary). The Comment field is an optional field that allows the programmer to document their code. Comment fields are ignored by the assembler and begin with an asterisk '*' or semi-colon ';'. Finally, there is another optional field called the Label field. A label allows a programmer to use symbols for memory locations, which make assembly programs easier to read and write. At build time, the assembler "replaces" any labels with the respective memory location. Labels will be used extensively when you begin programming, but for now, let's put this idea on hold.
Usual opcodes in every processor exist for moving data, arithmetic and logical manipulations etc. However their mnemonics vary depending on the will of the manufacturer. Some manufacturers name the mnemonics for data movement instructions as "move," some call it "load" and "store" and still other names are present. But the basic set of instructions is similar in every processor. A grouping of these instructions makes learning a new processor quick and easy. Just the group an instruction belongs tells a lot about the instruction.

When you begin writing assembly programs, you will follow a specified format; however, each instruction will use the following structure:

***LABEL*** ***OPCODE*** ***OPERAND*** ***COMMENTS***

The opcode field contains the assembly mnemonic for the corresponding machine instruction and cannot start in the first column of the program (only labels can exist in the first column). The most common practice is to separate each field with a tab (or multiple tabs). The operand field contains the arguments required for the machine instruction or assembler directive, with comments following at the end of the instruction.

## The x86 Instruction Set

The x86 CPUs provide 20 basic instruction classes. Seven of these instructions have two operands, eight of these instructions have a single operand, and five instructions have no operands at all. The instructions are mov (two forms), add, sub, cmp, and, or, not, je, jne, jb, jbe, ja, jae, jmp, brk, iret, halt, get, and put. The following paragraphs describe how each of these work.
The *mov* instruction is actually two instruction classes merged into the same instruction.
The two forms of the mov instruction take the following forms:

*mov reg, reg/memory/constant*
*mov memory, reg*

where reg is any of ax, bx, cx, or dx; constant is a numeric constant (using hexadecimal notation), and memory is an operand specifying a memory location. The next section describes the possible forms the memory operand can take. The “reg/memory/constant” operand tells you that this particular operand may be a register, memory location, or a constant.

The *arithmetic and logical instructions* take the following forms:

*add reg, reg/memory/constant*
*sub reg, reg/memory/constant*
*cmp reg, reg/memory/constant*
*and reg, reg/memory/constant*
*or reg, reg/memory/constant*
*not reg/memory*

The add instruction adds the value of the second operand to the first (register) operand, leaving the sum in the first operand. The sub instruction subtracts the value of the second operand from the first, leaving the difference in the first operand. The cmp instruction compares the first operand against the second and saves the result of this comparison for use with one of the conditional jump instructions (described in a moment). The and and or instructions compute the corresponding bitwise logical operation on the two operands and store the result into the first operand. The not instruction inverts the bits in the single memory or register operand.

The *control transfer instructions* interrupt the sequential execution of instructions in memory and transfer control to some other point in memory either unconditionally, or after testing the result of the previous cmp instruction. These instructions include the following:

ja dest -- Jump if above
jae dest -- Jump if above or equal
jb dest -- Jump if below
jbe dest -- Jump if below or equal
je dest -- Jump if equal
jne dest -- Jump if not equal
jmp dest -- Unconditional jump
iret -- Return from an interrupt

The first six instructions in this class let you check the result of the previous cmp instruction for greater than, greater or equal, less than, less or equal, equality, or inequality. For example, if you compare the ax and bx registers with the cmp instruction and execute the ja instruction, the x86 CPU will jump to the specified destination location if ax was greater than bx. If ax is not greater than bx, control will fall through to the next instruction in the program. The jmp instruction unconditionally transfers control to the instruction at the destination address. The iret instruction returns control from an *interrupt service routine*, which we will discuss later.

The *get* and *put* instructions let you read and write integer values. Get will stop and prompt the user for a hexadecimal value and then store that value into the ax register. Put displays (in hexadecimal) the value of the ax register.

The remaining instructions do not require any operands, they are halt and brk. Halt terminates program execution and brk stops the program in a state that it can be restarted. The x86 processors require a unique opcode for every different instruction, not just the instruction classes.

Although “mov ax, bx” and “mov ax, cx” are both in the same class, they must have different opcodes if the CPU is to differentiate them. However, before looking at all the possible opcodes, perhaps it would be a good idea to learn about all the possible operands for these instructions.

Some of the 80x86 instructions are discussed in more details below:

- **Data movement instructions:** These instructions are used to move data from one place to another. These places can be registers, memory, or even inside peripheral devices. (e.g., MOV, LEA, LES, PUSH, POP, PUSHF, POPF)

The MOV instruction is used to transfer 8 and 16-bit data to and from registers. Either the source or destination has to be a register. The other operand can come from another register, from memory, from immediate data (a value included in the instruction) or from a memory location pointed at” by register BX. For example, if COUNT is the label of a memory location the following are possible assembly-language instructions. MOV instruction does the following:

· Copies the second operand (source) to the first operand (destination).
· The source operand can be an immediate value, general-purpose register or memory location.
· The destination register can be a general-purpose register, or memory location.
· Both operands must be the same size, which can be a byte or a word.

In the x86 assembly language, the MOV instruction is mnemonic for the copying of data from one location to another. The x86 has a number of different move instructions. Depending on whether the program is in a 16-bit or 32-bit code segment and whether an override instruction prefix is used, a MOV instruction may transfer 8-bits, 16-bits or 32-bits of data for 64 bits in x86-64 mode. Data may be copied to and from memory and registers.

The word “move” for this operation is strictly speaking a misnomer: it has little to do with the physical concept of moving an object from A to B, with place A then becoming empty; a MOV instead makes a copy of the state of the object at A and overwrites the old state of B in this process. This is reflected in some other assembly languages by using words like load, store or copy instead of move.

The following is an example of a mov instruction which copies the value in the register Y into register X:

```
MOV X,Y
```

This operation is represented by the following pseudocode:

```
X   : =  Y
```

The operands for the MOV commands can either be registers, a segment register or a memory address suince the command is executed in a single CPU work cycle. There is a succession as in:

Move the contents of the register bx into register ax

```
MOV ax, bx
```

Move the contents of the register ax into the referenced memory block

```
MOV [address], ax
```

Memory to memory moves, such as

```
MOV [address1],[address2]
```

are not possible.  To achieve this, MOV must be used in sequence:

```
MOV ax, [address2]
MOV [address1], ax
```

Usually, there is one set of opcodes for

```
MOV register, [address]
MOV [address], register
```

There are also special MOV opcodes for accessing control registers (a control register is a process register which changes or controls the general behavior of a CPU or other digital devices. Common tasks performed by control registers include interrupt control, switching the addressing mode, paging control and coprocessor control.

The mov instruction takes several different forms:

```
mov reg, reg1
mov mem, reg
mov reg, mem
mov mem, immediate data
mov reg, immediate data
mov ax/al, mem
mov mem, ax/al
mov segreg, mem16
mov segreg, reg16
mov mem16, segreg
mov reg16, segreg
```

For example:

```
mov ax,15           ; put 15 into ax
mov bx,25           ; put 25 into bx
mov cx,35           ; put 35 into cx
mov dx,45           ; put 45 into dx

mov bx,cx           ; put the content of cx into bx
mov ax,bx           ; put the content of bx into ax
mov cx,dx           ; put the content of dx into cx
```

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

```
mov  cl, 55h
mov dl,cl
mov ah, dl
```

The use of 16-bit registers is demonstrated below.

```
mov cx, 468fh
mov ax, cx
mov dx,ax
```

Data can be moved directly into non-segment registers only, using the MOV instruction. For example,

```
mov ax, 58fch       ;legal
mov dx, 6678h       ; legal
mov si, 924bh       ;legal
mov bp, 2459h       ; legal
mov ds, 2341h       ;illegal
mov cx,8876h        ;illegal
mov cs, 3f47h       ;illegal
mov bh, 99h         ; illegal
```

In the 8086 CPU, data can be moved among all the registers discussed earlier except the flag register as long as the source and destination registers match in size.
The following should be taken note of:

1. Code such as “MOV AL, DX” will cause an error.
2. Values cannot be loaded directly into any segment register (CS, DS, ES, or SS).

```
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
```

3. 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.
4. 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
```

For example, a program to add three numbers is written below

```
; a program to add three numbers using registers
        org 100h
                mov ax, 5           ; load first number in ax
                mov bx, 10          ; load second number in bx
                add ax, bx          ; accumulate sum in ax
                mov bx, 15          ; load third number in bx
                add ax, bx          ; accumulate sum in ax

                ret                 ; return control to the operating system
```

- **Arithmetic and Logical instructions** (e.g., ADD, SUB, AND, OR, XOR , NOT, INC, DEC, CMP, NEG, MUL, IMUL, DIV, IDIV, TEST): Arithmetic instructions like addition, subtraction, multiplication, division and Logical instructions like logical and, logical or, logical xor, or complement are part of this group.

```
add reg, reg
add reg, mem
add mem, reg
add reg, immediate data
add mem, immediate data
add eax/ax/al, immediate data
```

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.
After operation between operands, the result is always stored in the first operand. CMP and TEST instructions affect flags only and do not store a result (these instructions are used to make decisions during program execution). They only affect the following flags:
CF, ZF, SF, OF, PF, AF.

ADD - add second operand to first.
SUB - Subtract second operand from first.
CMP - Subtract second operand from first for flags only.

For MUL, IMUL, DIV, IDIV
These types of operands are supported:
REG, Memory

REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.
memory: [BX], [BX+SI+7], variable, etc.

MUL and IMUL instructions affect only 2 flags: CF, OF

When result is over operand size these flags are set to 1, when result fits in operand size they are set to 0.

For DIV and IDIV flags are undefined.

MUL - Unsigned multiply:
when operand is a byte: AX = AL * operand.
when operand is a word: (DX AX) = AX * operand.

**Explanation**
MUL (multiply) performs an unsigned multiplication of the source operand and the accumulator. If the source operand is a byte, then it is multiplied by register AL and the double-length result is returned in AH and AL. If the source operand is a word, then it is multiplied by register AX, and the double-length result is returned in registers DX and AX.

IMUL - Signed multiply:
when operand is a byte: AX = AL * operand.
when operand is a word: (DX AX) = AX * operand.

DIV - Unsigned divide:
when operand is a byte: AL = AX / operand
AH = remainder (modulus).

when operand is a word: AX = (DX AX) / operand
DX = remainder (modulus).

IDIV - Signed divide:
when operand is a byte: AL = AX / operand
AH = remainder (modulus).

when operand is a word:
AX = (DX AX) / operand
DX = remainder (modulus).

**INC**, **DEC**, **NOT**, **NEG**
These types of operands are supported:
REG
memory

**REG**: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.
**memory**: [BX], [BX+SI+7], variable, etc...

**INC**, **DEC** instructions affect these flags only:
**ZF**, **SF**, **OF**, **PF**, **AF**.
**NOT** instruction does not affect any flags!
**NEG** instruction affects these flags only:
**CF**, **ZF**, **SF**, **OF**, **PF**, **AF**.
• **NOT** - Reverse each bit of operand.
• **NEG** - Make operand negative (two's complement). Actually it reverses each bit of operand and then adds
1 to it. For example 5 will become -5, and -2 will become 2.

- **Program flow control instructions** (jmp, call, ret, conditional jumps): The instruction pointer points to the next instruction and instructions run one after the other with the help of this register. We can say that the instructions are tied with one another. In some situations we don't want to follow this implied path and want to order the processor to break its flow if some condition becomes true instead of the spatially placed next instruction. In certain other cases we want the processor to first execute a separate block of code and then come back to resume processing where it left. These are instructions that control the program execution and flow by playing with the instruction pointer and altering its normal behavior to point to the next instruction.
  - Unconditional Jumps

The basic instruction that transfers control to another point in the program is JMP.
The basic syntax of JMP instruction:
JMP label
To declare a *label* in your program, just type its name and add "**:**" to the end, label can be any character combination but it cannot start with a number, for example here are 3 legal label definitions:

```
label1:
label2:
a:
```

Label can be declared on a separate line or before any other instruction, for example:

```
x1:
MOV AX, 1
x2: MOV AX, 2
```

Here is an example of JMP instruction:

```
ORG   100h
MOV   AX, 5                  ; set AX to 5.
MOV   BX, 2                  ; set BX to 2.
JMP  calc                     ; go to 'calc'.
back: JMP stop               ; go to 'stop'.
calc:
```

```
ADD   AX, BX                  ; add BX to AX.
JMP back                      ; go 'back'.
stop:
RET                           ; return to operating system.
END                           ; directive to stop the compiler.
```

Of course there is an easier way to calculate the sum of two numbers, but it's still a good example of JMP instruction. As you can see from this example JMP is able to transfer control both forward and backward. It can jump anywhere in current code segment (65,535 bytes).

- Short Conditional Jumps

Unlike JMP instruction that does an unconditional jump, there are instructions that do a conditional jump (i.e jump only when some conditions are in action). These instructions are divided in three groups:
The first group just tests a single flag,
The second compares numbers as signed, and
The third compares numbers as unsigned.

**TYPES OF JUMP**
The three types of jump, near, short, and far, differ in the size of instruction and the range of memory they can jump to with the smallest short form of two bytes and a range of just 256 bytes to the far form of five bytes and a range covering the whole memory.

**Near Jump**
When the relative address stored with the instruction is in 16 bits, the jump is called a near jump. Using a near jump we can jump anywhere within a segment. If we add a large number it will wrap around to the lower part. A negative number actually is a large number and works this way using the wraparound behavior.
**Short Jump**
If the offset is stored in a single byte as in 75F2 with the opcode 75 and operand F2, the jump is called a short jump. F2 is added to IP as a signed byte. If the byte is negative the complement is negated from IP otherwise the byte is added. Unconditional jumps can be short, near, and far. The far type is yet to be discussed. Conditional jumps can only be short. A short jump can go +127 bytes ahead in code and -128 bytes backwards and no more. This is the limitation of a byte in signed representation.

**Far Jump**
Far jump is not position relative but is absolute. Both segment and offset must be given to a far jump. The previous two jumps were used to jump within a segment. Sometimes we may need to go from one code segment to another, and near and short jumps cannot take us there. Far jump must be used and a two byte segment and a two byte offset are given to it. It loads CS with the segment part and IP with the offset part. Execution therefore resumes from that location in physical memory. The three instructions that have a far form are JMP, CALL, and RET, are related to program control. Far capability makes intra segment control possible.

**Jump instructions that test single flag**

| Instruction | Description | Condition | Opposite Instruction |
|---|---|---|---|

|  |  |  |  |
|---|---|---|---|
| JZ , JE | Jump if Zero (Equal). | ZF = 1 | JNZ, JNE |
| JC , JB, JNAE | Jump if Carry (Below, Not Above Equal). | CF = 1 | JNC, JNB, JAE |
| JS | Jump if Sign. | SF = 1 | JNS |
| JO | Jump if Overflow. | OF = 1 | JNO |
| JPE, JP | Jump if Parity Even. | PF = 1 | JPO |
| JNZ , JNE | Jump if Not Zero (Not Equal). | ZF = 0 | JZ, JE |
| JNC , JNB, JAE | Jump if Not Carry (Not Below, Above Equal). | CF = 0 | JC, JB, JNAE |
| JNS | Jump if Not Sign. | SF = 0 | JS |
| JNO | Jump if Not Overflow. | OF = 0 | JO |
| JPO, JNP | Jump if Parity Odd (No Parity). | PF = 0 | JPE, JP |

**Jump instructions for signed numbers**

| Instruction | Description | Condition | Opposite Instruction |
|---|---|---|---|
| JE , JZ | Jump if Equal (=).<br>Jump if Zero. | ZF = 1 | JNE, JNZ |
| JNE , JNZ | Jump if Not Equal (<>).<br>Jump if Not Zero. | ZF = 0 | JE, JZ |
| JG , JNLE | Jump if Greater (>).<br>Jump if Not Less or Equal (not <=). | ZF = 0<br>and<br>SF = OF | JNG, JLE |
| JL , JNGE | Jump if Less (<).<br>Jump if Not Greater or Equal (not >=). | SF <> OF | JNL, JGE |
| JGE , JNL | Jump if Greater or Equal (>=).<br>Jump if Not Less (not <). | SF = OF | JNGE, JL |
| JLE , JNG | Jump if Less or Equal (<=).<br>Jump if Not Greater (not >). | ZF = 1<br>or<br>SF <> OF | JNLE, JG |

**Jump instructions for unsigned numbers**

| Instruction | Description | Condition | Opposite Instruction |
|---|---|---|---|
| JE , JZ | Jump if Equal (=).<br>Jump if Zero. | ZF = 1 | JNE, JNZ |
| JNE , JNZ | Jump if Not Equal (<>).<br>Jump if Not Zero. | ZF = 0 | JE, JZ |
| JA , JNBE | Jump if Above (>).<br>Jump if Not Below or Equal (not <=). | CF = 0<br>and<br>ZF = 0 | JNA, JBE |
| JB , JNAE, JC | Jump if Below (<). | CF = 1 | JNB, JAE, JNC |

| | | | |
|---|---|---|---|
| | Jump if Not Above or Equal (not >=).<br>Jump if Carry. | | |
| JAE , JNB, JNC | Jump if Above or Equal (>=).<br>Jump if Not Below (not <).<br>Jump if Not Carry. | CF = 0 | JNAE, JB |
| JBE , JNA | Jump if Below or Equal (<=).<br>Jump if Not Above (not >). | CF = 1<br>or<br>ZF = 1 | JNBE, JA |

Generally, when it is required to compare numeric values **CMP** instruction is used (it does the same as **SUB** (subtract) instruction, but does not keep the result, just affects the flags).
The logic is very simple, for example: it's required to compare 5 and 2,
5 - 2 = 3
the result is not zero (Zero Flag is set to 0).
Another example:
it's required to compare 7 and 7,
7 - 7 = 0
the result is zero! (Zero Flag is set to 1 and **JZ** or **JE** will do the jump).

Here is an example of **CMP** instruction and conditional jump:

```
include emu8086.inc  ; include a library of function
ORG   100h
MOV   AL, 25         ; set AL to 25.
MOV   BL, 10         ; set BL to 10.
CMP   AL, BL         ; compare AL - BL.
JE equal             ; jump if AL = BL (ZF = 1).
PUTC 'N'             ; if it gets here, then AL <> BL,
JMP stop              ; so print 'N', and jump to stop.
equal:               ; if gets here,
PUTC 'Y'              ; then AL = BL, so print 'Y'.
stop:
RET                  ; gets here no matter what.
END
```

All conditional jumps have one big limitation, unlike **JMP** instruction they can only jump **127** bytes forward and **128** bytes backward (note that most instructions are assembled into 3 or more bytes). We can easily avoid this limitation using a cute trick:
Get an opposite conditional jump instruction from the table above, make it jump to *label_x*.
Use **JMP** instruction to jump to the desired location.

Define *label_x:* just after the **JMP** instruction.
*label_x:* - can be any valid label name.
Here is an example:

```
include emu8086.inc
ORG   100h
```

```
MOV   AL, 25          ; set AL to 25.
MOV   BL, 10          ; set BL to 10.
CMP   AL, BL          ; compare AL - BL.

JNE not_equal         ; jump if AL <> BL (ZF = 0).
JMP equal
not_equal:
; let's assume that here we have a code that is assembled to more than 127 bytes.
PUTC 'N'              ; if it gets here, then AL <> BL,
JMP stop              ; so print 'N', and jump to stop.
equal:                ; if gets here,
PUTC 'Y'              ; then AL = BL, so print 'Y'.
stop:
RET                   ; gets here no matter what.
END
```

Another, yet rarely used method is providing an immediate value instead of a label. When immediate value starts with a '$' character relative jump is performed, otherwise compiler calculates instruction that jumps directly to given offset. For example:

```
ORG 100h
; unconditional jump forward:
; skip over next 2 bytes,
JMP $2
a DB 3 ; 1 byte.
b DB 4 ; 1 byte.
; JCC jump back 7 bytes:
; (JMP takes 2 bytes itself)
MOV BL,9
DEC BL ; 2 bytes.
CMP BL, 0 ; 3 bytes.
JNE $-7
RET
END
```

**Use of Memory Variables**

Writing programs using just the immediate operand type is difficult. Every reasonable program needs some data in memory apart from constants. Constants cannot be changed, i.e. they cannot appear as the destination operand. In fact placing them as destination is meaningless and illegal according to assembly language syntax. Only registers or data placed in memory can be changed. So real data is the one stored in memory, with a very few constants. So there must be a mechanism in assembly language to store and retrieve data from memory.

To declare a part of our program as holding data instead of instructions we need a couple of very basic but special assembler directives. The first directive is “define byte” written as “db.”

```
db somevalue
```

As a result a cell in memory will be reserved containing the desired value in it and it can be used in a variety of ways. Now we can add variables instead of constants. The other directive is “define word” or “dw” with the same syntax as “db” but reserving a whole word of 16 bits instead of a byte.

There are directives to declare a double or a quad word as well but we will restrict ourselves to byte and word declarations for now. For single byte we use db and for two bytes we use dw.
To refer to this variable later in the program, we need the address occupied by this variable. The assembler is there to help us. We can associate a symbol with any address that we want to remember and use that symbol in the rest of the code. The symbol is there for our own comprehension of code.
The assembler will calculate the address of that symbol using our origin directive and calculating the instruction lengths or data declarations in between and replace all references to the symbol with the corresponding address. This is just like variables in a higher level language, where the compiler translates them into addresses; just the process is hidden from the programmer one level further. Such a symbol associated to a point in the program is called a label and is written as the label name followed by a colon.

In order to add numbers, we could use memory variables to store the values.
An example is shown below

```
; a program to add three numbers using memory variables
org 100h                    ; meaning originate the program at 0100. The first executable
                            ;instruction should be placed at this offset.
mov ax, [num1]              ; load first number in ax
mov bx, [num2]               ; load second number in bx
add ax, bx                  ; accumulate sum in ax
mov bx, [num3]               ; load third number in bx
add ax, bx                  ; accumulate sum in ax
mov [num4], ax              ; store sum in num4
ret                         ;  return control to the operating system
end                         ; terminate program
num1: dw 5
num2: dw 10
num3: dw 15
num4: dw 0
```

```
; a program to add three numbers directly in memory
org   100h
mov ax, [num1]              ; load first number in ax
mov [num1+6], ax            ; store first number in result
mov ax, [num1+2]            ; load second number in ax
add [num1+6], ax             ; add second number to result
mov ax, [num1+4]             ; load third number in ax
add [num1+6], ax            ; add third number to result
ret;
num1: dw 5, 10, 15, 0
```

```
; a program to add ten numbers without using array
org 100h
mov bx, num1           ; point bx to first number
mov cx, 10            ; load count of numbers in cx
mov ax, 0             ; initialize sum to zero
l1: add ax, [bx]      ; add number to ax
add bx, 2             ; advance bx to next number
```

```
sub cx, 1                ; numbers to be added reduced
jnz l1                   ; if numbers remain add next
mov [total], ax          ; write back sum in memory
ret
num1: dw 10, 20, 30, 40, 50, 10, 20, 30, 40, 50
```

```
; a program to add ten numbers without a separate counter
org 100h
jmp start ; unconditionally jump over data
num1: dw 10, 20, 30, 40, 50, 10, 20, 30, 40, 50
total: dw 0
start: mov bx, 0 ; initialize array index to zero
mov ax, 0 ; initialize sum to zero
l1: add ax, [num1+bx] ; add number to ax
add bx, 2 ; advance bx to next index
cmp bx, 20 ; are we beyond the last index
jne l1 ; if not add next number
mov [total], ax ; write back sum in memory
ret;
```

```
; sorting a list of ten numbers using bubble sort
org 100h
jmp start
data:    dw 60, 55, 45, 50, 40, 35, 25, 30, 10, 0
swap:    db 0
start:   mov bx, 0              ; initialize array index to zero
mov byte [swap], 0              ; rest swap flag to no swaps
loop1: mov ax, [data+bx]        ; load number in ax
cmp ax, [data+bx+2]             ; compare with next number
jbe noswap                      ; no swap if already in order
mov dx, [data+bx+2]             ; load second element in dx
mov [data+bx+2], ax             ; store first number in second
mov [data+bx], dx               ; store second number in first
mov byte [swap], 1              ; flag that a swap has been done
noswap: add bx, 2               ; advance bx to next index
cmp bx, 18                      ; are we at last index
jne loop1                       ; if not compare next two
cmp byte [swap], 1              ; check if a swap has been done
je bsort                        ; if yes make another pass
ret
```

# Chapter Four
## 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).

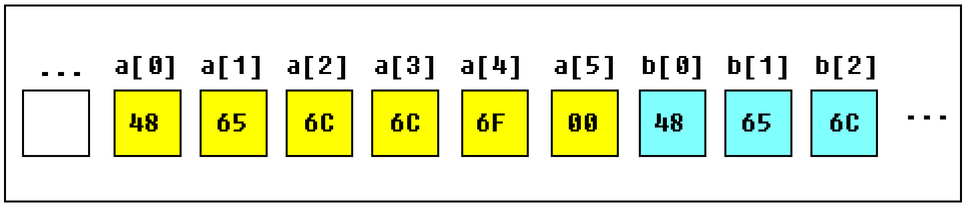
A single dimensional array implementation is of this sort:

A: array [0..4] of sometype;
We now have A[0], A[1], A[2], A[3] and A[4] , A[0] being the base address of A.

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

You can use **DW** instead of **DB** if it's required to keep values larger than 255, or smaller than -128. **DW** cannot be used to declare strings!
The expansion of **DUP** operand should not be over 1020 characters!
(the expansion of the last example is 13 chars), if you need to declare a huge array, divide the declaration into two lines (you will get a single huge array in the memory).

Other declaration examples:

```
CharArray char 128 dup (?) ;array[0..127] of char
IntArray integer 8 dup (?) ;array[0..7] of integer
BytArray byte 10 dup (?) ;array[0..9] of byte
PtrArray dword 4 dup (?) ;array[0..3] of dword
```

The first two examples, of course, assume that you've used the typedef statement to define the char and integer data types.

These examples all allocate storage for uninitialized arrays. You may also specify that the elements of the arrays be initialized to a single value using declarations like the following:

```
RealArray real4 8 dup (1.0)
IntegerAry integer 8 dup (1)
```

These definitions both create arrays with eight elements. The first definition initializes each four-byte real value to 1.0, the second declaration initializes each integer element to one.

# Chapter Five
# Library of Common Functions

## Library of common functions - emu8086.inc

To make programming easier there are some common functions that can be included in your program. To make your program use functions defined in other file you should use the **INCLUDE** directive followed by a file name. The compiler automatically searches for the file in the same folder where the source file is located, and if it cannot find the file there - it searches in **Inc** folder.

To use any of the functions in **emu8086.inc** you should have the following line in the beginning of your source file:

```
include 'emu8086.inc'
```

**emu8086.inc** defines the following **macros**:

**PUTC char** - macro with 1 parameter, prints out an ASCII char at current cursor position.
**GOTOXY col, row** - macro with 2 parameters, sets cursor position.
**PRINT string** - macro with 1 parameter, prints out a string.
**PRINTN string** - macro with 1 parameter, prints out a string. The same as PRINT but automatically adds "carriage return" at the end of the string.
**CURSOROFF** - turns off the text cursor.
**CURSORON** - turns on the text cursor.
To use any of the above macros simply type its name somewhere in your code, and if required parameters, for example:

```
ORG 100h
PRINT 'Hello World!'
GOTOXY 10, 5
PUTC 65 ; 65 - is an ASCII code for 'A'
PUTC 'B'
RET ; return to operating system.
END ; directive to stop the compiler.
```

When the compiler processes your source code it searches the **emu8086.inc** file for declarations of the macros and replaces the macro names with real code. Generally macros are relatively small parts of code; frequent use of a macro may make your executable too big (procedures are better for size optimization).

**emu8086.inc** also defines the following **procedures**:
**PRINT_STRING** - procedure to print a null terminated string at current cursor position, receives address of string in **DS:SI** register. To use it declare: **DEFINE_PRINT_STRING** before **END** directive.
**PTHIS** - procedure to print a null terminated string at current cursor position (just as PRINT_STRING), but receives address of string from Stack. The ZERO TERMINATED string should be defined just after the CALL instruction.

For example:

```
CALL PTHIS
db 'Hello World!', 0
```

To use it declare: **DEFINE_PTHIS** before **END** directive.

**GET_STRING** - procedure to get a null terminated string from a user, the received string is written to buffer at **DS:DI**, buffer size should be in **DX**. Procedure stops the input when 'Enter' is pressed. To use it declare:
**DEFINE_GET_STRING** before **END** directive.

**CLEAR_SCREEN** - procedure to clear the screen, (done by scrolling entire screen window), and set cursor position to top of it. To use it declare:
**DEFINE_CLEAR_SCREEN** before **END** directive.

**SCAN_NUM** - procedure that gets the multi-digit SIGNED number from the keyboard, and stores the result in **CX** register. To use it declare:
**DEFINE_SCAN_NUM** before **END** directive.

**PRINT_NUM** - procedure that prints a signed number in **AX** register. To use it declare: **DEFINE_PRINT_NUM** and
**DEFINE_PRINT_NUM_UNS** before **END** directive.

**PRINT_NUM_UNS** - procedure that prints out an unsigned number in **AX** register. To use it declare: **DEFINE_PRINT_NUM_UNS** before **END** directive.

To use any of the above procedures you should first declare the function in the bottom of your file (but before **END**!!), and then use **CALL** instruction followed by a procedure name. For example:

```
include 'emu8086.inc'
ORG 100h
LEA SI, msg1 ; ask for the number
CALL print_string ;
CALL scan_num ; get number in CX.
MOV AX, CX ; copy the number to AX.
; print the following string:
CALL pthis
DB 13, 10, 'You have entered: ', 0
CALL print_num ; print number in AX.
RET ; return to operating system.
msg1 DB 'Enter the number: ', 0
DEFINE_PRINT_STRING
DEFINE_PRINT_NUM
DEFINE_PRINT_NUM_UNS ; required for print_num.
DEFINE_PTHIS
END ; directive to stop the compiler.
```

# Chapter Six

## Procedures

A Procedure is a part of code that can be called from your program in order to make some specific task. Procedures make program more structural and easier to understand. Generally procedure returns to the same point from where it was called.

The syntax for procedure declaration:

```
name PROC
; here goes the code
; of the procedure ...
RET
name ENDP
```

name - is the procedure name, the same name should be in the top and the bottom, this is used to check correct closing of procedures.

Probably, you already know that **RET** instruction is used to return to operating system. The same instruction is used to return from procedure (actually operating system sees your program as a special procedure).

**PROC** and **ENDP** are compiler directives, so they are not assembled into any real machine code. Compiler just remembers the address of procedure.

**CALL** instruction is used to call a procedure.

```
ORG 100h
CALL m1
MOV AX, 2
RET ; return to operating system.
m1 PROC
MOV BX, 5
RET ; return to caller.
m1 ENDP
END
```

The above example calls procedure **m1**, does **MOV BX, 5**, and returns to the next instruction after **CALL**: **MOV AX, 2**.

There are several ways to pass parameters to procedure, the easiest way to pass parameters is by using registers, here is another example of a procedure that receives two parameters in **AL** and **BL** registers, multiplies these parameters and returns the result in **AX** register:

```
ORG 100h
MOV AL, 1
MOV BL, 2
CALL m2
CALL m2
CALL m2
CALL m2
RET ; return to operating system.
m2 PROC
MUL BL ; AX = AL * BL.
RET ; return to caller.
m2 ENDP
END
```

In the above example value of **AL** register is updated every time the procedure is called, **BL** register stays unchanged, so this algorithm calculates **2** in power of **4**, so final result in **AX** register is **16** (or 10h).

Here goes another example, that uses a procedure to print a *Hello World!* message:

```
ORG 100h
LEA SI, msg ; load address of msg to SI.
CALL print_me
RET ; return to operating system.
; ==========================================================
; this procedure prints a string, the string should be null
; terminated (have zero in the end),
; the string address should be in SI register:
print_me PROC
next_char:
CMP b.[SI], 0 ; check for zero to stop
JE stop ;
MOV AL, [SI] ; next get ASCII char.
MOV AH, 0Eh ; teletype function number.
INT 10h ; using interrupt to print a char in AL.
ADD SI, 1 ; advance index of string array.
JMP next_char ; go back, and type another char.
stop:
RET ; return to caller.
print_me ENDP
; ==========================================================
msg DB 'Hello World!', 0 ; null terminated string.
END
```

"**b.**" - prefix before [SI] means that we need to compare bytes, not
words. When you need to compare words add "**w.**" prefix instead.
When one of the compared operands is a register it's not required
because compiler knows the size of each register.

**Sample Codes and questions**

```
; sorting a list of ten numbers using bubble sort
[org 0x0100]
jmp start
data: dw 60, 55, 45, 50, 40, 35, 25, 30, 10, 0
swap: db 0
start: mov bx, 0 ; initialize array index to zero
mov byte [swap], 0 ; rest swap flag to no swaps
loop1: mov ax, [data+bx] ; load number in ax
cmp ax, [data+bx+2] ; compare with next number
jbe noswap ; no swap if already in order
mov dx, [data+bx+2] ; load second element in dx
mov [data+bx+2], ax ; store first number in second
mov [data+bx], dx ; store second number in first
mov byte [swap], 1 ; flag that a swap has been done
noswap: add bx, 2 ; advance bx to next index
cmp bx, 18 ; are we at last index
jne loop1 ; if not compare next two
cmp byte [swap], 1 ; check if a swap has been done
je bsort ; if yes make another pass
mov ax, 0x4c00 ; terminate program
int 0x21
```

**EXERCISES**

1. Which registers are changed by the CMP instruction?
2. What are the different types of jumps available? Describe position relative addressing.
3. If AX=8FFF and BX=0FFF and "cmp ax, bx" is executed, which of the following jumps will be taken? Each part is independent of others. Also give the value of Z, S, and C flags.
a. jg greater
b. jl smaller
c. ja above
d. jb below
4. Write a program to find the maximum number and the minimum number from an array of ten numbers.
5. Write a program to search a particular element from an array using binary search. If the element is found set AX to one and otherwise to zero.
6. Write a program to calculate the factorial of a number where factorial is defined as:
factorial(x) = x*(x-1)*(x-2)*...*1
factorial(0) = 1

**Exercises**:

1) Write instructions to:
Load character ? into register bx
Load space character into register cx
Load 26 (decimal) into register cx
Copy contents of ax to bx and dx

2) What errors are present in the following :

```
mov ax 3d
mov 23, ax
mov cx, ch
move ax, 1h
add 2, cx
add 3, 6
inc ax, 2
```

3) Write instructions to evaluate the arithmetic expression 5 + (6-2) leaving the result in ax using (a) 1 register, (b) 2 registers, (c) 3 registers

4) Write instructions to evaluate the expressions:

a = b + c –d

z = x + y + w – v +u

5) Rewrite the expression in 4) above but using the registers ah, al, bh, bl and so on to represent the variables: a, b, c, z, x, y, w, u, and v.