Assembly Language Programming
Lecture Notes

Volume 1 – Basic Concepts

Preface

Assembly language programming develops a very basic and low level understanding of the computer. In higher level languages there is a distance between the computer and the programmer. This is because higher level languages are designed to be closer and friendlier to the programmer, thereby creating distance with the machine. This distance is covered by translators called compilers and interpreters. The aim of programming in assembly language is to bypass these intermediates and talk directly with the computer.

There is a general impression that assembly language programming is a difficult chore and not everyone is capable enough to understand it. The reality is in contrast, as assembly language is a very simple subject. The wrong impression is created because it is very difficult to realize that the real computer can be so simple. Assembly language programming gives a freehand exposure to the computer and lets the programmer talk with it in its language. The only translator that remains between the programmer and the computer is there to symbolize the computer's numeric world for the ease of remembering.

To cover the practical aspects of assembly language programming, IBM PC based on Intel architecture will be used as an example. However this course will not be tied to a particular architecture as it is often done. In our view such an approach does not create versatile assembly language programmers. The concepts of assembly language that are common across all platforms will be developed in such a manner as to emphasize the basic low level understanding of the computer instead of the peculiarities of one particular architecture. Emphasis will be more on assembly language and less on the IBM PC.

Before attempting this course you should know basic digital logic operations of AND, OR, NOT etc. You should know binary numbers and their arithmetic. Apart from these basic concepts there is nothing much you need to know before this course. In fact if you are not an expert, you will learn assembly language quickly, as non-experts see things with simplicity and the basic beauty of assembly language is that it is exceptionally simple. Do not ever try to find a complication, as one will not be there. In assembly language what is written in the program is all that is there, no less and no more.

After successful completion of this course, you will be able to explain all the basic operations of the computer and in essence understand the psychology of the computer. Having seen the computer from so close, you will understand its limitations and its capabilities. Your logic will become fine grained and this is one of the basic objectives of teaching assembly language programming.

Then there is the question that why should we learn assembly language when there are higher level languages one better than the other; C, C++, Java, to name just a few, with a neat programming environment and a simple way to write programs. Then why do we need such a freehand with the computer that may be dangerous at times? The answer to this lies in a very simple example. Consider a translator translating from English to Japanese. The problem faced by the translator is that every language has its own vocabulary and grammer. He may need to translate a word into a sentence and destroy the beauty of the topic. And given that we do not know

Japanese, so we cannot verify that our intent was correctly conveyed or not. Compiler is such a translator, just a lot dumber, and having a scarce number of words in its target language, it is bound to produce a lot of garbage and unnecessary stuff as a result of its ignorance of our program logic. In normal programs such garbage is acceptable and the ease of programming overrides the loss in efficiency but there are a few situations where this loss is unbearable.

Think about a four color picture scanned at 300 dots per inch making 90000 pixels per square inch. Now a processing on this picture requires 360000 operations per square inch, one operation for each color of each pixel. A few extra instructions placed by the translator can cost hours of extra time. The only way to optimize this is to do it directly in assembly language. But this doesn't mean that the whole application has to be written in assembly language, which is almost never the case. It's only the performance critical part that is coded in assembly language to gain the few extra cycles that matter at that point.

Consider an arch just like the ones in mosques. It cannot be made of big stones alone as that would make the arch wildly jagged, not like the fine arch we are used to see. The fine grains of cement are used to smooth it to the desired level of perfection. This operation of smoothing is optimization. The core structure is built in a higher level language with the big blocks it provides and the corners that need optimization are smoothed with the fine grain of assembly language which allows extreme control.

Another use of assembly language is in a class of time critical systems called real time systems. Real time systems have time bound responses, with an upper limit of time on certain operations. For such precise timing requirement, we must keep the instructions in our total control. In higher level languages we cannot even tell how many computer instructions were actually used, but in assembly language we can have precise control over them. Any reasonable sized application or a serious development effort has nooks and corners where assembly language is needed. And at these corners if there is no assembly language, there can be no optimization and when there is no optimization, there is no beauty. Sometimes a useful application becomes useless just because of the carelessness of not working on these jagged corners.

The third major reason for learning assembly language and a major objective for teaching it is to produce fine grained logic in programmers. Just like big blocks cannot produce an arch, the big thick grained logic learnt in a higher level language cannot produce the beauty and fineness assembly language can deliver. Each and every grain of assembly language has a meaning, nothing is presumed. You have to put together these grains, the minimum number of them to produce the desired outcome. Just like a "for" loop in a higher level language is a block construct and has a hundred things hidden in it, but using the grains of assembly language we do a similar operation with a number of grains but in the process understand the minute logic hidden beside that simple "for" construct.

Assembly language cannot be learnt by reading a book or by attending a course. It is a language that must be tasted and enjoyed. There is no other way to learn it. You will need to try every example, observe and verify the things you are told about it, and experiment a lot with the computer. Only then you will know and become able to appreciate how powerful, versatile, and simple this language is; the three properties that are hardly ever present together.

Whether you program in C/C++ or Java, or in any programming paradigm be it object oriented or declarative, everything has to boil down to the bits and bytes of assembly language before the computer can even understand it.

Table of Contents

1
Introduction to Assembly Language

1.1. BASIC COMPUTER ARCHITECTURE

Address, Data, and Control Buses

A computer system comprises of a processor, memory, and I/O devices. I/O is used for interfacing with the external world, while memory is the processor's internal world. Processor is the core in this picture and is responsible for performing operations. The operation of a computer can be fairly described with processor and memory only. I/O will be discussed in a later part of the course. Now the whole working of the computer is performing an operation by the processor on data, which resides in memory.

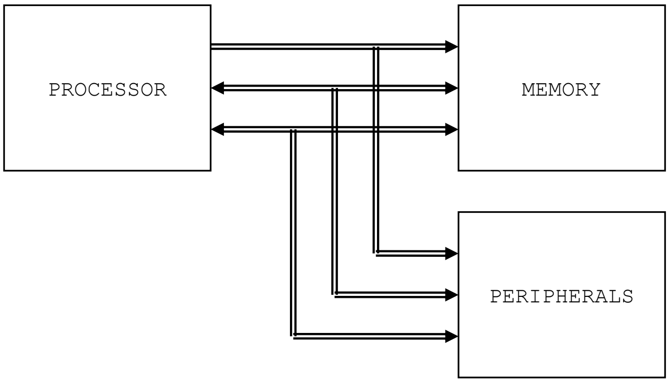
The scenario that the processor executes operations and the memory contains data elements requires a mechanism for the processor to read that data from the memory. "That data" in the previous sentence much be rigorously explained to the memory which is a dumb device. Just like a postman, who must be told the precise address on the letter, to inform him where the destination is located. Another significant point is that if we only want to read the data and not write it, then there must be a mechanism to inform the memory that we are interested in reading data and not writing it. Key points in the above discussion are:

- There must be a mechanism to inform memory that we want to do the read operation
- There must be a mechanism to inform memory that we want to read precisely which element
- There must be a mechanism to transfer that data element from memory to processor

The group of bits that the processor uses to inform the memory about which element to read or write is collectively known as the *address bus*. Another important bus called the *data bus* is used to move the data from the memory to the processor in a read operation and from the processor to the memory in a write operation. The third group consists of miscellaneous independent lines used for control purposes. For example, one line of the bus is used to inform the memory about whether to do the read operation or the write operation. These lines are collectively known as the *control bus*.

These three buses are the eyes, nose, and ears of the processor. It uses them in a synchronized manner to perform a meaningful operation. Although the programmer specifies the meaningful operation, but to fulfill it the processor needs the collaboration of other units and peripherals. And that collaboration is made available using the three buses. This is the very basic description of a computer and it can be extended on the same lines to I/O but we are leaving it out just for simplicity for the moment.

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. Data moves from both, processor to memory and memory to processor, so the data bus is bidirectional. Control bus is special and relatively complex, because different lines comprising it behave differently. Some take information from the processor to a peripheral and some take information from the peripheral to the processor. There can be certain events outside the processor that are of its interest. To bring information about these events the data bus cannot be used as it is owned by the processor and will only be used when the processor grants permission to use it. Therefore certain processors provide control lines to bring such information to processor's notice in the control bus. Knowing these signals in detail is unnecessary but the general idea of the control bus must be conceived in full.



We take an example to explain the collaboration of the processor and memory using the address, control, and data buses. Consider that you want your uneducated servant to bring a book from the shelf. You order him to bring the fifth book from top of the shelf. All the data movement operations are hidden in this one sentence. Such a simple everyday phenomenon seen from this perspective explains the seemingly complex working of the three buses. We told the servant to "bring a book" and the one which is "fifth from top," precise location even for the servant who is much more intelligent then our dumb memory. The dumb servant follows the steps one by one and the book is in your hand as a result. If however you just asked him for a book or you named the book, your uneducated servant will stand there gazing at you and the book will never come in your hand.

Even in this simplest of all examples, mathematics is there, "fifth from top." Without a number the servant would not be able to locate the book. He is unable to understand your will. Then you tell him to put it with the seventh book on the right shelf. Precision is involved and only numbers are precise in this world. One will always be one and two will always be two. So we tell in the form of a number on the address bus which cell is needed out of say the 2000 cells in the whole memory.

A binary number is generated on the address bus, fifth, seventh, eighth, tenth; the cell which is needed. So the cell number is placed on the address bus. A memory cell is an n-bit location to store data, normally 8-bit also called a byte. The number of bits in a cell is called the *cell width*. The two dimensions, cell width and number of cells, define the memory completely just like the width and depth of a well defines it completely. 200 feet deep by 15 feet wide and the well is completely described. Similarly for memory we define two dimensions. The first dimension defines how many parallel bits are there in a single memory cell. The memory is called 8-bit or 16-bit for this reason and this is also the word size of the memory. This need not match the size of a processor word which has other parameters to define it. In general the memory cell cannot be wider than the width of the data bus. Best and simplest operation requires the same size of data bus and memory cell width.

As we previously discussed that 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 only the mechanism. 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. Then why is the control bus bidirectional. Again we take the same example of the servant and the book further to elaborate this situation. Consider that the servant went to fetch the book just to find that the drawing room door is 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.

1.2. 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 concept of registers will be further elaborated as we progress into writing our first program.

Memory is a limited resource but the number of memory cells is large. Registers are relatively very small in number, and are therefore a very scarce and precious resource. Registers are more than one in number, so we have to precisely identify or name them. Some manufacturers number their registers like r0, r1, r2, others name them like A, B, C, D. Naming is useful since the registers are few in number. This is called the nomenclature of the particular architecture. Still other manufacturers name their registers according to their function like X stands for an index register. This also informs us that there are special functions of registers as well, some of which are closely associated to the particular architecture. For example index registers do not hold data instead they are used to hold the address of data. There are other functions as well and the whole spectrum of register functionalities is quite large. However most of the details will become clear as the registers of the Intel architecture are discussed in detail.

Accumulator

There is a central register in every processor called the accumulator. Traditionally all mathematical and logical operations are performed on the accumulator. The word size of a processor is defined by the width of its accumulator. A 32bit processor has an accumulator of 32 bits.

Pointer, Index, or Base Register

The name varies from manufacturer to manufacturer, but the basic distinguishing property is that it does not hold data but holds the address of data. The rationale can be understood by examining a “for” loop in a higher level language, zeroing elements in an array of ten elements located in consecutive memory cells. The location to be zeroed changes with every iteration. That is the address where the operation is performed is changing. Index register is used in such a situation to hold the address of the current array location. Now the value in the index register cannot be treated as data, but it is the address of data. In general whenever we need access to a memory location whose address is not known until runtime we need an index register. Without this register we would have needed to explicitly code each iteration separately.

In newer architectures the distinction between accumulator and index registers has become vague. They have general registers which are more versatile and can do both functions. They do have some specialized behaviors but basic operations can be done on all general registers.

Flags Register or Program Status Word

This is a special register in every architecture called the flags register or the program status word. Like the accumulator it is an 8, 16, or 32 bits register but unlike the accumulator it is meaningless as a unit, rather the individual bits carry different meanings. The bits of the accumulator work in parallel as a unit and each bit mean the same thing. The bits of the flags register work independently and individually, and combined its value is meaningless.

An example of a bit commonly present in the flags register is the carry flag. The carry can be contained in a single bit as in binary arithmetic the carry can only be zero or one. If a 16bit number is added to a 16bit accumulator, and the result is of 17 bits the 17th bit is placed in the carry bit of the flags register. Without this 17th bit the answer is incorrect. More examples of flags will be discussed when dealing with the Intel specific register set.

Program Counter or Instruction Pointer

Everything must translate into a binary number for our dumb processor to understand it, be it an operand or an operation itself. Therefore the instructions themselves must be translated into numbers. For example to add numbers we understand the word “add.” We translate this word into a number to make the processor understand it. This number is the actual instruction for the computer. All the objects, inheritance and encapsulation constructs in higher level languages translate down to just a number in assembly language in the end. Addition, multiplication, shifting; all big programs are made using these simple building blocks. A number is at the bottom line since this is the only thing a computer can understand.

A program is defined to be "an ordered set of instructions." Order in this definition is a key part. Instructions run one after another, first, second, third and so on. Instructions have a positional relationship. The whole logic depends on this positioning. If the computer executes the fifth instructions after the first and not the second, all our logic is gone. The processor should ensure this ordering of instructions. A special register exists in every processor called the program counter or the instruction pointer that ensures this ordering. "The program counter holds the address of the next instruction to be executed." A number is placed in the memory cell pointed to by this register and that number tells the processor which instruction to execute; for example 0xEA, 255, or 152. For the processor 152 might be the add instruction. Just this one number tells it that it has to add, where its operands are, and where to store the result. This number is called the *opcode*. The instruction pointer moves from one opcode to the next. This is how our program executes and progresses. One instruction is picked, its operands are read and the instruction is executed, then the next instruction is picked from the new address in instruction pointer and so on.

Remembering 152 for the add operation or 153 for the subtract operation is difficult. To make a simple way to remember difficult things we associate a symbol to every number. As when we write "add" everyone understands what we mean by it. Then we need a small program to convert this "add" of ours to 152 for the processor. Just a simple search and replace operation to translate all such symbols to their corresponding opcodes. We have mapped the numeric world of the processor to our symbolic world. "Add" conveys a meaning to us but the number 152 does not. We can say that add is closer to the programmer's thinking. This is the basic motive of adding more and more translation layers up to higher level languages like C++ and Java and Visual Basic. These symbols are called *instruction mnemonics*. Therefore the mnemonic "add a to b" conveys more information to the reader. The dumb translator that will convert these mnemonics back to the original opcodes is a key program to be used throughout this course and is called the *assembler*.

1.3. INSTRUCTION GROUPS

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.

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. Some examples are:

```
mov  ax, bx
lda  1234
```

Arithmetic and Logic Instructions

Arithmetic instructions like addition, subtraction, multiplication, division and Logical instructions like logical and, logical or, logical xor, or complement are part of this group. Some examples are:

```
and  ax, 1234
add  bx, 0534
add  bx, [1200]
```

The bracketed form is a complex variation meaning to add the data placed at address 1200. Addressing data in memory is a detailed topic and is discussed in the next chapter.

Program Control Instructions

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. Some examples are:

```
cmp  ax, 0
jne  1234
```

We are changing the program flow to the instruction at 1234 address if the condition that we checked becomes true.

Special Instructions

Another group called special instructions works like the special service commandos. They allow changing specific processor behaviors and are used to play with it. They are used rarely but are certainly used in any meaningful program. Some examples are:

```
cli
sti
```

Where cli clears the interrupt flag and sti sets it. Without delving deep into it, consider that the cli instruction instructs the processor to close its ears from the outside world and never listen to what is happening outside, possibly to do some very important task at hand, while sti restores normal behavior. Since these instructions change the processor behavior they are placed in the special instructions group.

1.4. INTEL IAPX88 ARCHITECTURE

Now we select a specific architecture to discuss these abstract ideas in concrete form. We will be using IBM PC based on Intel architecture because of its wide availability, because of free assemblers and debuggers available for it, and because of its wide use in a variety of domains. However the concepts discussed will be applicable on any other architecture as well; just the mnemonics of the particular language will be different.

Technically iAPX88 stands for “Intel Advanced Processor Extensions 88.” It was a very successful processor also called 8088 and was used in the very first IBM PC machines. Our discussion will revolve around 8088 in the first half of the course while in the second half we will use iAPX386 which is very advanced and powerful processor. 8088 is a 16bit processor with its accumulator and all registers of 16 bits. 386 on the other hand, is a 32bit processor. However it is downward compatible with iAPX88 meaning that all code written for 8088 is valid on the 386. The architecture of a processor means the organization and functionalities of the registers it contains and the instructions that are valid on the processor. We will discuss the register architecture of 8088 in detail below while its instructions are discussed in the rest of the book at appropriate places.

1.5. HISTORY

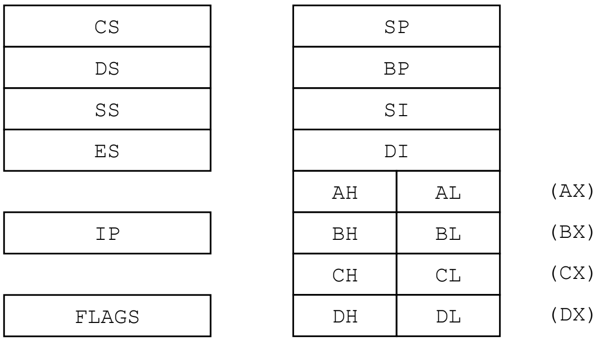
Intel did release some 4bit processors in the beginning but the first meaningful processor was 8080, an 8bit processor. The processor became popular due to its simplistic design and versatile architecture. Based on the experience gained from 8080, an advanced version was released as 8085. The processor became widely popular in the engineering community again due to its simple and logical nature.

Intel introduced the first 16bit processor named 8088 at a time when the concept of personal computer was evolving. With a maximum memory of 64K on the 8085, the 8088 allowed a whole mega byte. IBM embedded this processor in their personal computer. The first machines ran at 4.43 MHz; a blazing speed at that time. This was the right thing at the right moment. No one expected this to become the biggest success of computing history. IBM PC XT became so popular and successful due to its open architecture and easily available information.

The success was unexpected for the developers themselves. As when Intel introduced the processor it contained a timer tick count which was valid for five years only. They never anticipated the architecture to stay around for more than five years but the history took a turn and the architecture is there at every desk even after 25 years and the tick is to be specially handled every now and then.

1.6. REGISTER ARCHITECTURE

The iAPX88 architecture consists of 14 registers.



General Registers (AX, BX, CX, and DX)

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 as discussed in the next chapter. 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 A, B, C, and D are in letter sequence as well as depict some special functionality of the register.

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.

Instruction Pointer (IP)

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.

Stack Pointer (SP)

It is a memory pointer and is used indirectly by a set of instructions. This register will be explored in the discussion of the system stack.

Base Pointer (BP)

It is also a memory pointer containing the address in a special area of memory called the stack and will be explored alongside SP in the discussion of the stack.

Flags Register

The flags register as previously discussed 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

Segment Registers (CS, DS, SS, and ES)

The code segment register, data segment register, stack segment register, and the extra segment register are special registers related to the Intel segmented memory model and will be discussed later.

1.7. OUR FIRST PROGRAM

The first program that we will write will only add three numbers. This very simple program will clarify most of the basic concepts of assembly language. We will start with writing our algorithm in English and then moving on to convert it into assembly language.

English Language Version

"Program is an ordered set of instructions for the processor." Our first program will be instructions manipulating AX and BX in plain English.

```
move 5 to ax
move 10 to bx
add bx to ax
move 15 to bx
add bx to ax
```

Even in this simple reflection of thoughts in English, there are some key things to observe. One is the concept of destination as every instruction has a "to destination" part and there is a source before it as well. For example the second line has a constant 10 as its source and the register BX as its destination. The key point in giving the first program in English is to convey that the concepts of assembly language are simple but fine. Try to understand them considering that all above is everyday English that you know very well and every concept will eventually be applicable to assembly language.

Assembly Language Version

Intel could have made their assembly language exactly identical to our program in plain English but they have abbreviated a lot of symbols to avoid unnecessarily lengthy program when the meaning could be conveyed with less effort. For example Intel has named their move instruction “mov” instead of “move.” Similarly the Intel order of placing source and destination is opposite to what we have used in our English program, just a change of interpretation. So the Intel way of writing things is:

```
operation destination, source
operation destination
operation source
operation
```

The later three variations are for instructions that have one or both of their operands implied or they work on a single or no operand. An implied operand means that it is always in a particular register say the accumulator, and it need not be mentioned in the instruction. Now we attempt to write our program in actual assembly language of the iapx88.

Example 1.1

```
001  ; a program to add three numbers using registers
002  [org 0x0100]
003                mov  ax, 5             ; load first number in ax
004                mov  bx, 10            ; load second number in bx
005                add  ax, bx            ; accumulate sum in ax
006                mov  bx, 15            ; load third number in bx
007                add  ax, bx            ; accumulate sum in ax
008
009                mov  ax, 0x4c00        ; terminate program
010                int  0x21
```

001	To start a comment a semicolon is used and the assembler ignores everything else on the same line. Comments must be extensively used in assembly language programs to make them readable.
002	Leave the org directive for now as it will be discussed later.
003	The constant 5 is loaded in one register AX.
004	The constant 10 is loaded in another register BX.
005	Register BX is added to register AX and the result is stored in register AX. Register AX should contain 15 by now.
006	The constant 15 is loaded in the register BX.
007	Register BX is again added to register AX now producing 15+15=30 in the AX register. So the program has computed 5+10+15=30.
008	Vertical spacing must also be used extensively in assembly language programs to separate logical blocks of code.
009-010	The ending lines are related more to the operating system than to assembly language programming. It is a way to inform DOS that our program has terminated so it can display its command prompt again. The computer may reboot or behave improperly if this termination is not present.

Assembler, Linker, and Debugger

We need an assembler to assemble this program and convert this into executable binary code. The assembler that we will use during this course is “Netwide Assembler” or NASM. It is a free and open source assembler. And the tool that will be most used will be the debugger. We will use a free debugger called “A fullscreen debugger” or AFD. These are the whole set of weapons an assembly language programmer needs for any task whatsoever at hand.

To assemble we will give the following command to the processor assuming that our input file is named EX01.ASM.

```
nasm ex01.asm -o ex01.com -l ex01.lst
```

This will produce two files EX01.COM that is our executable file and EX01.LST that is a special listing file that we will explore now. The listing file produced for our example above is shown below with comments removed for neatness.

```
     1
     2                                  [org 0x0100]
     3 00000000 B80500                          mov  ax, 5
     4 00000003 BB0A00                          mov  bx, 10
     5 00000006 01D8                            add  ax, bx
     6 00000008 BB0F00                          mov  bx, 15
     7 0000000B 01D8                            add  ax, bx
     8
     9 0000000D B8004C                          mov  ax, 0x4c00
    10 00000010 CD21                            int  0x21
```

The first column in the above listing is offset of the listed instruction in the output file. Next column is the opcode into which our instruction was translated. In this case this opcode is B8. Whenever we move a constant into AX register the opcode B8 will be used. After it 0500 is appended which is the immediate operand to this instruction. An immediate operand is an operand which is placed directly inside the instruction. Now as the AX register is a word sized register, and one hexadecimal digit takes 4 bits so 4 hexadecimal digits make one word or two bytes. Which of the two bytes should be placed first in the instruction, the least significant or the most significant? Similarly for 32bit numbers either the order can be most significant, less significant, lesser significant, and least significant called the big-endian order used by Motorola and some other companies or it can be least significant, more significant, more significant, and most significant called the little-endian order and is used by Intel. The big-endian have the argument that it is more natural to read and comprehend while the little-endian have the argument that this scheme places the less significant value at a lesser address and more significant value at a higher address.

Commented [I1]: LEN used by Intel
BEN used by Motorola

Because of this the constant 5 in our instruction was converted into 0500 with the least significant byte of 05 first and the most significant byte of 00 afterwards. When read as a word it is 0005 but when written in memory it will become 0500. As the first instruction is three bytes long, the listing file shows that the offset of the next instruction in the file is 3. The opcode BB is for moving a constant into the BX register, and the operand 0A00 is the number 10 in little-endian byte order. Similarly the offsets and opcodes of the remaining instructions are shown in order. The last instruction is placed at offset 0x10 or 16 in decimal. The size of the last instruction is two bytes, so the size of the complete COM file becomes 18 bytes. This can be verified from the directory listing, using the DIR command, that the COM file produced is exactly 18 bytes long.

Now the program is ready to be run inside the debugger. The debugger shows the values of registers, flags, stack, our code, and one or two areas of the system memory as data. Debugger allows us to step our program one instruction at a time and observe its effect on the registers and program data. The details of using the AFD debugger can be seen from the AFD manual.

After loading the program in the debugger observe that the first instruction is now at 0100 instead of absolute zero. This is the effect of the org directive at the start of our program. The first instruction of a COM file must be at offset 0100 (decimal 255) as a requirement. Also observe that the debugger is showing your program even though it was provided only the COM file and neither of the listing file or the program source. This is because the translation from mnemonic to opcode is reversible and the debugger mapped back from the opcode to the instruction mnemonic. This will become apparent for instructions that have two mnemonics as the debugger might not show the one that was written in the source file.

As a result of program execution either registers or memory will change. Since our program yet doesn't touch memory the only changes will be in the registers. Keenly observe the registers AX, BX, and IP change after every instruction. IP will change after every instruction to point to the next instruction while AX will accumulate the result of our addition.

1.8. SEGMENTED MEMORY MODEL

Rationale

In earlier processors like 8080 and 8085 the linear memory model was used to access memory. In linear memory model the whole memory appears like a single array of data. 8080 and 8085 could access a total memory of 64K using the 16 lines of their address bus. When designing iAPX88 the Intel designers wanted to remain compatible with 8080 and 8085 however 64K was too small to continue with, for their new processor. To get the best of both worlds they introduced the segmented memory model in 8088.

There is also a logical argument in favor of a segmented memory model in additional to the issue of compatibility discussed above. We have two logical parts of our program, the code and the data, and actually there is a third part called the program stack as well, but higher level languages make this invisible to us. These three logical parts of a program should appear as three distinct units in memory, but making this division is not possible in the linear memory model. The segmented memory model does allow this distinction.

Mechanism

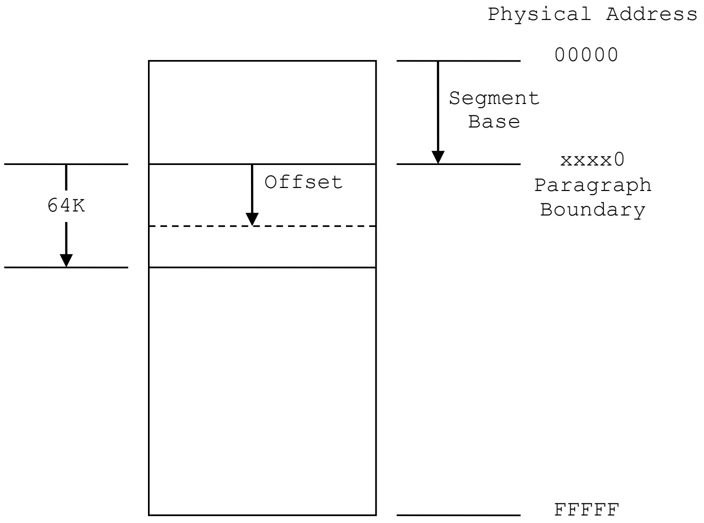
The segmented memory model allows multiple functional windows into the main memory, a code window, a data window etc. The processor sees code from the code window and data from the data window. The size of one window is restricted to 64K. 8085 software fits in just one such window. It sees code, data, and stack from this one window, so downward compatibility is attained.

However the maximum memory iAPX88 can access is 1MB which can be accessed with 20 bits. Compare this with the 64K of 8085 that were accessed using 16 bits. The idea is that the 64K window just discussed can be moved anywhere in the whole 1MB. The four segment registers discussed in the Intel register architecture are used for this purpose. Therefore four windows can exist at one time. For example one window that is pointed to by the CS register contains the currently executing code.

To understand the concept, consider the windows of a building. We say that a particular window is 3 feet above the floor and another one is 20 feet above the floor. The reference point, the floor is the base of the segment called the datum point in a graph and all measurement is done from that datum point considering it to be zero. So CS holds the zero or the base of code. DS holds the zero of data. Or we can say CS tells how high code from the floor is, and DS tells how high data from the floor is, while SS tells how high the stack is. One extra segment ES can be used if we need to access two distant areas of memory at the same time that both cannot be seen through the same window. ES also has special role in string instructions. ES is used as an extra data segment and cannot be used as an extra code or stack segment.

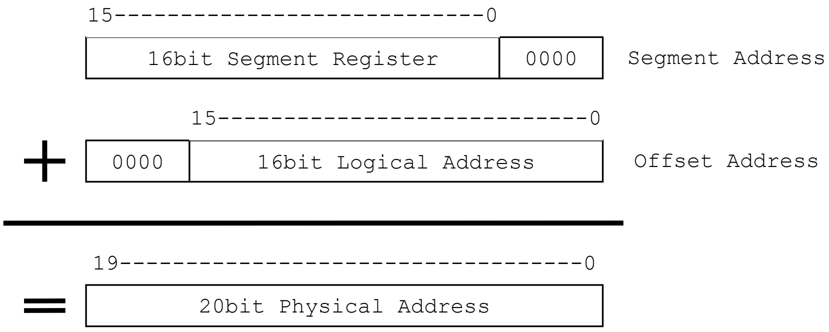
Revisiting the concept again, like the datum point of a graph, the segment registers tell the start of our window which can be opened anywhere in the megabyte of memory available. The window is of a fixed size of 64KB. Base and offset are the two key variables in a segmented address. Segment tells the base while offset is added into it. The registers IP, SP, BP, SI, DI, and BX all can contain a 16bit offset in them and access memory relative to a segment base.

The IP register cannot work alone. It needs the CS register to open a 64K window in the 1MB memory and then IP works to select code from this window as offsets. IP works only inside this window and cannot go outside of this 64K in any case. If the window is moved i.e. the CS register is changed, IP will change its behavior accordingly and start selecting from the new window. The IP register always works relatively, relative to the segment base stored in the CS register. IP is a 16bit register capable of accessing only 64K memory so how the whole megabyte can contain code anywhere. Again the same concept is there, it can access 64K at one instance of time. As the base is changed using the CS register, IP can be made to point anywhere is the whole megabyte. The process is illustrated with the following diagram.



Physical Address Calculation

Now for the whole megabyte we need 20 bits while CS and IP are both 16bit registers. We need a mechanism to make a 20bit number out of the two 16bit numbers. Consider that the segment value is stored as a 20 bit number with the lower four bits zero and the offset value is stored as another 20 bit number with the upper four bits zeroed. The two are added to produce a 20bit absolute address. A carry if generated is dropped without being stored anywhere and the phenomenon is called address wraparound. The process is explained with the help of the following diagram.



Therefore memory is determined by a segment-offset pair and not alone by any one register which will be an ambiguous reference. Every offset register is assigned a default segment register to resolve such ambiguity. For example the program we wrote when loaded into memory had a value of 0100 in IP register and some value say 1DDD in the CS register. Making both 20 bit numbers, the segment base is 1DDD0 and the offset is 00100 and adding them we get the physical memory address of 1DED0 where the opcode B80500 is placed.

Paragraph Boundaries

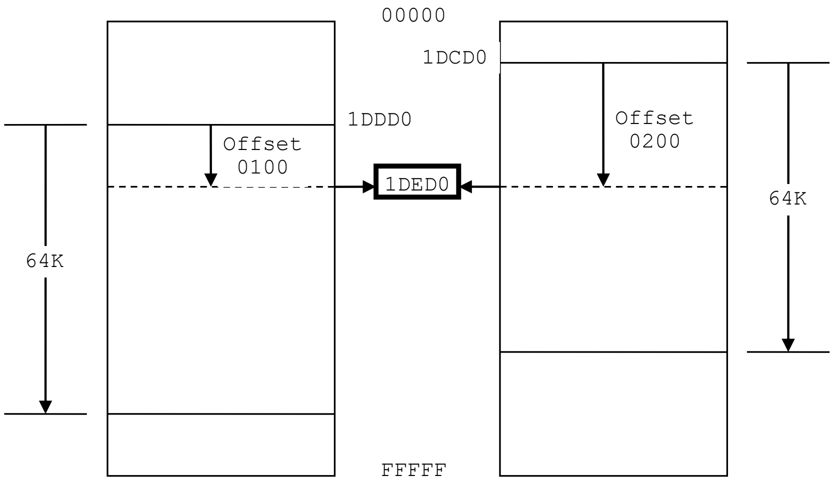
As the segment value is a 16bit number and four zero bits are appended to the right to make it a 20bit number, segments can only be defined a 16byte boundaries called paragraph boundaries. The first possible segment value is 0000 meaning a physical base of 00000 and the next possible value of 0001 means a segment base of 00010 or 16 in decimal. Therefore segments can only be defined at 16 byte boundaries.

Overlapping Segments

We can also observe that in the case of our program CS, DS, SS, and ES all had the same value in them. This is called overlapping segments so that we can see the same memory from any window. This is the structure of a COM file.

Using partially overlapping segments we can produce a number of segment, offset pairs that all access the same memory. For example 1DDD:0100 and IDED:0000 both point to the same physical memory. To test this we can open a data window at 1DED:0000 in the debugger and change the first three bytes to “90” which is the opcode for NOP (no operation). The change is immediately visible in the code window which is pointed to by CS containing 1DDD. Similarly IDCD:0200 also points to the same memory location. Consider this like a portion of wall that three different people on three different floors are seeing through their own windows. One of them painted the wall red; it will be changed for all of them though their perspective is different. It is the same phenomenon occurring here.

The segment, offset pair is called a logical address, while the 20bit address is a physical address which is the real thing. Logical addressing is a mechanism to access the physical memory. As we have seen three different logical addresses accessed the same physical address.



EXERCISES

1. How the processor uses the address bus, the data bus, and the control bus to communicate with the system memory?
2. Which of the following are unidirectional and which are bidirectional?
   a. Address Bus
   b. Data Bus
   c. Control Bus
3. What are registers and what are the specific features of the accumulator, index registers, program counter, and program status word?
4. What is the size of the accumulator of a 64bit processor?
5. What is the difference between an instruction mnemonic and its opcode?
6. How are instructions classified into groups?
7. A combination of 8bits is called a byte. What is the name for 4bits and for 16bits?
8. What is the maximum memory 8088 can access?
9. List down the 14 registers of the 8088 architecture and briefly describe their uses.
10. What flags are defined in the 8088 FLAGS register? Describe the function of the zero flag, the carry flag, the sign flag, and the overflow flag.
11. Give the value of the zero flag, the carry flag, the sign flag, and the overflow flag after each of the following instructions if AX is initialized with 0x1254 and BX is initialized with 0x0FFF.
    a. add ax, 0xEDAB
    b. add ax, bx
    c. add bx, 0xF001
12. What is the difference between little endian and big endian formats? Which format is used by the Intel 8088 microprocessor?
13. For each of the following words identify the byte that is stored at lower memory address and the byte that is stored at higher memory address in a little endian computer.
    a. 1234
    b. ABFC
    c. B100
    d. B800

14. What are the contents of memory locations 200, 201, 202, and 203 if the word 1234 is stored at offset 200 and the word 5678 is stored at offset 202?
15. What is the offset at which the first exeutable instruction of a COM file must be placed?
16. Why was segmentation originally introduced in 8088 architecture?
17. Why a segment start cannot start from the physical address 55555.
18. Calculate the physical memory address generated by the following segment offset pairs.
    a. 1DDD:0436
    b. 1234:7920
    c. 74F0:2123
    d. 0000:6727
    e. FFFF:4336
    f. 1080:0100
    g. AB01:FFFF
19. What are the first and the last physical memory addresses accessible using the following segment values?
    a. 1000
    b. 0FFF
    c. 1002
    d. 0001
    e. E000
20. Write instructions that perform the following operations.
    a. Copy BL into CL
    b. Copy DX into AX
    c. Store 0x12 into AL
    d. Store 0x1234 into AX
    e. Store 0xFFFF into AX
21. Write a program in assembly language that calculates the square of six by adding six to the accumulator six times.

2
Addressing Modes

2.1. DATA DECLARATION

The first instruction of our first assembly language program was “mov ax, 5.” Here MOV was the opcode, AX was the destination operand, while 5 was the source operand. The value of 5 in this case was stored as part of the instruction encoding. In the opcode B80500, B8 was the opcode and 0500 was the operand stored immediately afterwards. Such an operand is called an immediate operand. It is one of the many types of operands available.

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.

2.2. DIRECT ADDRESSING

Now we will rewrite our first program such that the numbers 5, 10, and 15 are stored as memory variables instead of constants and we access them from there.

Example 2.1

```
; a program to add three numbers using memory variables
[org 0x0100]
              mov  ax, [num1]         ; load first number in ax
```

```
004                 mov  bx, [num2]         ; load second number in bx
005                 add  ax, bx             ; accumulate sum in ax
006                 mov  bx, [num3]         ; load third number in bx
007                 add  ax, bx             ; accumulate sum in ax
008                 mov  [num4], ax         ; store sum in num4
009
010                 mov  ax, 0x4c00         ; terminate program
011                 int  0x21
012
013 num1:           dw   5
014 num2:           dw   10
015 num3:           dw   15
016 num4:           dw   0
```

002 Originate our program at 0100. The first executable instruction should be placed at this offset.

003 The source operand is changed from constant 5 to [num1]. The bracket is signaling that the operand is placed in memory at address num1. The value 5 will be loaded in ax even though we did not specified it in our program code, rather the value will be picked from memory. The instruction should be read as “read the contents of memory location num1 in the ax register.” The label num1 is a symbol for us but an address for the processor while the conversion is done by the assembler.

013 The label num1 is defined as a word and the assembler is requested to place 5 in that memory location. The colon signals that num1 is a label and not an instruction.

Using the same process to assemble as discussed before we examine the listing file generated as a result with comments removed.

```
     1
     2                                  [org 0x0100]
     3 00000000 A1[1700]                              mov  ax, [num1]
     4 00000003 8B1E[1900]                            mov  bx, [num2]
     5 00000007 01D8                                  add  ax, bx
     6 00000009 8B1E[1B00]                            mov  bx, [num3]
     7 0000000D 01D8                                  add  ax, bx
     8 0000000F A3[1D00]                              mov  [num4], ax
     9
    10 00000012 B8004C                                mov  ax, 0x4c00
    11 00000015 CD21                                  int  0x21
    12
    13 00000017 0500                    num1:         dw   5
    14 00000019 0A00                    num2:         dw   10
    15 0000001B 0F00                    num3:         dw   15
    16 0000001D 0000                    num4:         dw   0
```

The first instruction of our program has changed from B80500 to A11700. The opcode B8 is used to move constants into AX, while the opcode A1 is used when moving data into AX from memory. The immediate operand to our new instruction is 1700 or as a word 0017 (23 decimal) and from the bottom of the listing file we can observe that this is the offset of num1. The assembler has calculated the offset of num1 and used it to replace references to num1 in the whole program. Also the value 0500 can be seen at offset 0017 in the file. We can say contents of memory location 0017 are 0005 as a word. Similarly num2, num3, and num4 are placed at 0019, 001B, and 001D addresses.

When the program is loaded in the debugger, it is loaded at offset 0100, which displaces all memory accesses in our program. The instruction A11700 is changed to A11701 meaning that our variable is now placed at 0117 offset. The instruction is shown as mov ax, [0117]. Also the data window can be used to verify that offset 0117 contains the number 0005.

Execute the program step by step and examine how the memory is read and the registers are updated, how the instruction pointer moves forward, and how the result is saved back in memory. Also observe inside the debugger code window below the code for termination, that the debugger is interpreting our data as code and showing it as some meaningless instructions. This is because the debugger sees everything as code in the code window and cannot differentiate our declared data from opcodes. It is our responsibility that we terminate execution before our data is executed as code.

Also observe that our naming of num1, num2, num3, and num4 is no longer there inside the debugger. The debugger is only showing the numbers 0117, 0119, 011B, and 011D. Our numerical machine can only work with numbers. We used symbols for our ease to label or tag certain positions in our program. The assembler converts these symbols into the appropriate numbers automatically. Also observe that the effect of “dw” is to place 5 in two bytes as 0005. Had we used “db” this would have been stored as 05 in one byte.

Given the fact that the assembler knows only numbers we can write the same program using a single label. As we know that num2 is two ahead of num1, we can use num1+2 instead of num2 and let the assembler calculate the sum during assembly process.

Example 2.2

```
001  ; a program to add three numbers accessed using a single label
002  [org 0x0100]
003                mov  ax, [num1]         ; load first number in ax
004                mov  bx, [num1+2]       ; load second number in bx
005                add  ax, bx             ; accumulate sum in ax
006                mov  bx, [num1+4]       ; load third number in bx
007                add  ax, bx             ; accumulate sum in ax
008                mov  [num1+6], ax       ; store sum at num1+6
009
010                mov  ax, 0x4c00         ; terminate program
011                int  0x21
012
013  num1:         dw   5
014                dw   10
015                dw   15
016                dw   0
```

004	The second number is read from num1+2. Similarly the third number is read from num1+4 and the result is accessed at num1+6.
013-016	The labels num2, num3, and num4 are removed and the data there will be accessed with reference to num1.

Every location is accessed with reference to num1 in this example. The expression “num1+2” comprises of constants only and can be evaluated at the time of assembly. There are no variables involved in this expression. As we open the program inside the debugger we see a verbatim copy of the previous program. There is no difference at all since the assembler catered for the differences during assembly. It calculated 0117+2=0119 while in the previous it directly knew from the value of num2 that it has to write 0119, but the end result is a ditto copy of the previous execution.

Another way to declare the above data and produce exactly same results is shown in the following example.

Example 2.3

```
001  ; a program to add three numbers accessed using a single label
002  [org 0x0100]
003                mov  ax, [num1]         ; load first number in ax
004                mov  bx, [num1+2]       ; load second number in bx
```

```
005               add  ax, bx              ; accumulate sum in ax
006               mov  bx, [num1+4]        ; load third number in bx
007               add  ax, bx              ; accumulate sum in ax
008               mov  [num1+6], ax        ; store sum at num1+6
009
010               mov  ax, 0x4c00          ; terminate program
011               int  0x21
012
013 num1:         dw   5, 10, 15, 0
```

013 As we do not need to place labels on individual variables we can save space and declare all data on a single line separated by commas. This declaration will declare four words in consecutive memory locations while the address of first one is num1.

The method used to access memory in the above examples is called direct addressing. In direct addressing the memory address is fixed and is given in the instruction. The actual data used is placed in memory and now that data can be used as the destination operand as well. Also the source and destination operands must have the same size. For example a word defined memory is read in a word sized register. A last observation is that the data 0500 in memory was corrected to 0005 when read in a register. So registers contain data in proper order as a word.

A last variation using direct addressing shows that we can directly add a memory variable and a register instead of adding a register into another that we were doing till now.

Example 2.4

```
; a program to add three numbers directly in memory
[org 0x0100]
              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

              mov  ax, 0x4c00          ; terminate program
              int  0x21

num1:         dw   5, 10, 15, 0
```

We generate the following listing file as a result of the assembly process described previously. Comments are again removed.

```
 1
 2                                 [org 0x0100]
 3 00000000 A1[1900]                             mov  ax, [num1]
 4 00000003 A3[1F00]                             mov  [num1+6], ax
 5 00000006 A1[1B00]                             mov  ax, [num1+2]
 6 00000009 0106[1F00]                           add  [num1+6], ax
 7 0000000D A1[1D00]                             mov  ax, [num1+4]
 8 00000010 0106[1F00]                           add  [num1+6], ax
 9
10 00000014 B8004C                               mov  ax, 0x4c00
11 00000017 CD21                                 int  0x21
12
13 00000019 05000A000F000000        num1:         dw   5, 10, 15, 0
```

The opcode of add is changed because the destination is now a memory location instead of a register. No other significant change is seen in the listing file. Inside the debugger we observe that few opcodes are longer now and the location num1 is now translating to 0119 instead of 0117. This is done automatically by the assembler as a result of using labels instead of hard coding addresses. During execution we observe that the word data as it is read into a register is read in correct order. The significant change in this example is that the destination of addition is memory. Method to access memory is direct addressing, whether it is the MOV instruction or the ADD instruction.

The first two instructions of the last program read a number into AX and placed it at another memory location. A quick thought reveals that the following might be a possible single instruction to replace the couple.

```
        mov  [num1+6], [num1]   ; ILLEGAL
```

However this form is illegal and not allowed on the Intel architecture. None of the general operations of mov, add, sub etc. allow moving data from memory to memory. Only register to register, register to memory, memory to register, constant to memory, and constant to register operations are allowed. The other register to constant, memory to constant, and memory to memory are all disallowed. Only string instructions allow moving data from memory to memory and will be discussed in detail later. As a rule one instruction can have at most one operand in brackets, otherwise assembler will give an error.

2.3. SIZE MISMATCH ERRORS

If we change the directive in the last example from DW to DB, the program will still assemble and debug without errors, however the results will not be the same as expected. When the first operand is read 0A05 will be read in the register which was actually two operands place in consecutive byte memory locations. The second number will be read as 000F which is the zero byte of num4 appended to the 15 of num3. The third number will be junk depending on the current state of the machine. According to our data declaration the third number should be at 0114 but it is accessed at 011D calculated with word offsets. This is a logical error of the program. To keep the declarations and their access synchronized is the responsibility of the programmer and not the assembler. The assembler allows the programmer to do everything he wants to do, and that can possibly run on the processor. The assembler only keeps us from writing illegal instructions which the processor cannot execute. This is the difference between a syntax error and a logic error. So the assembler and debugger have both done what we asked them to do but the programmer asked them to do the wrong chore.

The programmer is responsible for accessing the data as word if it was declared as a word and accessing it as a byte if it was declared as a byte. The word case is shown in lot of previous examples. If however the intent is to treat it as a byte the following code shows the appropriate way.

Example 2.5

```
001  ; a program to add three numbers using byte variables
002  [org 0x0100]
003                mov  al, [num1]         ; load first number in al
004                mov  bl, [num1+1]       ; load second number in bl
005                add  al, bl             ; accumulate sum in al
006                mov  bl, [num1+2]       ; load third number in bl
007                add  al, bl             ; accumulate sum in al
008                mov  [num1+3], al       ; store sum at num1+3
009
010                mov  ax, 0x4c00         ; terminate program
011                int  0x21
012
013  num1:         db   5, 10, 15, 0
```

003 The number is read in AL register which is a byte register since the memory location read is also of byte size.

005 The second number is now placed at num1+1 instead of num1+2 because of byte offsets.

To declare data db is used instead of dw so that each data declared occupies one byte only.

Inside the debugger we observe that the AL register takes appropriate values and the sum is calculated and stored in num1+3. This time there is no alignment or synchronization error. The key thing to understand here is that the processor does not match defines to accesses. It is the programmer’s responsibility. In general assembly language gives a lot of power to the programmer but power comes with responsibility. Assembly language programming is not a difficult task but a responsible one.

In the above examples, the processor knew the size of the data movement operation from the size of the register involved, for example in “mov ax, [num1]” memory can be accessed as byte or as word, it has no hard and fast size, but the AX register tells that this operation has to be a word operation. Similarly in “mov al, [num1]” the AL register tells that this operation has to be a byte operation. However in “mov ax, bl” the AX register tells that the operation has to be a word operation while BL tells that this has to be a byte operation. The assembler will declare that this is an illegal instruction. A 5Kg bag cannot fit inside a 1Kg bag and according to Intel a 1Kg cannot also fit in a 5Kg bag. They must match in size. The instruction “mov [num1], [num2]” is illegal as previously discussed not because of data movement size but because memory to memory moves are not allowed at all.

The instruction “mov [num1], 5” is legal but there is no way for the processor to know the data movement size in this operation. The variable num1 can be treated as a byte or as a word and similarly 5 can be treated as a byte or as a word. Such instructions are declared ambiguous by the assembler. The assembler has no way to guess the intent of the programmer as it previously did using the size of the register involved but there is no register involved this time. And memory is a linear array and label is an address in it. There is no size associated with a label. Therefore to resolve its ambiguity we clearly tell our intent to the assembler in one of the following ways.

```
mov  byte [num1], 5
mov  word [num1], 5
```

2.4. REGISTER INDIRECT ADDRESSING

We have done very elementary data access till now. Assume that the numbers we had were 100 and not just three. This way of adding them will cost us 200 instructions. There must be some method to do a task repeatedly on data placed in consecutive memory cells. The key to this is the need for some register that can hold the address of data. So that we can change the address to access some other cell of memory using the same instruction. In direct addressing mode the memory cell accessed was fixed inside the instruction. There is another method in which the address can be placed in a register so that it can be changed. For the following example we will take 10 instead of 100 numbers but the algorithm is extensible to any size.

There are four registers in iAPX88 architecture that can hold address of data and they are BX, BP, SI, and DI. There are minute differences in their working which will be discussed later. For the current example, we will use the BX register and we will take just three numbers and extend the concept with more numbers in later examples.

Example 2.6

```
; a program to add three numbers using indirect addressing
[org 0x100]
              mov  bx, num1           ; point bx to first number
              mov  ax, [bx]           ; load first number in ax
              add  bx, 2              ; advance bx to second number
```

```
006                 add  ax, [bx]           ; add second number to ax
007                 add  bx, 2              ; advance bx to third number
008                 add  ax, [bx]           ; add third number to ax
009                 add  bx, 2              ; advance bx to result
010                 mov  [bx], ax           ; store sum at num1+6
011
012                 mov  ax, 0x4c00         ; terminate program
013                 int  0x21
014
015     num1:       dw   5, 10, 15, 0
```

003 Observe that no square brackets around num1 are used this time. The address is loaded in bx and not the contents. Value of num1 is 0005 and the address is 0117. So BX will now contain 0117.

004 Brackets are now used around BX. In iapx88 architecture brackets can be used around BX, BP, SI, and DI only. In iapx386 more registers are allowed. The instruction will be read as “move into ax the contents of the memory location whose address is in bx.” Now since bx contains the address of num1 the contents of num1 are transferred to the ax register. Without square brackets the meaning of the instruction would have been totally different.

005 This instruction is changing the address. Since we have words not bytes, we add two to bx so that it points to the next word in memory. BX now contains 0119 the address of the second word in memory. This was the mechanism to change addresses that we needed.

Inside the debugger we observe that the first instruction is “mov bx, 011C.” A constant is moved into BX. This is because we did not use the square brackets around “num1.” The address of “num1” has moved to 011C because the code size has changed due to changed instructions. In the second instruction BX points to 011C and the value read in AX is 0005 which can be verified from the data window. After the addition BX points to 011E containing 000A, our next word, and so on. This way the BX register points to our words one after another and we can add them using the same instruction “mov ax, [bx]” without fixing the address of our data in the instructions. We can also subtract from BX to point to previous cells. The address to be accessed is now in total program control.

One thing that we needed in our problem to add hundred numbers was the capability to change address. The second thing we need is a way to repeat the same instruction and a way to know that the repetition is done a 100 times, a terminal condition for the repetition. For the task we are introducing two new instructions that you should read and understand as simple English language concepts. For simplicity only 10 numbers are added in this example. The algorithm is extensible to any size.

Example 2.7

```
001     ; a program to add ten numbers
002     [org 0x0100]
003                 mov  bx, num1           ; point bx to first number
004                 mov  cx, 10             ; load count of numbers in cx
005                 mov  ax, 0              ; initialize sum to zero
006
007     l1:         add  ax, [bx]           ; add number to ax
008                 add  bx, 2              ; advance bx to next number
009                 sub  cx, 1              ; numbers to be added reduced
010                 jnz  l1                 ; if numbers remain add next
011
012                 mov  [total], ax        ; write back sum in memory
013
014                 mov  ax, 0x4c00         ; terminate program
015                 int  0x21
016
017     num1:       dw   10, 20, 30, 40, 50, 10, 20, 30, 40, 50
```

018	`total:        dw   0`
006	Labels can be used on code as well. Just like data labels they remember the address at which they are used. The assembler does not differentiate between code labels and data labels. The programmer is responsible for using a data label as data and a code label as code. The label l1 in this case is the address of the following instruction.
009	SUB is the counterpart to ADD with the same rules as that of the ADD instruction.
010	JNZ stands for “jump if not zero.” NZ is the condition in this instruction. So the instruction is read as “jump to the location l1 if the zero flag is not set.” And revisiting the zero flag definition “the zero flag is set if the last mathematical or logical operation has produced a zero in its destination.” For example “mov ax, 0” will not set the zero flag as it is not a mathematical or logical instruction. However subtraction and addition will set it. Also it is set even when the destination is not a register. Now consider the subtraction immediately preceding it. If the CX register becomes zero as a result of this subtraction the zero flag will be set and the jump will be taken. And jump to l1, the processor needs to be told each and everything and the destination is an important part of every jump. Just like when we ask someone to go, we mention go to this market or that house. The processor is much more logical than us and needs the destination in every instruction that asks it to go somewhere. The processor will load l1 in the IP register and resume execution from there. The processor will blindly go to the label we mention even if it contains data and not code.

The CX register is used as a counter in this example, BX contains the changing address, while AX accumulates the result. We have formed a loop in assembly language that executes until its condition remains true. Inside the debugger we can observe that the subtract instruction clears the zero flag the first nine times and sets it on the tenth time. While the jump instruction moves execution to address l1 the first nine times and to the following line the tenth time. The jump instruction breaks program flow.

The JNZ instruction is from the program control group and is a conditional jump, meaning that if the condition NZ is true (ZF=0) it will jump to the address mentioned and otherwise it will progress to the next instruction. It is a selection between two paths. If the condition is true go right and otherwise go left. Or we can say if the weather is hot, go this way, and if it is cold, go this way. Conditional jump is the most important instruction, as it gives the processor decision making capability, so it must be given a careful thought. Some processors call it branch, probably a more logical name for it, however the functionality is same. Intel chose to name it “jump.”

An important thing in the above example is that a register is used to reference memory so this form of access is called register indirect memory access. We used the BX register for it and the B in BX and BP stands for base therefore we call register indirect memory access using BX or BP, “based addressing.” Similarly when SI or DI is used we name the method “indexed addressing.” They have the same functionality, with minor differences because of which the two are called base and index. The differences will be explained later, however for the above example SI or DI could be used as well, but we would name it indexed addressing instead of based addressing.

2.5. REGISTER + OFFSET ADDRESSING

Direct addressing and indirect addressing using a single register are two basic forms of memory access. Another possibility is to use different combinations of direct and indirect references. In the above example we used BX to access different array elements which were placed consecutively in memory like an array. We can also place in BX only the array index and not the exact address and form the exact address when we are going to access the actual memory. This way the same register can be used for accessing different arrays and also the register can be used for index comparison like the following example does.

Example 2.8

```
001  ; a program to add ten numbers using register + offset addressing
002  [org 0x0100]
003                mov  bx, 0              ; initialize array index to zero
004                mov  cx, 10             ; load count of numbers in cx
005                mov  ax, 0              ; initialize sum to zero
006
007  l1:           add  ax, [num1+bx]      ; add number to ax
008                add  bx, 2              ; advance bx to next index
009                sub  cx, 1              ; numbers to be added reduced
010                jnz  l1                 ; if numbers remain add next
011
012                mov  [total], ax        ; write back sum in memory
013
014                mov  ax, 0x4c00         ; terminate program
015                int  0x21
016
017  num1:         dw   10, 20, 30, 40, 50, 10, 20, 30, 40, 50
018  total:        dw   0
```

003 This time BX is initialized to zero instead of array base

007 The format of memory access has changed. The array base is added to BX containing array index at the time of memory access.

008 As the array is of words, BX jumps in steps of two, i.e. 0, 2, 4. Higher level languages do appropriate incrementing themselves and we always use sequential array indexes. However in assembly language we always calculate in bytes and therefore we need to take care of the size of one array element which in this case is two.

Inside the debugger we observe that the memory access instruction is shown as “mov ax, [011F+bx]” and the actual memory accessed is the one whose address is the sum of 011F and the value contained in the BX register. This form of access is of the register indirect family and is called base + offset or index + offset depending on whether BX or BP is used or SI or DI is used.

2.6. SEGMENT ASSOCIATION

All the addressing mechanisms in iAPX88 return a number called *effective address*. For example in base + offset addressing, neither the base nor the offset alone tells the desired cell in memory to be accessed. It is only after the addition is done that the processor knows which cell to be accessed. This number which came as the result of addition is called the effective address. But the effective address is just an offset and is meaningless without a segment. Only after the segment is known, we can form the physical address that is needed to access a memory cell.

We discussed the segmented memory model of iAPX88 in reasonable detail at the end of previous chapter. However during the discussion of addressing modes we have not seen the effect of segments. Segmentation is there and it’s all happening relative to a segment base. We saw DS, CS, SS, and ES inside the debugger. Everything is relative to its segment base, even though we have not explicitly explained its functionality. An offset alone is not complete without a segment. As previously discussed there is a default segment associated to every register which accesses memory. For example CS is associated to IP by default; rather it is tied with it. It cannot access memory in any other segment.

In case of data, there is a bit relaxation and nothing is tied. Rather there is a default association which can be overridden. In the case of register indirect memory access, if the register used is one of SI, DI, or BX the default segment is DS. If however the register used in BP the default segment used is SS. The stack segment has a very critical and fine use and there is a reason why BP is attached to SS by default. However these will be discussed in detail in the chapter on stack. IP is tied to CS while SP is tied to SS. The association of these registers cannot be changed; they are locked with no option. Others are not locked and can be changed.

To override the association for one instruction of one of the registers BX, BP, SI or DI, we use the segment override prefix. For example "mov ax, [cs:bx]" associates BX with CS for this one instruction. For the next instruction the default association will come back to act. The processor places a special byte before the instruction called a prefix, just like prefixes and suffixes in English language. No prefix is needed or placed for default association. For example for CS the byte 2E is placed and for ES the byte 26 is placed. Opcode has not changed, but the prefix byte has modified the default association to association with the desired segment register for this one instruction.

In all our examples, we never declared a segment or used it explicitly, but everything seemed to work fine. The important thing to note is that CS, DS, SS, and ES all had the same value. The value itself is not important but the fact that all had the same value is important. All four segment windows exactly overlap. Whatever segment register we use the same physical memory will be accessed. That is why everything was working without the mention of a single segment register. This is the formation of COM files in IBM PC. A single segment contains code, data, and the stack. This format is operating system dependant, in our case defined by DOS. And our operating system defines the format of COM files such that all segments have the same value. Thus the only meaningful thing that remains is the offset.

For example if BX=0100, SI=0200, and CS=1000 and the memory access under consideration is [cs:bx+si+0x0700], the effective address formed is bx+si+0700 = 0100 + 0200 + 0700 = 0A00. Now multiplying the segment value by 16 makes it 10000 and adding the effective address 00A00 forms the physical address 10A00.

2.7. ADDRESS WRAPAROUND

There are two types of wraparounds. One is within a single segment and the other is inside the whole physical memory. Segment wraparound occurs when during the effective address calculation a carry is generated. This carry is dropped giving the effect that when we try to access beyond the segment limit, we are actually wrapped around to the first cell in the segment. For example if BX=9100, DS=1500 and the access is [bx+0x7000] we form the effective address 9100 + 7000 = 10100. The carry generated is dropped forming the actual effective address of 0100. Just like a circle when we reached the end we started again from the beginning. An arc at 370 degrees is the same as an arc at 10 degrees. We tried to cross the segment boundary and it pushed us back to the start. This is called segment wraparound. The physical address in the above example will be 15100.

The same can also happen at the time of physical address calculation. For example BX=0100, DS=FFF0 and the access under consideration is [bx+0x0100]. The effective address will be 0200 and the physical address will be 100100. This is a 21bit answer and cannot be sent on the address bus which is 20 bits wide. The carry is dropped and just like the segment wraparound our physical memory has wrapped around at its very top. When we tried to access beyond limits the actual access is made at the very start. This second wraparound is a bit different in newer processor with more address lines but that will be explained in later chapters.

2.8. ADDRESSING MODES SUMMARY

The iAPX88 processor supports seven modes of memory access. Remember that immediate is not an addressing mode but an operand type. Operands can be immediate, register, or memory. If the operand is memory one of the seven addressing modes will be used to access it. The memory access mechanisms can also be written in the general form “base + index + offset” and we can define the possible addressing modes by saying that any one, two, or none can be skipped from the general form to form a legal memory access.

There are a few common mistakes done in forming a valid memory access. Part of a register cannot be used to access memory. Like BX is allowed to hold an address but BL or BH are not. Address is 16bit and must be contained in a 16bit register. BX-SI is not possible. The only thing that we can do is addition of a base register with an index register. Any other operation is disallowed. BS+BP and SI+DI are both disallowed as we cannot have two base or two index registers in one memory access. One has to be a base register and the other has to be an index register and that is the reason of naming them differently.

Direct

A fixed offset is given in brackets and the memory at that offset is accessed. For example “mov [1234], ax” stores the contents of the AX registers in two bytes starting at address 1234 in the current data segment. The instruction “mov [1234], al” stores the contents of the AL register in the byte at offset 1234.

Based Register Indirect

A base register is used in brackets and the actual address accessed depends on the value contained in that register. For example “mov [bx], ax” moves the two byte contents of the AX register to the address contained in the BX register in the current data segment. The instruction “mov [bp], al” moves the one byte content of the AL register to the address contained in the BP register in the current stack segment.

Indexed Register Indirect

An index register is used in brackets and the actual address accessed depends on the value contained in that register. For example “mov [si], ax” moves the contents of the AX register to the word starting at address contained in SI in the current data segment. The instruction “mov [di], ax” moves the word contained in AX to the offset stored in DI in the current data segment.

Based Register Indirect + Offset

A base register is used with a constant offset in this addressing mode. The value contained in the base register is added with the constant offset to get the effective address. For example “mov [bx+300], ax” stores the word contained in AX at the offset attained by adding 300 to BX in the current data segment. The instruction “mov [bp+300], ax” stores the word in AX to the offset attained by adding 300 to BP in the current stack segment.

Indexed Register Indirect + Offset

An index register is used with a constant offset in this addressing mode. The value contained in the index register is added with the constant offset to get the effective address. For example "mov [si+300], ax" moves the word contained in AX to the offset attained by adding 300 to SI in the current data segment and the instruction "mov [di+300], al" moves the byte contained in AL to the offset attained by adding 300 to DI in the current data segment.

Base + Index

One base and one index register is used in this addressing mode. The value of the base register and the index register are added together to get the effective address. For example "mov [bx+si], ax" moves the word contained in the AX register to offset attained by adding BX and SI in the current data segment. The instruction "mov [bp+di], al" moves the byte contained in AL to the offset attained by adding BP and DI in the current stack segment. Observe that the default segment is based on the base register and not on the index register. This is why base registers and index registers are named separately. Other examples are "mov [bx+di], ax" and "mov [bp+si], ax." This method can be used to access a two dimensional array such that one dimension is in a base register and the other is in an index register.

Base + Index + Offset

This is the most complex addressing method and is relatively infrequently used. A base register, an index register, and a constant offset are all used in this addressing mode. The values of the base register, the index register, and the constant offset are all added together to get the effective address. For example "mov [bx+si+300], ax" moves the word contents of the AX register to the word in memory starting at offset attained by adding BX, SI, and 300 in the current data segment. Default segment association is again based on the base register. It might be used with the array base of a two dimensional array as the constant offset, one dimension in the base register and the other in the index register. This way all calculation of location of the desired element has been delegated to the processor.

EXERCISES

1. What is a label and how does the assembler differentiates between code labels and data labels?
2. List the seven addressing modes available in the 8088 architecture.
3. Differentiate between effective address and physical address.
4. What is the effective address generated by the following instructions? Every instruction is independent of others. Initially BX=0x0100, num1=0x1001, [num1]=0x0000, and SI=0x0100
    a. mov ax, [bx+12]
    b. mov ax, [bx+num1]
    c. mov ax, [num1+bx]
    d. mov ax, [bx+si]
5. What is the effective address generated by the following combinations if they are valid. If not give reason. Initially BX=0x0100, SI=0x0010, DI=0x0001, BP=0x0200, and SP=0xFFFF
    a. bx-si
    b. bx-bp
    c. bx+10
    d. bx-10
    e. bx+sp
    f. bx+di
6. Identify the problems in the following instructions and correct them by replacing them with one or two instruction having the same effect.

   a. mov [02], [22]
   b. mov [wordvar], 20
   c. mov bx, al
   d. mov ax, [si+di+100]

7. What is the function of segment override prefix and what changes it brings to the opcode?
8. What are the two types of address wraparound? What physical address is accessed with [BX+SI] if FFFF is loaded in BX, SI, and DS.
9. Write instructions to do the following.
   a. Copy contents of memory location with offset 0025 in the current data segment into AX.
   b. Copy AX into memory location with offset 0FFF in the current data segment.
   c. Move contents of memory location with offset 0010 to memory location with offset 002F in the current data segment.
10. Write a program to calculate the square of 20 by using a loop that adds 20 to the accumulator 20 times.

3
Branching

3.1. COMPARISON AND CONDITIONS

Conditional jump was introduced in the last chapter to loop for the addition of a fixed number of array elements. The jump was based on the zero flag. There are many other conditions possible in a program. For example an operand can be greater than another operand or it can be smaller. We use comparisons and boolean expressions extensively in higher level languages. They must be available is some form in assembly language, otherwise they could not possibly be made available in a higher level language. In fact they are available in a very fine and purified form.

The basic root instruction for all comparisons is CMP standing for compare. The operation of CMP is to subtract the source operand from the destination operand, updating the flags without changing either the source or the destination. CMP is one of the key instructions as it introduces the capability of conditional routing in the processor.

A closer thought reveals that with subtraction we can check many different conditions. For example if a larger number is subtracted from a smaller number a borrow is needed. The carry flag plays the role of borrow during the subtraction operation. And in this condition the carry flag will be set. If two equal numbers are subtracted the answer is zero and the zero flag will be set. Every significant relation between the destination and source is evident from the sign flag, carry flag, zero flag, and the overflow flag. CMP is meaningless without a conditional jump immediately following it.

Another important distinction at this point is the difference between signed and unsigned numbers. In unsigned numbers only the magnitude of the number is important, whereas in signed numbers both the magnitude and the sign are important. For example -2 is greater than -3 but 2 is smaller than 3. The sign has affected our comparisons.

Inside the computer signed numbers are represented in two’s complement notation. In essence a number in this representation is still a number, just that now our interpretation of this number will be signed. Whether we use jump above and below or we use jump greater or less will convey our intention to the processor. The jump above and greater operations at first sight seem to be doing the same operation, and similarly below and less operations seem to be similar. However for signed numbers JG and JL will work properly and for unsigned JA and JB will work properly and not the other way around.

It is important to note that at the time of comparison, the intent of the programmer to treat the numbers as signed or unsigned is not clear. The subtraction in CMP is a normal subtraction. It is only after the comparison, during the conditional jump operation, that the intent is conveyed. At that time with a specific combination of flags checked the intent is satisfied.

For example a number 2 is represented in a word as 0002 while the number -2 is represented as FFFE. In a byte they would be represented as 02 and FE. Now both have the same magnitude however the different sign has caused very different representation in two’s complement form. Now if the intent is to use FFFE or decimal 65534 then the same data would be placed in the word as in case of -2. In fact if -2 and 65534 are compared the processor will set the zero flag signaling that they are exactly equal. As regards an unsigned comparison the number 65534 is much greater than 2.

So if a JA is taken after comparing -2 in the destination with 2 in the source the jump will be taken. If however JG is used after the same comparison the jump will not be taken as it will consider the sign and with the sign -2 is smaller than 2. The key idea is that -2 and 65534 were both stored in memory in the same form. It was the interpretation that treated it as a signed or as an unsigned number.

The unsigned comparisons see the numbers as 0 being the smallest and 65535 being the largest with the order that 0 < 1 < 2 … < 65535. The signed comparisons see the number -32768 which has the same memory representation as 32768 as the smallest number and 32767 as the largest with the order -32768 < -32767 < … < -1 < 0 < 1 < 2 < … < 32767. All the negative numbers have the same representation as an unsigned number in the range 32768 … 65535 however the signed interpretation of the signed comparisons makes them be treated as negative numbers smaller than zero.

All meaningful situations both for signed and unsigned numbers than occur after a comparison are detailed in the following table.

DEST = SRC	ZF = 1	When the source is subtracted from the destination and both are equal the result is zero and therefore the zero flag is set. This works for both signed and unsigned numbers.
UDEST < USRC	CF = 1	When an unsigned source is subtracted from an unsigned destination and the destination is smaller, a borrow is needed which sets the carry flag.
UDEST ≤ USRC	ZF = 1 OR CF = 1	If the zero flag is set, it means that the source and destination are equal and if the carry flag is set it means a borrow was needed in the subtraction and therefore the destination is smaller.
UDEST ≥ USRC	CF = 0	When an unsigned source is subtracted from an unsigned destination no borrow will be needed either when the operands are equal or when the destination is greater than the source.
UDEST > USRC	ZF = 0 AND CF = 0	The unsigned source and destination are not equal if the zero flag is not set and the destination is not smaller since no borrow was taken. Therefore the destination is greater than the source.
SDEST < SSRC	SF ≠ OF	When a signed source is subtracted from a signed destination and the answer is negative with no overflow than the destination is smaller than the source. If however there is an overflow meaning that the sign has changed unexpectedly, the meanings are reversed and a

		positive number signals that the destination is smaller.
SDEST ≤ SSRC	ZF = 1 OR SF ≠ OF	If the zero flag is set, it means that the source and destination are equal and if the sign and overflow flags differ it means that the destination is smaller as described above.
SDEST ≥ SSRC	SF = OF	When a signed source is subtracted from a signed destination and the answer is positive with no overflow than the destination is greater than the source. When an overflow is there signaling that sign has changed unexpectedly, we interpret a negative answer as the signal that the destination is greater.
SDEST > SSRC	ZF = 0 AND SF = OF	If the zero flag is not set, it means that the signed operands are not equal and if the sign and overflow match in addition to this it means that the destination is greater than the source.

3.2. CONDITIONAL JUMPS

For every interesting or meaningful situation of flags, a conditional jump is there. For example JZ and JNZ check the zero flag. If in a comparison both operands are same, the result of subtraction will be zero and the zero flag will be set. Thus JZ and JNZ can be used to test equality. That is why there are renamed versions JE and JNE read as jump if equal or jump if not equal. They seem more logical in writing but mean exactly the same thing with the same opcode. Many jumps are renamed with two or three names for the same jump, so that the appropriate logic can be conveyed in assembly language programs. This renaming is done by Intel and is a standard for iAPX88. JC and JNC test the carry flag. For example we may need to test whether there was an overflow in the last unsigned addition or subtraction. Carry flag will also be set if two unsigned numbers are subtracted and the first is smaller than the second. Therefore the renamed versions JB, JNAE, and JNB, JAE are there standing for jump if below, jump if not above or equal, jump if not below, and jump if above or equal respectively. The operation of all jumps can be seen from the following table.

JC JB JNAE	Jump if carry Jump if below Jump if not above or equal	CF = 1	This jump is taken if the last arithmetic operation generated a carry or required a borrow. After a CMP it is taken if the unsigned source is smaller than the unsigned destination.
JNC JNB JAE	Jump if not carry Jump if not below Jump if above or equal	CF = 0	This jump is taken if the last arithmetic operation did not

			generated a carry or required a borrow. After a CMP it is taken if the unsigned source is larger or equal to the unsigned destination.
JE JZ	Jump if equal Jump if zero	ZF = 1	This jump is taken if the last arithmetic operation produced a zero in its destination. After a CMP it is taken if both operands were equal.
JNE JNZ	Jump if not equal Jump if not zero	ZF = 0	This jump is taken if the last arithmetic operation did not produced a zero in its destination. After a CMP it is taken if both operands were different.
JA JNBE	Jump if above Jump if not below or equal	ZF = 0 AND CF = 0	This jump is taken after a CMP if the unsigned source is larger than the unsigned destination.
JNA JBE	Jump if not above Jump if not below or equal	ZF = 1 OR CF = 1	This jump is taken after a CMP if the unsigned source is smaller than or equal to the unsigned destination.
JL JNGE	Jump if less Jump if not greater or equal	SF ≠ OF	This jump is taken after a CMP if the signed source is smaller than the signed destination.
JNL JGE	Jump if not less Jump if greater or equal	SF = OF	This jump is taken after a CMP if the signed source is larger than or equal to the signed destination.
JG JNLE	Jump if greater Jump if not less or equal	ZF = 0 AND SF = OF	This jump is taken after a CMP if the signed source is larger than the signed destination.
JNG JLE	Jump if not greater Jump if less or equal	ZF = 1 OR SF ≠ OF	This jump is taken after a CMP if the signed source is smaller than or equal to the signed destination.

JO	Jump if overflow.	OF = 1	This jump is taken if the last arithmetic operation changed the sign unexpectedly.
JNO	Jump if not overflow	OF = 0	This jump is taken if the last arithmetic operation did not change the sign unexpectedly.
JS	Jump if sign	SF = 1	This jump is taken if the last arithmetic operation produced a negative number in its destination.
JNS	Jump if not sign	SF = 0	This jump is taken if the last arithmetic operation produced a positive number in its destination.
JP JPE	Jump if parity Jump if even parity	PF = 1	This jump is taken if the last arithmetic operation produced a number in its destination that has even parity.
JNP JPO	Jump if not parity Jump if odd parity	PF = 0	This jump is taken if the last arithmetic operation produced a number in its destination that has odd parity.
JCXZ	Jump if CX is zero	CX = 0	This jump is taken if the CX register is zero.

The CMP instruction sets the flags reflecting the relation of the destination to the source. This is important as when we say jump if above, then what is above what. The destination is above the source or the source is above the destination.

The JA and JB instructions are related to unsigned numbers. That is our interpretation for the destination and source operands is unsigned. The 16th bit holds data and not the sign. In the JL and JG instructions standing for jump if lower and jump if greater respectively, the interpretation is signed. The 16th bit holds the sign and not the data. The difference between them will be made clear as an elaborate example will be given to explain the difference.

One jump is special that it is not dependant on any flag. It is JCXZ, jump if the CS register is zero. This is because of the special treatment of the CX register as a counter. This jump is regardless of the zero flag. There is no counterpart or not form of this instruction.

The adding numbers example of the last chapter can be a little simplified using the compare instruction on the BX register and eliminating the need for a separate counter as below.

Example 3.1

```
001  ; a program to add ten numbers without a separate counter
002  [org 0x0100]
003                mov  bx, 0              ; initialize array index to zero
004                mov  ax, 0              ; initialize sum to zero
005
006  l1:           add  ax, [num1+bx]      ; add number to ax
007                add  bx, 2              ; advance bx to next index
008                cmp  bx, 20             ; are we beyond the last index
009                jne  l1                 ; if not add next number
010
011                mov  [total], ax        ; write back sum in memory
012
013                mov  ax, 0x4c00         ; terminate program
014                int  0x21
015
016  num1:         dw   10, 20, 30, 40, 50, 10, 20, 30, 40, 50
017  total:        dw   0
```

006 The format of memory access is still base + offset.

008 BX is used as the array index as well as the counter. The offset of 11th number will be 20, so as soon as BX becomes 20 just after the 10th number has been added, the addition is stopped.

009 The jump is displayed as JNZ in the debugger even though we have written JNE in our example. This is because it is a renamed jump with the same opcode as JNZ and the debugger has no way of knowing the mnemonic that we used after looking just at the opcode. Also every code and data reference that we used till now is seen in the opcode as well. However for the jump instruction we see an operand of F2 in the opcode and not 0116. This will be discussed in detail with unconditional jumps. It is actually a short relative jump and the operand is stored in the form of positive or negative offset from this instruction.

With conditional branching in hand, there are just a few small things left in assembly language that fills some gaps. Now there is just imagination and the skill to conceive programs that can make you write any program.

3.3. UNCONDITIONAL JUMP

Till now we have been placing data at the end of code. There is no such restriction and we can define data anywhere in the code. Taking the previous example, if we place data at the start of code instead of at the end and we load our program in the debugger. We can see our data placed at the start but the debugger is intending to start execution at our data. The COM file definition said that the first executable instruction is at offset 0100 but we have placed data there instead of code. So the debugger will try to interpret that data as code and showed whatever it could make up out of those opcodes.

We introduce a new instruction called JMP. It is the unconditional jump that executes regardless of the state of all flags. So we write an unconditional jump as the very first instruction of our program and jump to the next instruction that follows our data declarations. This time 0100 contains a valid first instruction of our program.

Example 3.2

```
001  ; a program to add ten numbers without a separate counter
002  [org 0x0100]
003                jmp start               ; unconditionally jump over data
004
005  num1:         dw   10, 20, 30, 40, 50, 10, 20, 30, 40, 50
006  total:        dw   0
```

```
007
008   start:        mov  bx, 0             ; initialize array index to zero
009                 mov  ax, 0             ; initialize sum to zero
010
011   l1:           add  ax, [num1+bx]     ; add number to ax
012                 add  bx, 2             ; advance bx to next index
013                 cmp  bx, 20            ; are we beyond the last index
014                 jne  l1                ; if not add next number
015
016                 mov  [total], ax       ; write back sum in memory
017
018                 mov  ax, 0x4c00        ; terminate program
019                 int  0x21
```

003 JMP jumps over the data declarations to the start label and execution resumes from there.

3.4. RELATIVE ADDRESSING

Inside the debugger the instruction is shown as JMP 0119 and the location 0119 contains the original first instruction of the logic of our program. This jump is unconditional, it will always be taken. Now looking at the opcode we see F21600 where F2 is the opcode and 1600 is the operand to it. 1600 is 0016 in proper word order. 0119 is not given as a parameter rather 0016 is given.

This is position relative addressing in contrast to absolute addressing. It is not telling the exact address rather it is telling how much forward or backward to go from the current position of IP in the current code segment. So the instruction means to add 0016 to the IP register. At the time of execution of the first instruction at 0100 IP was pointing to the next instruction at 0103, so after adding 16 it became 0119, the desired target location. The mechanism is important to know, however all calculations in this mechanism are done by the assembler and by the processor. We just use a label with the JMP instruction and are ensured that the instruction at the target label will be the one to be executed.

3.5. 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.

Short Jump

EB	Disp

Near Jump

EB	Disp Low	Disp High

Far Jump

EB	IP Low	IP High	CS Low	CS High

Near Jump

When the relative address stored with the instruction is in 16 bits as in the last example 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 singed 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 wit 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.

3.6. SORTING EXAMPLE

Moving ahead from our example of adding numbers we progress to a program that can sort a list of numbers using the tools that we have accumulated till now. Sorting can be ascending or descending like if the largest number comes at the top, followed by a smaller number and so on till the smallest number the sort will be called descending. The other order starting with the smallest number and ending at the largest is called ascending sort. This is a common problem and many algorithms have been developed to solve it. One simple algorithm is the bubble sort algorithm.

In this algorithm we compare consecutive numbers. If they are in required order e.g. if it is a descending sort and the first is larger then the second, then we leave them as it is and if they are not in order, we swap them. Then we do the same process for the next two numbers and so on till the last two are compared and possibly swapped.

A complete iteration is called a pass over the array. We need N passes at least in the simplest algorithm if N is the number of elements to be sorted. A finer algorithm is to check if any swap was done in this pass and stop as soon as a pass goes without a swap. The array is now sorted as every pair of elements is in order.

For example if our list of numbers is 60, 55, 45, and 58 and we want to sort them in ascending order, the first comparison will be of 60 and 55 and as the order will be reversed to 55 and 60. The next comparison will be of 60 and 45 and again the two will be swapped. The next comparison of 60 and 58 will also cause a swap. At the end of first pass the numbers will be in order of 55, 45, 58, and 60. Observe that the largest number has bubbled down to the bottom. Just like a bubble at bottom of water. In the next pass 55 and 45 will be swapped. 55 and 58 will not be swapped and 58 and 60 will also not be swapped. In the next pass there will be no swap as the elements are in order i.e. 45, 55, 58, and 60. The passes will be stopped as the last pass did not cause any swap. The application of bubble sort on these numbers is further explained with the following illustration.

State of Data				Swap Done	Swap Flag
Pass 1					Off
60	55	45	58	Yes	On
55	60	45	58	Yes	On
55	45	60	58	Yes	On
Pass 2					Off
55	45	58	60	Yes	On
45	55	58	60	No	On
45	55	58	60	No	On
Pass 3					Off
45	55	58	60	No	Off
45	55	58	60	No	Off
45	55	58	60	No	Off

No more passes since swap flag is Off

Example 3.3

```
; 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
```

003	The jump instruction is placed to skip over data.
006	The swap flag can be stored in a register but as an example it is stored in memory and also to extend the concept at a later stage.
011-012	One element is read in AX and it is compared with the next element because memory to memory comparisons are not allowed.
013	If the JBE is changed to JB, not only the unnecessary swap on equal will be performed, there will be a major algorithmic flaw due to a logical error as in the case of equal elements the algorithm will never stop. JBE won't swap in the case of equal elements.
015-017	The swap is done using DX and AX registers in such a way that the values are crossed. The code uses the information that one of the elements is already in the AX register.
021	This time BX is compared with 18 instead of 20 even though the number of elements is same. This is because we pick an element and compare it with the next element. When we pick the 9th element we compare it with the next element and this is the last comparison, since if we pick the 10th element we will compare it with the 11th element and there is no 11th element in our case.
024-025	If a swap is done we repeat the whole process for possible more swaps.

Inside the debugger we observe that the JBE is changed to JNA due to the same reason as discussed for JNE and JNZ. The passes change the data in the same manner as we presented in our illustration above. If JBE in the code is changed to JAE the sort will change from ascending to descending. For signed numbers we can use JLE and JGE respectively for ascending and descending sort.

To clarify the difference of signed and unsigned jumps we change the data array in the last program to include some negative numbers as well. When JBE will be used on this data, i.e. with unsigned interpretation of the data and an ascending sort, the negative numbers will come at the end after the largest positive number. However JLE will bring the negative numbers at the very start of the list to bring them in proper ascending order according to a signed interpretation, even though they are large in magnitude. The data used is shown as below.

```
data:       dw 60, 55, 45, 50, -40, -35, 25, 30, 10, 0
```

This data includes some signed numbers as well. The JBE instruction will treat this data as an unsigned number and will cater only for the magnitude ignoring the sign. If the program is loaded in the debugger, the numbers will appear in their hexadecimal equivalent. The two numbers -40 and -35 are especially important as they are represented as FFD8 and FFDD. This data is not telling whether it is signed or unsigned. Our interpretation will decide whether it is a very large unsigned number or a signed number in two's complement form.

If the sorting algorithm is applied on the above data with JBE as the comparison instruction to sort in ascending order with unsigned interpretation, observe the comparisons of the two numbers FFD8 and FFDD. For example it will decide that FFDD > FFD8 since the first is larger in magnitude. At the end of sorting FFDD will be at the end of the list being declared the largest number and FFD8 will precede it to be the second largest.

If however the comparison instruction is changed to JLE and sorting is done on the same data it works similarly except on the two numbers FFDD and FFD8. This time JLE declares them to be smaller than every other number and also declares FFDD < FFD8. At the end of sorting, FFDD is declared to be the smallest number followed by FFD8 and then 0000. This is in contrast to the last example where JBE was used. This happened because JLE interpreted our data as signed numbers, and as a signed number FFDD has its sign bit on signaling that it is a negative number in two's complement form which is smaller than 0000 and every positive number. However JBE did not give any significance to the sign bit and included it in the magnitude. Therefore it declared the negative numbers to be the largest numbers.

If the required interpretation was of signed numbers the result produced by JLE is correct and if the required interpretation was of unsigned numbers the result produced by JBE is correct. This is the very difference between signed and unsigned integers in higher level languages, where the compiler takes the responsibility of making the appropriate jump depending on the type of integer used. But it is only at this level that we can understand the actual mechanism going on. In assembly language, use of proper jump is the responsibility of the programmer, to convey the intentions to use the data as signed or as unsigned.

The remaining possibilities of signed descending sort and unsigned descending sort can be done on the same lines and are left as an exercise. Other conditional jumps work in the same manner and can be studied from the reference at the end. Several will be discussed in more detail when they are used in subsequent chapters.

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
```

4
Bit Manipulations

4.1. MULTIPLICATION ALGORITHM

With the important capability of decision making in our repertoire we move on to the discussion of an algorithm, which will help us uncover an important set of instructions in our processor used for bit manipulations.

Multiplication is a common process that we use, and we were trained to do in early schooling. Remember multiplying by a digit and then putting a cross and then multiplying with the next digit and putting two crosses and so on and summing the intermediate results in the end. Very familiar process but we never saw the process as an algorithm, and we need to see it as an algorithm to convey it to the processor.

To highlight the important thing in the algorithm we revise it on two 4bit binary numbers. The numbers are 1101 i.e. 13 and 0101 i.e. 5. The answer should be 65 or in binary 01000001. Observe that the answer is twice as long as the multiplier and the multiplicand. The multiplication is shown in the following figure.

```
    1101 = 13
    0101 = 5
    -----
    1101
   0000x
  1101xx
 0000xxx
--------
01000001 = 65
```

We take the first digit of the multiplier and multiply it with the multiplicand. As the digit is one the answer is the multiplicand itself. So we place the multiplicand below the bar. Before multiplying with the next digit a cross is placed at the right most place on the next line and the result is placed shifted one digit left. However since the digit is zero, the result is zero. Next digit is one, multiplying with which, the answer is 1101. We put two crosses on the next line at the right most positions and place the result there shifted two places to the left. The fourth digit is zero, so the answer 0000 is placed with three crosses to its right.

Observe the beauty of binary base, as no real multiplication is needed at the digit level. If the digit is 0 the answer is 0 and if the digit is 1 the answer is the multiplicand itself. Also observe that for every next digit in the multiplier the answer is written shifted one more place to the left. No shifting for the first digit, once for the second, twice for the third and thrice for the fourth one. Adding all the intermediate answers the result is 01000001=65 as desired. Crosses are treated as zero in this addition.

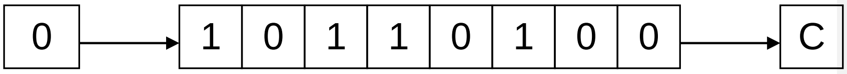
Before formulating the algorithm for this problem, we need some more instructions that can shift a number so that we use this instruction for our multiplicand shifting and also some way to check the bits of the multiplier one by one.

4.2. SHIFTING AND ROTATIONS

The set of shifting and rotation instructions is one of the most useful set in any processor’s instruction set. They simplify really complex tasks to a very neat and concise algorithm. The following shifting and rotation operations are available in our processor.

Shift Logical Right (SHR)

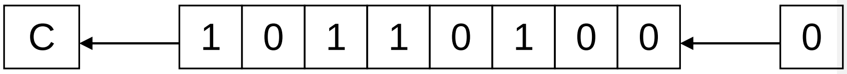
The shift logical right operation inserts a zero from the left and moves every bit one position to the right and copies the rightmost bit in the carry flag. Imagine that there is a pipe filled to capacity with eight balls. The pipe is open from both ends and there is a basket at the right end to hold anything dropping from there. The operation of shift logical right is to force a white ball from the left end. The operation is depicted in the following illustration.



White balls represent zero bits while black balls represent one bits. Sixteen bit shifting is done the same way with a pipe of double capacity.

Shift Logical Left (SHL) / Shift Arithmetic Left (SAL)

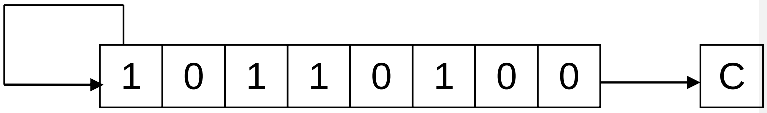
The shift logical left operation is the exact opposite of shift logical right. In this operation the zero bit is inserted from the right and every bit moves one position to its left with the most significant bit dropping into the carry flag. Shift arithmetic left is just another name for shift logical left. The operation is again exemplified with the following illustration of ball and pipes.



Shift Arithmetic Right (SAR)

A signed number holds the sign in its most significant bit. If this bit was one a logical right shifting will change the sign of this number because of insertion of a zero from the left. The sign of a signed number should not change because of shifting.

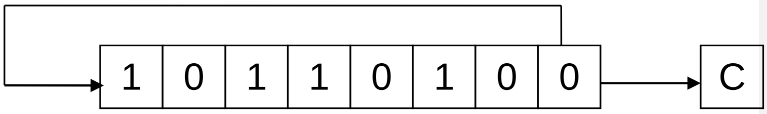
The operation of shift arithmetic right is therefore to shift every bit one place to the right with a copy of the most significant bit left at the most significant place. The bit dropped from the right is caught in the carry basket. The sign bit is retained in this operation. The operation is further illustrated below.



The left shifting operation is basically multiplication by 2 while the right shifting operation is division by two. However for signed numbers division by two can be accomplished by using shift arithmetic right and not shift logical right. The left shift operation is equivalent to multiplication except when an important bit is dropped from the left. The overflow flag will signal this condition if it occurs and can be checked with JO. For division by 2 of a signed number logical right shifting will give a wrong answer for a negative number as the zero inserted from the left will change its sign. To retain the sign flag and still effectively divide by two the shift arithmetic right instruction must be used on signed numbers.

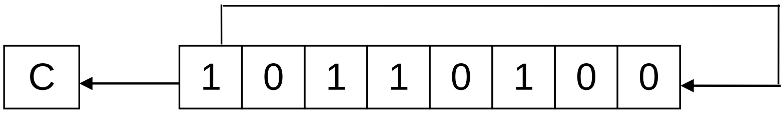
Rotate Right (ROR)

In the rotate right operation every bit moves one position to the right and the bit dropped from the right is inserted at the left. This bit is also copied into the carry flag. The operation can be understood by imagining that the pipe used for shifting has been molded such that both ends coincide. Now when the first ball is forced to move forward, every ball moves one step forward with the last ball entering the pipe from its other end occupying the first ball's old position. The carry basket takes a snapshot of this ball leaving one end of the pipe and entering from the other.



Rotate Left (ROL)

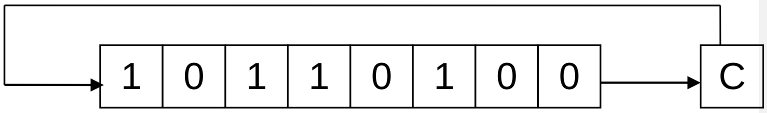
In the operation of rotate left instruction, the most significant bit is copied to the carry flag and is inserted from the right, causing every bit to move one position to the left. It is the reverse of the rotate right instruction. Rotation can be of eight or sixteen bits. The following illustration will make the concept clear using the same pipe and balls example.



Rotate Through Carry Right (RCR)

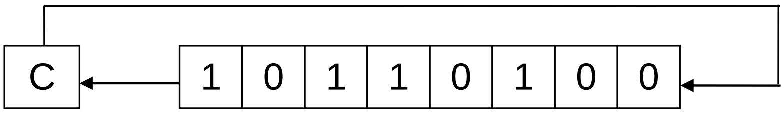
In the rotate through carry right instruction, the carry flag is inserted from the left, every bit moves one position to the right, and the right most bit is dropped in the carry flag. Effectively this is a nine bit or a seventeen bit rotation instead of the eight or sixteen bit rotation as in the case of simple rotations.

Imagine the circular molded pipe as used in the simple rotations but this time the carry position is part of the circle between the two ends of the pipe. Pushing the carry ball from the left causes every ball to move one step to its right and the right most bit occupying the carry place. The idea is further illustrated below.



Rotate Through Carry Left (RCL)

The exact opposite of rotate through carry right instruction is the rotate through carry left instruction. In its operation the carry flag is inserted from the right causing every bit to move one location to its left and the most significant bit occupying the carry flag. The concept is illustrated below in the same manner as in the last example.



4.3. MULTIPLICATION IN ASSEMBLY LANGUAGE

In the multiplication algorithm discussed above we revised the way we multiplied number in lower classes, and gave an example of that method on binary numbers. We make a simple modification to the traditional algorithm before we proceed to formulate it in assembly language.

In the traditional algorithm we calculate all intermediate answers and then sum them to get the final answer. If we add every intermediate answer to accumulate the result, the result will be same in the end, except that we do not have to remember a lot of intermediate answers during the whole multiplication. The multiplication with the new algorithm is shown below.

```
      1101 = 13       Accumulated Result
      0101 = 5
      -----                     0 (Initial Value)
      1101 = 13       0 + 13  = 13
     0000x = 0        13 + 0  = 13
    1101xx = 52       13 + 52 = 65
   0000xxx = 0        65 + 0  = 65 (Answer)
```

We try to identify steps of our algorithm. First we set the result to zero. Then we check the right most bit of multiplier. If it is one add the multiplicand to the result, and if it is zero perform no addition. Left shift the multiplicand before the next bit of multiplier is tested. The left shifting of the multiplicand is performed regardless of the value of the multiplier’s right most bit. Just like the crosses in traditional multiplication are always placed to mark the ones, tens, thousands, etc. places. Then check the next bit and if it is one add the shifted value of the multiplicand to the result. Repeat for as many digits as there are in the multiplier, 4 in our example. Formulating the steps of the algorithm we get:

- Shift the multiplier to the right.
- If CF=1 add the multiplicand to the result.
- Shift the multiplicand to the right.
- Repeat the algorithm 4 times.

For an 8bit multiplication the algorithm will be repeated 8 times and for a sixteen bit multiplication it will be repeated 16 times, whatever the size of the multiplier is.

The algorithm uses the fact that shifting right forces the right most bit to drop in the carry flag. If we test the carry flag using JC we are effectively testing the right most bit of the multiplier. Another shifting will cause the next bit to drop in the next iteration and so on. So our task of checking bits one by one is satisfied using the shift operation. There are many other methods to do this bit testing as well, however we exemplify one of the methods in this example.

In the first iteration there is no shifting just like there is no cross in traditional multiplication in the first pass. Therefore we placed the left shifting of the multiplicand after the addition step. However the right shifting of multiplier must be before the addition as the addition step’s execution depends upon its result.

We introduce an assembly language program to perform this 4bit multiplication. The algorithm is extensible to more bits but there are a few complications, which are left to be discussed later. For now we do a 4bit multiplication to keep the algorithm simple.

Example 4.1

```
; 4bit multiplication algorithm
[org 0x100]
              jmp  start

multiplicand: db   13                 ; 4bit multiplicand (8bit space)
multiplier:   db   5                  ; 4bit multiplier
```

```
07   result:       db   0                   ; 8bit result
08
09   start:        mov  cl, 4               ; initialize bit count to four
10                 mov  bl, [multiplicand] ; load multiplicand in bl
11                 mov  dl, [multiplier]   ; load multiplier in dl
12
13   checkbit:     shr  dl, 1               ; move right most bit in carry
14                 jnc  skip                ; skip addition if bit is zero
15
16                 add  [result], bl        ; accumulate result
17
18   skip:         shl  bl, 1               ; shift multiplicand left
19                 dec  cl                  ; decrement bit count
20                 jnz  checkbit            ; repeat if bits left
21
22                 mov  ax, 0x4c00          ; terminate program
23                 int  0x21
```

04-06 The numbers to be multiplied are constants for now.

07 The multiplication is four bit so the answer is stored in an 8bit register. If the operands were 8bit the answer would be 16bit and if the operands were 16bit the answer would be 32bit. Since eight bits can fit in a byte we have used 4bit multiplication as our first example.

14-16 Since addition by zero means nothing we skip the addition step if the rightmost bit of the multiplier is zero. If the jump is not taken the shifted value of the multiplicand is added to the result.

18 The multiplicand is left shifted in every iteration regardless of the multiplier bit.

19 DEC is a new instruction but its operation should be immediately understandable with the knowledge gained till now. It simply subtracts one from its single operand.

20 The JNZ instruction causes the algorithm to repeat till any bits of the multiplier are left

Inside the debugger observe the working of the SHR and SHL instructions. The SHR instruction is effectively dividing its operand by two and the remainder is stored in the carry flag from where we test it. The SHL instruction is multiplying its operand by two so that it is added at one place more towards the left in the result.

4.4. EXTENDED OPERATIONS

We performed a 4bit multiplication to explain the algorithm however the real advantage of the computer is when we ask it to multiply large numbers. Numbers whose multiplication takes real time. If we have an 8bit number we can do the multiplication in word registers, but are we limited to word operations? What if we want to multiply 32bit or even larger numbers? We are certainly not limited. Assembly language only provides us the basic building blocks. We build a plaza out of these blocks, or a building, or a classic piece of architecture is only dependant upon our imagination. With our logic we can extend these algorithms as much as we want.

Our next example will be multiplication of 16bit numbers to produce a 32bit answer. However for a 32bit answer we need a way to shift a 32bit number and a way to add 32bit numbers. We cannot depend on 16bit shifting as we have 16 significant bits in our multiplicand and shifting any bit towards the left may drop a valuable bit causing a totally wrong result. A valuable bit means any bit that is one. Dropping a zero bit doesn't cause any difference. So we place the 16it number in 32bit space with the upper 16 bits zeroed so that the sixteen shift operations don't cause any valuable bit to drop. Even though the numbers were 16bit we need 32bit operations to multiply correctly.

To clarify this necessity, we take example of a number 40000 or 9C40 in hexadecimal. In binary it is represented as 1001110001000000. To multiply by two we shift it one place to the left. The answer we get is 0011100010000000 and the left most one is dropped in the carry flag. The answer should be the 17bit number 0x13880 but it is 0x3880, which is 14464 in decimal instead of the expected 80000. We should be careful of this situation whenever shifting is used.

Extended Shifting

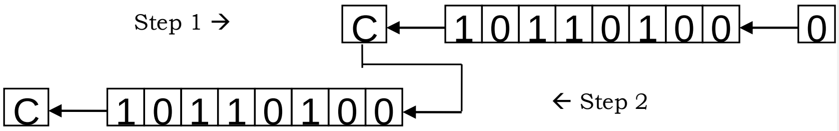
Using our basic shifting and rotation instructions we can effectively shift a 32bit number in memory word by word. We cannot shift the whole number at once since our architecture is limited to word operations. The algorithm we use consists of just two instructions and we name it extended shifting.

```
num1:           dd   40000
                shl  word [num1], 1
                rcl  word [num1+2], 1
```

The DD directive reserves a 32bit space in memory, however the value we placed there will fit in 16bits. So we can safely shift the number left 16 times. The least significant word is accessible at num1 and the most significant word is accessible at num1+2.

The two instructions are carefully crafted such that the first one shifts the lower word towards the left and the most significant bit of that word is dropped in carry. With the next instruction we push that dropped bit into the least significant bit of the next word effectively joining the two 16bit words. The final carry after the second instruction will be the most significant bit of the higher word, which for this number will always be zero.

The following illustration will clarify the concept. The pipe on the right contains the lower half and the pipe on the left contains the upper half. The first instruction forced a zero from the right into the lower half and the left most bit is saved in carry, and from there it is pushed into the upper half and the upper half is shifted as well.



For shifting right the exact opposite is done however care must be taken to shift right the upper half first and then rotate through carry right the lower half for obvious reasons. The instructions to do this are.

```
num1:           dd   40000
                shr  word [num1+2], 1
                rcr  word [num1], 1
```

The same logic has worked. The shift placed the least significant bit of the upper half in the carry flag and it was pushed from right into the lower half. For a singed shift we would have used the shift arithmetic right instruction instead of the shift logical right instruction.

The extension we have done is not limited to 32bits. We can shift a number of any size say 1024 bits. The second instruction will be repeated a number of times and we can achieve the desired effect. Using two simple instructions we have increased the capability of the operation to effectively an unlimited number of bits. The actual limit is the available memory as even the segment limit can be catered with a little thought.

Extended Addition and Subtraction

We also needed 32bit addition for multiplication of 16bit numbers. The idea of extension is same here. However we need to introduce a new instruction at this place. The instruction is ADC or "add with carry." Normal addition has two operands and the second operand is added to the first operand. However ADC has three operands. The third implied operand is the carry flag. The ADC instruction is specifically placed for extending the capability of ADD. Numbers of any size can be added using a proper combination of ADD and ADC. All basic building blocks are provided for the assembly language programmer, and the programmer can extend its capabilities as much as needed by using these fine instructions in appropriate combinations.

Further clarifying the operation of ADC, consider an instruction “ADC AX, BX.” Normal addition would have just added BX to AX, however ADC first adds the carry flag to AX and then adds BX to AX. Therefore the last carry is also included in the result.

The algorithm should be apparent by now. The lower halves of the two numbers to be added are firsted added with a normal addition. For the upper halves a normal addition would lose track of a possible carry from the lower halves and the answer would be wrong. If a carry was generated it should go to the upper half. Therefore the upper halves are added with an addition with carry instruction.

Since one operand must be in register, ax is used to read the lower and upper halves of the source one by one. The destination is directly updated. The set of instructions goes here.

```
dest:          dd   40000
src:           dd   80000

               mov  ax, [src]
               add  word [dest], ax
               mov  ax, [src+2]
               adc  word [dest+2], ax
```

To further extend it more addition with carries will be used. However the carry from last addition will be wasted as there will always be a size limit where the results and the numbers are stored. This carry will remain in the carry flag to be tested for a possible overflow.

For subtraction the same logic will be used and just like addition with carry there is an instruction to subtract with borrow called SBB. Borrow in the name means the carry flag and is used just for clarity. Or we can say that the carry flag holds the carry for addition instructions and the borrow for subtraction instructions. Also the carry is generated at the 17th bit and the borrow is also taken from the 17th bit. Also there is no single instruction that needs borrow and carry in their independent meanings at the same time. Therefore it is logical to use the same flag for both tasks.

We extend subtraction with a very similar algorithm. The lower halves must be subtracted normally while the upper halves must be subtracted with a subtract with borrow instruction so that if the lower halves needed a borrow, a one is subtracted from the upper halves. The algorithm is as under.

```
dest:          dd   40000
src:           dd   80000

               mov  ax, [src]
               sub  word [dest], ax
               mov  ax, [src+2]
               sbb  word [dest+2], ax
```

Extended Multiplication

We use extended shifting and extended addition to formulate our algorithm to do extended multiplication. The multiplier is still stored in 16bits since we only need to check its bits one by one. The multiplicand however cannot be stored in 16bits otherwise on left shifting its significant bits might get lost. Therefore it has to be stored in 32bits and the shifting and addition used to accumulate the result must be 32bits as well.

Example 4.2

```
01  ; 16bit multiplication
02  [org 0x0100]
03                jmp  start
04
05  multiplicand: dd   1300               ; 16bit multiplicand 32bit space
06  multiplier:   dw   500                ; 16bit multiplier
07  result:       dd   0                  ; 32bit result
08
09  start:        mov  cl, 16             ; initialize bit count to 16
10                mov  dx, [multiplier]   ; load multiplier in dx
11
12  checkbit:     shr  dx, 1              ; move right most bit in carry
13                jnc  skip               ; skip addition if bit is zero
14
15                mov  ax, [multiplicand]
16                add  [result], ax       ; add less significant word
17                mov  ax, [multiplicand+2]
18                adc  [result+2], ax     ; add more significant word
19
20  skip:         shl  word [multiplicand], 1
21                rcl  word [multiplicand+2], 1 ; shift multiplicand left
22                dec  cl                 ; decrement bit count
23                jnz  checkbit           ; repeat if bits left
24
25                mov  ax, 0x4c00         ; terminate program
26                int  0x21
```

05-07	The multiplicand and the multiplier are stored in 32bit space while the multiplier is stored as a word.
10	The multiplier is loaded in DX where it will be shifted bit by bit. It can be directly shifted in memory as well.
15-18	The multiplicand is added to the result using extended 32bit addition.
20-21	The multiplicand is shifted left as a 32bit number using extended shifting operation.

The multiplicand will occupy the space from 0103-0106, the multiplier will occupy space from 0107-0108 and the result will occupy the space from 0109-010C. Inside the debugger observe the changes in these memory locations during the course of the algorithm. The extended shifting and addition operations provide the same effect as would be provided if there were 32bit addition and shifting operations available in the instruction set.

At the end of the algorithm the result memory locations contain the value 0009EB10 which is 65000 in decimal; the desired answer. Also observe that the number 00000514 which is 1300 in decimal, our multiplicand, has become 05140000 after being left shifted 16 times. Our extended shifting has given the same result as if a 32bit number is left shifted 16 times as a unit.

There are many other important applications of the shifting and rotation operations in addition to this example of the multiplication algorithm. More examples will come in coming chapters.

4.5. BITWISE LOGICAL OPERATIONS

The 8088 processor provides us with a few logical operations that operate at the bit level. The logical operations are the same as discussed in computer logic design; however our perspective will be a little different. The four basic operations are AND, OR, XOR, and NOT.

The important thing about these operations is that they are bitwise. This means that if “and ax, bx” instruction is given, then the operation of AND is applied on corresponding bits of AX and BX. There are 16 AND operations as a result; one for every bit of AX. Bit 0 of AX will be set if both its original value and Bit 0 of BX are set, bit 1 will be set if both its original value and Bit 1 of BX are set, and so on for the remaining bits. These operations are conducted in parallel on the sixteen bits. Similarly the operations of other logical operations are bitwise as well.

AND operation

X	Y	X and Y
0	0	0
0	1	0
1	0	0
1	1	1

AND performs the logical bitwise *and* of the two operands (byte or word) and returns the result to the destination operand. A bit in the result is set if both corresponding bits of the original operands are set; otherwise the bit is cleared as shown in the truth table. Examples are "and ax, bx" and "and byte [mem], 5." All possibilities that are legal for addition are also legal for the AND operation. The different thing is the bitwise behavior of this operation.

OR operation

X	Y	X or Y
0	0	0
0	1	1
1	0	1
1	1	1

OR performs the logical bitwise "inclusive or" of the two operands (byte or word) and returns the result to the destination operand. A bit in the result is set if either or both corresponding bits in the original operands are set otherwise the result bit is cleared as shown in the truth table. Examples are "or ax, bx" and "or byte [mem], 5."

XOR operation

X	Y	X xor Y
0	0	0
0	1	1
1	0	1
1	1	0

XOR (Exclusive Or) performs the logical bitwise "exclusive or" of the two operands and returns the result to the destination operand. A bit in the result is set if the corresponding bits of the original operands contain opposite values (one is set, the other is cleared) otherwise the result bit is cleared as shown in the truth table. XOR is a very important operation due to the property that it is a reversible operation. It is used in many cryptography algorithms, image processing, and in drawing operations. Examples are "xor ax, bx" and "xor byte [mem], 5."

NOT operation

NOT inverts the bits (forms the one's complement) of the byte or word operand. Unlike the other logical operations, this is a single operand instruction, and is not purely a logical operation in the sense the others are, but it is still traditionally counted in the same set. Examples are "not ax" and "not byte [mem], 5."

4.6. MASKING OPERATIONS

Selective Bit Clearing

Another use of AND is to make selective bits zero in its destination operand. The source operand is loaded with a mask containing one at positions which are retain their old value and zero at positions which are to be zeroed. The effect of applying this operation on the destination with mask in the source is to clear the desired bits. This operation is called masking. For example if the lower nibble is to be cleared then the operation can be applied with F0 in the source. The upper nibble will retain its old value and the lower nibble will be cleared.

Selective Bit Setting

The operation can be used as a masking operation to set selective bits. The bits in the mask are cleared at positions which are to retain their values, and are set at positions which are to be set. For example to set the lower nibble of the destination operand, the operation should be applied with a mask of 0F in the source. The upper nibble will retain its value and the lower nibble will be set as a result.

Selective Bit Inversion

XOR can also be used as a masking operation to invert selective bits. The bits in the mask are cleared at positions, which are to retain their values, and are set at positions, which are to be inverted. For example to invert the lower nibble of the destination operand, the operand should be applied with a mask of 0F in the source. The upper nibble will retain its value and the lower nibble will be set as a result. Compare this with NOT which inverts everything. XOR on the other hand allows inverting selective bits.

Selective Bit Testing

AND can be used to check whether particular bits of a number are set or not. Previously we used shifting and JC to test bits one by one. Now we introduce another way to test bits, which is more powerful in the sense that any bit can be tested anytime and not necessarily in order. AND can be applied on a destination with a 1-bit in the desired position and a source, which is to be checked. If the destination is zero as a result, which can be checked with a JZ instruction, the bit at the desired position in the source was clear.

However the AND operation destroys the destination mask, which might be needed later as well. Therefore Intel provided us with another instruction analogous to CMP, which is non-destructive subtraction. This is the TEST instruction and is a non-destructive AND operation. It doesn't change the destination and only sets the flags according to the AND operation. By checking the flags, we can see if the desired bit was set or cleared.

We change our multiplication algorithm to use selective bit testing instead of checking bits one by one using the shifting operations.

Example 4.3

```
01  ; 16bit multiplication using test for bit testing
02  [org 0x0100]
03                jmp  start
04
05  multiplicand: dd   1300              ; 16bit multiplicand 32bit space
06  multiplier:   dw   500               ; 16bit multiplier
07  result:       dd   0                 ; 32bit result
08
09  start:        mov  cl, 16            ; initialize bit count to 16
10                mov  bx, 1             ; initialize bit mask
11
12  checkbit:     test bx, [multiplier]  ; move right most bit in carry
13                jz   skip              ; skip addition if bit is zero
14
15                mov  ax, [multiplicand]
16                add  [result], ax      ; add less significant word
17                mov  ax, [multiplicand+2]
18                adc  [result+2], ax    ; add more significant word
19
20  skip:         shl  word [multiplicand], 1
21                rcl  word [multiplicand+2], 1 ; shift multiplicand left
22                shl  bx, 1             ; shift mask towards next bit
23                dec  cl                ; decrement bit count
24                jnz  checkbit          ; repeat if bits left
25
26                mov  ax, 0x4c00        ; terminate program
27                int  0x21
```

12 The test instruction is used for bit testing. BX holds the mask and in every next iteration it is shifting left, as our concerned bit is now the next bit.

22-24 We can do without counting in this example. We can stop as soon as our mask in BX becomes zero. These are the small tricks that assembly allows us to do and optimize our code as a result.

Inside the debugger observe that both the memory location and the mask in BX do not change as a result of TEST instruction. Also observe how our mask is shifting towards the left so that the next TEST instruction tests the next bit. In the end we get the same result of 0009EB10 as in the previous example.

EXERCISES

1. Write a program to swap every pair of bits in the AX register.
2. Give the value of the AX register and the carry flag after each of the following instructions.

```
stc
mov ax, <your rollnumber>
adc ah, <first character of your name>
cmc
xor ah, al
mov cl, 4
shr al, cl
rcr ah, cl
```

3. Write a program to swap the nibbles in each byte of the AX register.
4. Calculate the number of one bits in BX and complement an equal number of least significant bits in AX.
   HINT: Use the XOR instruction
5. Write a program to multiply two 32bit numbers and store the answer in a 64bit location.
6. Declare a 32byte buffer containing random data. Consider for this problem that the bits in these 32 bytes are numbered from 0 to 255. Declare another byte that contains the starting bit number. Write a program to copy the byte starting at this starting bit number in the AX register. Be careful that the starting bit number may not be a multiple of 8 and therefore the bits of the desired byte will be split into two bytes.
7. AX contains a number between 0-15. Write code to complement the corresponding bit in BX. For example if AX contains 6; complement the 6th bit of BX.
8. AX contains a non-zero number. Count the number of ones in it and store the result back in AX. Repeat the process on the result (AX) until AX contains one. Calculate in BX the number of iterations it took to make AX one. For example BX should contain 2 in the following case:
   AX = 1100 0101 1010 0011 (input – 8 ones)
   AX = 0000 0000 0000 1000 (after first iteration – 1 one)
   AX = 0000 0000 0000 0001 (after second iteration – 1 one) STOP

5 Subroutines

5.1. PROGRAM FLOW

Till now we have accumulated the very basic tools of assembly language programming. A very important weapon in our arsenal is the conditional jump instruction. During the course of last two chapters we used these tools to write two very useful algorithms of sorting and multiplication. The multiplication algorithm is useful even though there is a MUL instruction in the 8088 instruction set, which can multiply 8bit and 16bit operands. This is because of the extensibility of our algorithm, as it is not limited to 16bits and can do 32bit or 64bit multiplication with minor changes.

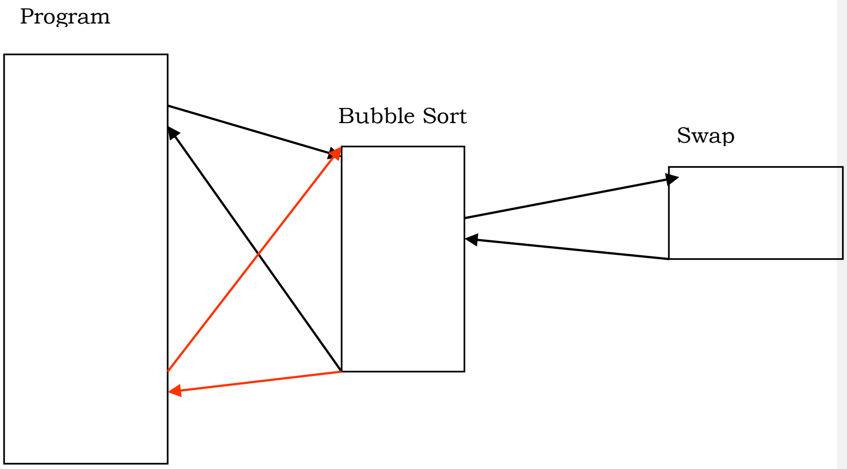
Both of these algorithms will be used a number of times in any program of a reasonable size and complexity. An application does not only need to multiply at a single point in code; it multiplies at a number of places. If multiplication or sorting is needed at 100 places in code, copying it 100 times is a totally infeasible solution. Maintaining such a code is an impossible task.

The straightforward solution to this problem using the concepts we have acquainted till now is to write the code at one place with a label, and whenever we need to sort we jump to this label. But there is problem with this logic, and the problem is that after sorting is complete how the processor will know where to go back. The immediate answer is to jump back to a label following the jump to bubble sort. But we have jumped to bubble sort from 100 places in code. Which of the 100 positions in code should we jump back? Jump back at the first invocation, but jump has a single fixed target. How will the second invocation work? The second jump to bubble sort will never have control back at the next line.

Instruction are tied to one another forming an execution thread, just like a knitted thread where pieces of cotton of different sizes are twisted together to form a thread. This thread of execution is our program. The jump instruction breaks this thread permanently, making a permanent diversion, like a turn on a highway. The conditional jump selects one of the two possible directions, like right or left turn on a road. So there is no concept of returning.

However there are roundabouts on roads as well that take us back from where we started after having traveled on the boundary of the round. This is the concept of a temporary diversion. Two or more permanent diversions can take us back from where we started, just like two or more road turns can take us back to the starting point, but they are still permanent diversions in their nature.

We need some way to implement the concept of temporary diversion in assembly language. We want to create a roundabout of bubble sort, another roundabout of our multiplication algorithm, so that we can enter into the roundabout whenever we need it and return back to wherever we left from after completing the round.



Key point in the above discussion is returning to where we left from, like a loop in a knitted thread. Diversion should be temporary and not permanent. The code of bubble sort written at one place, multiply at another, and we temporarily divert to that place, thus avoiding a repetition of code at a 100 places.

CALL and RET

In every processor, instructions are available to divert temporarily and to divert permanently. The instructions for permanent diversion in 8088 are the jump instructions, while the instruction for temporary diversion is the CALL instruction. The word call must be familiar to the readers from subroutine call in higher level languages. The CALL instruction allows temporary diversion and therefore reusability of code. Now we can place the code for bubble sort at one place and reuse it again and again. This was not possible with permanent diversion. Actually the 8088 permanent diversion mechanism can be tricked to achieve temporary diversion. However it is not possible without getting into a lot of trouble. The key idea in doing it this way is to use the jump instruction form that takes a register as argument. Therefore this is not impossible but this is not the way it is done.

The natural way to do this is to use the CALL instruction followed by a label, just like JMP is followed by a label. Execution will divert to the code following the label. Till now the operation has been similar to the JMP instruction. When the subroutine completes we need to return. The RET instruction is used for this purpose. The word return holds in its meaning that we are to return from where we came and need no explicit destination. Therefore RET takes no arguments and transfers control back to the instruction following the CALL that took us in this subroutine. The actual technical process that informs RET where to return will be discussed later after we have discussed the system stack.

CALL takes a label as argument and execution starts from that label, until the RET instruction is encountered and it takes execution back to the instruction following the CALL. Both the instructions are commonly used as a pair, however technically they are independent in their operation. The RET works regardless of the CALL and the CALL works regardless of the RET. If you CALL a subroutine it will not complain if there is no RET present and similarly if you RET without being called it won't complain. It is a logical pair and is used as a pair in every decent code. However sometimes we play tricks with the processor and we use CALL or RET alone. This will become clear when we need to play such tricks in later chapters.

Parameters

We intend to write the bubble sort code at one place and CALL it whenever needed. An immediately visible problem is that whenever we call this subroutine it will sort the same array in the same order. However in a real application we will need to sort various arrays of various sizes. We might sometimes need an ascending sort and descending at other times. Similarly our data may be signed or unsigned. Such pieces of information that may change from invocation to invocation and should be passed from the caller to the subroutine are called parameters.

There must be some way of passing these parameters to the subroutine. Revising the subroutine temporary flow breakage mechanism, the most straightforward way is to use registers. The CALL mechanism breaks the thread of execution and does not change registers, except IP which must change for processor to start executing at another place, and SP whose change will be discussed in detail later. Any of the other registers can hold parameters for the subroutine.

5.2. OUR FIRST SUBROUTINE

Now we want to modify the bubble sort code so that it works as a subroutine. We place a label at the start of bubble sort code, which works as the anchor point and will be used in the CALL instruction to call the subroutine. We also place a RET at the end of the algorithm to return from where we called the subroutine.

Example 5.1

```
01   ; bubble sort algorithm as a subroutine
02   [org 0x0100]
03                 jmp start
04
05   data:         dw   60, 55, 45, 50, 40, 35, 25, 30, 10, 0
06   swap:         db   0
07
08   bubblesort:   dec  cx                 ; last element not compared
09                 shl  cx, 1              ; turn into byte count
10
11   mainloop:     mov  si, 0              ; initialize array index to zero
12                 mov  byte [swap], 0     ; reset swap flag to no swaps
13
14   innerloop:    mov  ax, [bx+si]        ; load number in ax
15                 cmp  ax, [bx+si+2]      ; compare with next number
16                 jbe  noswap             ; no swap if already in order
17
18                 mov  dx, [bx+si+2]      ; load second element in dx
19                 mov  [bx+si], dx        ; store first number in second
20                 mov  [bx+si+2], ax      ; store second number in first
21                 mov  byte [swap], 1     ; flag that a swap has been done
22
23   noswap:       add  si, 2              ; advance si to next index
24                 cmp  si, cx             ; are we at last index
25                 jne  innerloop          ; if not compare next two
26
27                 cmp  byte [swap], 1     ; check if a swap has been done
28                 je   mainloop           ; if yes make another pass
29
30                 ret                     ; go back to where we came from
31
32   start:        mov  bx, data           ; send start of array in bx
33                 mov  cx, 10             ; send count of elements in cx
34                 call bubblesort         ; call our subroutine
35
36                 mov  ax, 0x4c00         ; terminate program
37                 int  0x21
```

08-09 The routine has received the count of elements in CX. Since it makes one less comparison than the number of elements it decrements it. Then it multiplies it by two since this a word array and each element

	takes two bytes. Left shifting has been used to multiply by two.
14	Base+index+offset addressing has been used. BX holds the start of array, SI the offset into it and an offset of 2 when the next element is to be read. BX can be directly changed but then a separate counter would be needed, as SI is directly compared with CX in our case.
32-37	The code starting from the start label is our main program analogous to the main in the C language. BX and CX hold our parameters for the bubblesort subroutine and the CALL is made to invoke the subroutine.

Inside the debugger we observe the same unsigned data that we are so used to now. The number 0103 is passed via BX to the subroutine which is the start of our data and the number 000A via CX which is the number of elements in our data. If we step over the CALL instruction we see our data sorted in a single step and we are at the termination instructions. The processor has jumped to the bubblesort routine, executed it to completion, and returned back from it but the process was hidden due to the step over command. If however we trace into the CALL instruction, we land at the first instruction of our routine. At the end of the routine, when the RET instruction is executed, we immediately land back to our termination instructions, to be precise the instruction following the CALL.

Also observe that with the CALL instruction SP is decremented by two from FFFE to FFFC, and the stack windows shows 0150 at its top. As the RET is executed SP is recovered and the 0150 is also removed from the stack. Match it with the address of the instruction following the CALL which is 0150 as well. The 0150 removed from the stack by the RET instruction has been loaded into the IP register thereby resuming execution from address 0150. CALL placed where to return on the stack for the RET instruction. The stack is automatically used with the CALL and RET instructions. Stack will be explained in detail later, however the idea is that the one who is departing stores the address to return at a known place. This is the place using which CALL and RET coordinate. How this placed is actually used by the CALL and RET instructions will be described after the stack is discussed.

After emphasizing reusability so much, it is time for another example which uses the same bubblesort routine on two different arrays of different sizes.

Example 5.2

```
; bubble sort subroutine called twice
[org 0x0100]
              jmp start

data:         dw   60, 55, 45, 50, 40, 35, 25, 30, 10, 0
data2:        dw   328, 329, 898, 8923, 8293, 2345, 10, 877, 355, 98
              dw   888, 533, 2000, 1020, 30, 200, 761, 167, 90, 5
swap:         db   0

bubblesort:   dec  cx                 ; last element not compared
              shl  cx, 1              ; turn into byte count

mainloop:     mov  si, 0              ; initialize array index to zero
              mov  byte [swap], 0     ; reset swap flag to no swaps

innerloop:    mov  ax, [bx+si]        ; load number in ax
              cmp  ax, [bx+si+2]      ; compare with next number
              jbe  noswap             ; no swap if already in order

              mov  dx, [bx+si+2]      ; load second element in dx
              mov  [bx+si], dx        ; store first number in second
              mov  [bx+si+2], ax      ; store second number in first
              mov  byte [swap], 1     ; flag that a swap has been done

noswap:       add  si, 2              ; advance si to next index
```

```
26                cmp  si, cx            ; are we at last index
27                jne  innerloop         ; if not compare next two
28
29                cmp  byte [swap], 1    ; check if a swap has been done
30                je   mainloop          ; if yes make another pass
31
32                ret                    ; go back to where we came from
33
34  start:        mov  bx, data          ; send start of array in bx
35                mov  cx, 10            ; send count of elements in cx
36                call bubblesort        ; call our subroutine
37
38                mov bx, data2          ; send start of array in bx
39                mov cx, 20             ; send count of elements in cx
40                call bubblesort        ; call our subroutine again
41
42                mov  ax, 0x4c00        ; terminate program
43                int  0x21
```

05-07 There are two different data arrays declared. One of 10 elements and the other of 20 elements. The second array is declared on two lines, where the second line is continuation of the first. No additional label is needed since they are situated consecutively in memory.

34-40 The other change is in the main where the bubblesort subroutine is called twice, once on the first array and once on the second.

Inside the debugger observe that stepping over the first call, the first array is sorted and stepping over the second call the second array is sorted. If however we step in SP is decremented and the stack holds 0178 which is the address of the instruction following the call. The RET consumes that 0178 and restores SP. The next CALL places 0181 on the stack and SP is again decremented. The RET consumes this number and execution resumes from the instruction at 0181. This is the coordinated function of CALL and RET using the stack.

In both of the above examples, there is a shortcoming. The subroutine to sort the elements is destroying the registers AX, CX, DX, and SI. That means that the caller of this routine has to make sure that it does not hold any important data in these registers before calling this function, because after the call has returned the registers will be containing meaningless data for the caller. With a program containing thousands of subroutines expecting the caller to remember the set of modified registers for each subroutine is unrealistic and unreasonable. Also registers are limited in number, and restricting the caller on the use of register will make the caller's job very tough. This shortcoming will be removed using the very important system stack.

5.3. STACK

Stack is a data structure that behaves in a first in last out manner. It can contain many elements and there is only one way in and out of the container. When an element is inserted it sits on top of all other elements and when an element is removed the one sitting at top of all others is removed first. To visualize the structure consider a test tube and put some balls in it. The second ball will come above the first and the third will come above the second. When a ball is taken out only the one at the top can be removed. The operation of placing an element on top of the stack is called pushing the element and the operation of removing an element from the top of the stack is called popping the element. The last thing pushed is popped out first; the last in first out behavior.

We can peek at any ball inside the test tube but we cannot remove it without removing every ball on top of it. Similarly we can read any element from the stack but cannot remove it without removing everything above it. The stack operations of pushing and popping only work at the top of the stack. This top of stack is contained in the SP register. The physical address of the stack is obtained by the SS:SP combination. The stack segment registers tells where the stack is located and the stack pointer marks the top of stack inside this segment.

Whenever an element is pushed on the stack SP is decremented by two as the 8088 stack works on word sized elements. Single bytes cannot be pushed or popped from the stack. Also it is a decrementing stack. Another possibility is an incrementing stack. A decrementing stack moves from higher addresses to lower addresses as elements are added in it while an incrementing stack moves from lower addresses to higher addresses as elements are added. There is no special reason or argument in favor of one or another, and more or less depends on the choice of the designers. Another processor 8051 by the same manufacturer has an incrementing stack while 8088 has a decrementing one.

Memory is like a shelf numbered as zero at the top and the maximum at the bottom. If a decrementing stack starts at shelf 5, the first item is placed in shelf 5, the next item is placed in shelf 4, the next in shelf 3 and so on. The operations of placing items on the stack and removing them from there are called push and pop. The push operation copies its operand on the stack, while the pop operation makes a copy from the top of the stack into its operand. When an item is pushed on a decrementing stack, the top of the stack is first decremented and the element is then copied into this space. With a pop the element at the top of the stack is copied into the pop operand and the top of stack is incremented afterwards.

The basic use of the stack is to save things and recover from there when needed. For example we discussed the shortcoming in our last example that it destroyed the caller's registers, and the callers are not supposed to remember which registers are destroyed by the thousand routines they use. Using the stack the subroutine can save the caller's value of the registers on the stack, and recover them from there before returning. Meanwhile the subroutine can freely use the registers. From the caller's point of view if the registers contain the same value before and after the call, it doesn't matter if the subroutine used them meanwhile.

Similarly during the CALL operation, the current value of the instruction pointer is automatically saved on the stack, and the destination of CALL is loaded in the instruction pointer. Execution therefore resumes from the destination of CALL. When the RET instruction is executed, it recovers the value of the instruction pointer from the stack. The next instruction executed is therefore the one following the CALL. Observe how playing with the instruction pointer affects the program flow.

There is a form of the RET instruction called "RET n" where n is a numeric argument. After performing the operation of RET, it further increments the stack pointer by this number, i.e. SP is first incremented by two and then by n. Its function will become clear when parameter passing is discussed.

Now we describe the operation of the stack in CALL and RET with an example. The top of stack stored in the stack pointer is initialized at 2000. The space above SP is considered empty and free. When the stack pointer is decremented by two, we took a word from the empty space and can use it for our purpose. The unit of stack operations is a word. Some instructions push multiple words; however byte pushes cannot be made. Now the value 017B is stored in the word reserved on the stack. The RET will copy this value in the instruction pointer and increment the stack pointer by two making it 2000 again, thereby reverting the operation of CALL.

This is how CALL and RET behave for near calls. There is also a far version of these functions when the target routine is in another segment. This version of CALL takes a segment offset pair just like the far jump instruction. The CALL will push both the segment and the offset on the stack in this case, followed by loading CS and IP with the values given in the instruction.

The corresponding instruction RETF will pop the offset in the instruction pointer followed by popping the segment in the code segment register.

Apart from CALL and RET, the operations that use the stack are PUSH and POP. Two other operations that will be discussed later are INT and IRET. Regarding the stack, the operation of PUSH is similar to CALL however with a register other than the instruction pointer. For example “push ax” will push the current value of the AX register on the stack. The operation of PUSH is shown below.

```
SP ← SP – 2
[SP] ← AX
```

The operation of POP is the reverse of this. A copy of the element at the top of the stack is made in the operand, and the top of the stack is incremented afterwards. The operation of “pop ax” is shown below.

```
AX ← [SP]
SP ← SP + 2
```

Making corresponding PUSH and POP operations is the responsibility of the programmer. If “push ax” is followed by “pop dx” effectively copying the value of the AX register in the DX register, the processor won’t complain. Whether this sequence is logically correct or not should be ensured by the programmer. For example when PUSH and POP are used to save and restore registers from the stack, order must be correct so that the saved value of AX is reloaded in the AX register and not any other register. For this the order of POP operations need to be the reverse of the order of PUSH operations.

Now we consider another example that is similar to the previous examples, however the code to swap the two elements has been extracted into another subroutine, so that the formation of stack can be observed during nested subroutine calls.

Example 5.3

```
; bubble sort subroutine using swap subroutine
[org 0x0100]
              jmp start

data:         dw   60, 55, 45, 50, 40, 35, 25, 30, 10, 0
data2:        dw   328, 329, 898, 8923, 8293, 2345, 10, 877, 355, 98
              dw   888, 533, 2000, 1020, 30, 200, 761, 167, 90, 5
swapflag:     db   0

swap:         mov  ax, [bx+si]        ; load first number in ax
              xchg ax, [bx+si+2]      ; exchange with second number
              mov  [bx+si], ax        ; store second number in first
              ret                     ; go back to where we came from

bubblesort:   dec  cx                 ; last element not compared
              shl  cx, 1              ; turn into byte count

mainloop:     mov  si, 0              ; initialize array index to zero
              mov  byte [swapflag], 0 ; reset swap flag to no swaps

innerloop:    mov  ax, [bx+si]        ; load number in ax
              cmp  ax, [bx+si+2]      ; compare with next number
              jbe  noswap             ; no swap if already in order

              call swap               ; swaps two elements
              mov  byte [swapflag], 1 ; flag that a swap has been done

noswap:       add  si, 2              ; advance si to next index
              cmp  si, cx             ; are we at last index
              jne  innerloop          ; if not compare next two

              cmp  byte [swapflag], 1 ; check if a swap has been done
              je   mainloop           ; if yes make another pass
              ret                     ; go back to where we came from

start:        mov  bx, data           ; send start of array in bx
              mov  cx, 10             ; send count of elements in cx
```

```
38                call bubblesort         ; call our subroutine
39
40                mov bx, data2           ; send start of array in bx
41                mov cx, 20              ; send count of elements in cx
42                call bubblesort         ; call our subroutine again
43
44                mov  ax, 0x4c00         ; terminate program
45                int  0x21
```

11 A new instruction XCHG has been introduced. The instruction swaps its source and its destination operands however at most one of the operands could be in memory, so the other has to be loaded in a register. The instruction has reduced the code size by one instruction.

13 The RET at the end of swap makes it a subroutine.

Inside the debugger observe the use of stack by CALL and RET instructions, especially the nested CALL.

5.4. SAVING AND RESTORING REGISTERS

The subroutines we wrote till now have been destroying certain registers and our calling code has been carefully written to not use those registers. However this cannot be remembered for a good number of subroutines. Therefore our subroutines need to implement some mechanism of retaining the callers' value of any registers used.

The trick is to use the PUSH and POP operations and save the callers' value on the stack and recover it from there on return. Our swap subroutine destroyed the AX register while the bubblesort subroutine destroyed AX, CX, and SI. BX was not modified in the subroutine. It had the same value at entry and at exit; it was only used by the subroutine. Our next example improves on the previous version by saving and restoring any registers that it will modify using the PUSH and POP operations.

Example 5.4

```
; bubble sort and swap subroutines saving and restoring registers
[org 0x0100]
              jmp start

data:         dw   60, 55, 45, 50, 40, 35, 25, 30, 10, 0
data2:        dw   328, 329, 898, 8923, 8293, 2345, 10, 877, 355, 98
              dw   888, 533, 2000, 1020, 30, 200, 761, 167, 90, 5
swapflag:     db   0

swap:         push ax                 ; save old value of ax

              mov  ax, [bx+si]        ; load first number in ax
              xchg ax, [bx+si+2]      ; exchange with second number
              mov  [bx+si], ax        ; store second number in first

              pop  ax                 ; restore old value of ax
              ret                     ; go back to where we came from

bubblesort:   push ax                 ; save old value of ax
              push cx                 ; save old value of cx
              push si                 ; save old value of si

              dec  cx                 ; last element not compared
              shl  cx, 1              ; turn into byte count

mainloop:     mov  si, 0              ; initialize array index to zero
              mov  byte [swapflag], 0 ; reset swap flag to no swaps

innerloop:    mov  ax, [bx+si]        ; load number in ax
              cmp  ax, [bx+si+2]      ; compare with next number
              jbe  noswap             ; no swap if already in order

```

```
33                   call swap              ; swaps two elements
34                   mov  byte [swapflag], 1 ; flag that a swap has been done
35
36    noswap:        add  si, 2             ; advance si to next index
37                   cmp  si, cx            ; are we at last index
38                   jne  innerloop         ; if not compare next two
39
40                   cmp  byte [swapflag], 1 ; check if a swap has been done
41                   je   mainloop          ; if yes make another pass
42
43                   pop  si                ; restore old value of si
44                   pop  cx                ; restore old value of cx
45                   pop  ax                ; restore old value of ax
46                   ret                    ; go back to where we came from
47
48    start:         mov  bx, data          ; send start of array in bx
49                   mov  cx, 10            ; send count of elements in cx
50                   call bubblesort        ; call our subroutine
51
52                   mov  bx, data2         ; send start of array in bx
53                   mov  cx, 20            ; send count of elements in cx
54                   call bubblesort        ; call our subroutine again
55
56                   mov  ax, 0x4c00        ; terminate program
57                   int  0x21
```

19-21 When multiple registers are pushed, order is very important. If AX, CX, and SI are pushed in this order, they must be popped in the reverse order of SI, CX, and AX. This is again because the stack behaves in a Last In First Out manner.

Inside the debugger we can observe that the registers before and after the CALL operation are exactly identical. Effectively the caller can assume the registers are untouched. By tracing into the subroutines we can observe how their value is saved on the stack by the PUSH instructions and recovered from their before exit. Saving and restoring registers this way in subroutines is a standard way and must be followed.

PUSH

PUSH decrements SP (the stack pointer) by two and then transfers a word from the source operand to the top of stack now pointed to by SP. PUSH often is used to place parameters on the stack before calling a procedure; more generally, it is the basic means of storing temporary data on the stack.

POP

POP transfers the word at the current top of stack (pointed to by SP) to the destination operand and then increments SP by two to point to the new top of stack. POP can be used to move temporary variables from the stack to registers or memory.

Observe that the operand of PUSH is called a source operand since the data is moving to the stack from the opernad, while the operand of POP is called destination since data is moving from the stack to the operand.

CALL

CALL activates an out-of-line procedure, saving information on the stack to permit a RET (return) instruction in the procedure to transfer control back to the instruction following the CALL. For an intrasegment direct CALL, SP is decremented by two and IP is pushed onto the stack. The target procedure’s relative displacement from the CALL instruction is then added to the instruction pointer. For an intersegment direct CALL, SP is decremented by two, and CS is pushed onto the stack. CS is replaced by the segment word contained in the instruction. SP again is decremented by two. IP is pushed onto the stack and replaced by the offset word in the instruction.

The out-of-line procedure is the temporary division, the concept of roundabout that we discussed. Near calls are also called intrasegment calls, while far calls are called intersegment calls. There are also versions that are called indirect calls; however they will be discuss later when they are used.

RET

RET (Return) transfers control from a procedure back to the instruction following the CALL that activated the procedure. RET pops the word at the top of the stack (pointed to by register SP) into the instruction pointer and increments SP by two. If RETF (intersegment RET) is used the word at the top of the stack is popped into the IP register and SP is incremented by two. The word at the new top of stack is popped into the CS register, and SP is again incremented by two. If an optional pop value has been specified, RET adds that value to SP. This feature may be used to discard parameters pushed onto the stack before the execution of the CALL instruction.

5.5. PARAMETER PASSING THROUGH STACK

Due to the limited number of registers, parameter passing by registers is constrained in two ways. The maximum parameters a subroutine can receive are seven when all the general registers are used. Also, with the subroutines are themselves limited in their use of registers, and this limited increases when the subroutine has to make a nested call thereby using certain registers as its parameters. Due to this, parameter passing by registers is not expandable and generalizable. However this is the fastest mechanism available for passing parameters and is used where speed is important.

Considering stack as an alternate, we observe that whatever data is placed there, it stays there, and across function calls as well. For example the bubble sort subroutine needs an array address and the count of elements. If we place both of these on the stack, and call the subroutine afterwards, it will stay there. The subroutine is invoked with its return address on top of the stack and its parameters beneath it.

To access the arguments from the stack, the immediate idea that strikes is to pop them off the stack. And this is the only possibility using the given set of information. However the first thing popped off the stack would be the return address and not the arguments. This is because the arguments were first pushed on the stack and the subroutine was called afterwards. The arguments cannot be popped without first popping the return address. If a heaving thing falls on someone's leg, the heavy thing is removed first and the leg is not pulled out to reduce the damage. Same is the case with our parameters on which the return address has fallen.

To handle this using PUSH and POP, we must first pop the return address in a register, then pop the operands, and push the return address back on the stack so that RET will function normally. However so much effort doesn't seem to pay back the price. Processor designers should have provided a logical and neat way to perform this operation. They did provided a way and infact we will do this without introducing any new instruction.

Recall that the default segment association of the BP register is the stack segment and the reason for this association had been deferred for now. The reason is to peek inside the stack using the BP register and read the parameters without removing them and without touching the stack pointer. The stack pointer could not be used for this purpose, as it cannot be used in an effective address. It is automatically used as a pointer and cannot be explicitly used. Also the stack pointer is a dynamic pointer and sometimes changes without telling us in the background. It is just that whenever we touch it, it is where we expect it to be. The base pointer is provided as a replacement of the stack pointer so that we can peek inside the stack without modifying the structure of the stack.

When the bubble sort subroutine is called, the stack pointer is pointing to the return address. Two bytes below it is the second parameter and four bytes below is the first parameter. The stack pointer is a reference point to these parameters. If the value of SP is captured in BP, then the return address is located at [bp+0], the second parameter is at [bp+2], and the first parameter is at [bp+4]. This is because SP and BP both had the same value and they both defaulted to the same segment, the stack segment.

This copying of SP into BP is like taking a snapshot or like freezing the stack at that moment. Even if more pushes are made on the stack decrementing the stack pointer, our reference point will not change. The parameters will still be accessible at the same offsets from the base pointer. If however the stack pointer increments beyond the base pointer, the references will become invalid. The base pointer will act as the datum point to access our parameters. However we have destroyed the original value of BP in the process, and this will cause problems in nested calls where both the outer and the ineer subroutines need to access their own parameters. The outer subroutine will have its base pointer destroyed after the call and will be unable to access its parameters.

To solve both of these problems, we reach at the standard way of accessing parameters on the stack. The first two instructions of any subroutines accessing its parameters from the stack are given below.

```
push bp
mov  bp, sp
```

As a result our datum point has shifted by a word. Now the old value of BP will be contained in [bp] and the return address will be at [bp+2]. The second parameters will be [bp+4] while the first one will be at [bp+6]. We give an example of bubble sort subroutine using this standard way of argument passing through stack.

Example 5.5

```
; bubble sort subroutine taking parameters from stack
[org 0x0100]
              jmp start

data:         dw   60, 55, 45, 50, 40, 35, 25, 30, 10, 0
data2:        dw   328, 329, 898, 8923, 8293, 2345, 10, 877, 355, 98
              dw   888, 533, 2000, 1020, 30, 200, 761, 167, 90, 5
swapflag:     db   0

bubblesort:   push bp                 ; save old value of bp
              mov  bp, sp             ; make bp our reference point
              push ax                 ; save old value of ax
              push bx                 ; save old value of bx
              push cx                 ; save old value of cx
              push si                 ; save old value of si

              mov  bx, [bp+6]         ; load start of array in bx
              mov  cx, [bp+4]         ; load count of elements in cx
              dec  cx                 ; last element not compared
              shl  cx, 1              ; turn into byte count

mainloop:     mov  si, 0              ; initialize array index to zero
              mov  byte [swapflag], 0 ; reset swap flag to no swaps

innerloop:    mov  ax, [bx+si]        ; load number in ax
              cmp  ax, [bx+si+2]      ; compare with next number
              jbe  noswap             ; no swap if already in order

              xchg ax, [bx+si+2]      ; exchange ax with second number
              mov  [bx+si], ax        ; store second number in first
              mov  byte [swapflag], 1 ; flag that a swap has been done

noswap:       add  si, 2              ; advance si to next index
              cmp  si, cx             ; are we at last index
              jne  innerloop          ; if not compare next two

```

```
37              cmp  byte [swapflag], 1 ; check if a swap has been done
38              je   mainloop           ; if yes make another pass
39
40              pop  si                 ; restore old value of si
41              pop  cx                 ; restore old value of cx
42              pop  bx                 ; restore old value of bx
43              pop  ax                 ; restore old value of ax
44              pop  bp                 ; restore old value of bp
45              ret  4                  ; go back and remove two params
46
47 start:       mov  ax, data
48              push ax                 ; place start of array on stack
49              mov  ax, 10
50              push ax                 ; place element count on stack
51              call bubblesort         ; call our subroutine
52
53              mov  ax, data2
54              push ax                 ; place start of array on stack
55              mov  ax, 20
56              push ax                 ; place element count on stack
57              call bubblesort         ; call our subroutine again
58
59              mov  ax, 0x4c00         ; terminate program
60              int  0x21
```

11	The value of the stack pointer is captured in the base pointer. With further pushes SP will change but BP will not and therefore we will read parameters from bp+4 and bp+6.
45	The form of RET that takes an argument is used causing four to be added to SP after the return address has been popped in the instruction pointer. This will effectively discard the parameters that are still there on the stack.
47-50	We push the address of the array we want to sort followed by the count of elements. As immediates cannot be directly pushed in the 8088 architecture, we first load it in the AX register and then push the AX register on the stack.

Inside the debugger, concentrate on the operation of BP and the stack. The parameters are placed on the stack by the caller, the subroutine accesses them using the base pointer, and the special form of RET removes them without any extra instruction. The value of stack pointer of FFF6 is turned into FFFE by the RET instruction. This was the value in SP before any of the parameters was pushed.

Stack Clearing by Caller or Callee

Parameters pushed for a subroutine are a waste after the subroutine has returned. They have to be cleared from the stack. Either of the caller and the callee can take the responsibility of clearing them from there. If the callee has to clear the stack it cannot do this easily unless RETn exists. That is why most general processors have this instruction. Stack clearing by the caller needs an extra instruction on behalf of the caller after every call made to the subroutine, unnecessarily increasing instructions in the program. If there are thousand calls to a subroutine the code to clear the stack is repeated a thousand times. Therefore the prevalent convention in most high level languages is stack clearing by the callee; even though the other convention is still used in some languages.

If RETn is not available, stack clearing by the callee is a complicated process. It will have to save the return address in a register, then remove the parameters, and then place back the return address so that RET will function. When this instruction was introduced in processors, only then high level language designers switched to stack clearing by the callee. This is also exactly why RETn adds n to SP after performing the operation of RET. The other way around would be totally useless for our purpose. Consider the stack condition at the time of RET and this will become clear why this will be useless. Also observe that RETn has discarded the arguments rather than popping them as they were no longer of any use either of the caller or the callee.

The strong argument in favour of callee cleared stacks is that the arguments were placed on the stack for the subroutine, the caller did not needed them for itself, so the subroutine is responsible for removing them. Removing the arguments is important as if the stack is not cleared or is partially cleared the stack will eventually become full, SP will reach 0, and thereafter wraparound producing unexpected results. This is called stack overflow. Therefore clearing anything placed on the stack is very important.

5.6. LOCAL VARIABLES

Another important role of the stack is in the creation of local variables that are only needed while the subroutine is in execution and not afterwards. They should not take permanent space like global variables. Local variables should be created when the subroutine is called and discarded afterwards. So that the spaced used by them can be reused for the local variables of another subroutine. They only have meaning inside the subroutine and no meaning outside it.

The most convenient place to store these variables is the stack. We need some special manipulation of the stack for this task. We need to produce a gap in the stack for our variables. This is explained with the help of the swapflag in the bubble sort example.

The swapflag we have declared as a word occupying space permanently is only needed by the bubble sort subroutine and should be a local variable. Actually the variable was introduced with the intent of making it a local variable at this time. The stack pointer will be decremented by an extra two bytes thereby producing a gap in which a word can reside. This gap will be used for our temporary, local, or automatic variable; however we name it. We can decrement it as much as we want producing the desired space, however the decrement must be by an even number, as the unit of stack operation is a word. In our case we needed just one word. Also the most convenient position for this gap is immediately after saving the value of SP in BP. So that the same base pointer can be used to access the local variables as well; this time using negative offsets. The standard way to start a subroutine which needs to access parameters and has local variables is as under.

```
push bp
mov  bp, sp
sub  sp, 2
```

The gap could have been created with a dummy push, but the subtraction makes it clear that the value pushed is not important and the gap will be used for our local variable. Also gap of any size can be created in a single instruction with subtraction. The parameters can still be accessed at bp+4 and bp+6 and the swapflag can be accessed at bp-2. The subtraction in SP was after taking the snapshot; therefore BP is above the parameters but below the loacal variables. The parameters are therefore accessed using positive offsets from BP and the local variables are accessed using negative offsets.

We modify the bubble sort subroutine to use a local variable to store the swap flag. The swap flag remembered whether a swap has been done in a particular iteration of bubble sort.

Example 5.6

```
; bubble sort subroutine using a local variable
[org 0x0100]
              jmp start

data:         dw   60, 55, 45, 50, 40, 35, 25, 30, 10, 0
data2:        dw   328, 329, 898, 8923, 8293, 2345, 10, 877, 355, 98
```

```
07                 dw   888, 533, 2000, 1020, 30, 200, 761, 167, 90, 5
08
09   bubblesort:   push bp                 ; save old value of bp
10                 mov  bp, sp             ; make bp our reference point
11                 sub sp, 2               ; make two byte space on stack
12                 push ax                 ; save old value of ax
13                 push bx                 ; save old value of bx
14                 push cx                 ; save old value of cx
15                 push si                 ; save old value of si
16
17                 mov  bx, [bp+6]         ; load start of array in bx
18                 mov  cx, [bp+4]         ; load count of elements in cx
19                 dec  cx                 ; last element not compared
20                 shl  cx, 1              ; turn into byte count
21
22   mainloop:     mov  si, 0              ; initialize array index to zero
23                 mov  word [bp-2], 0     ; reset swap flag to no swaps
24
25   innerloop:    mov  ax, [bx+si]        ; load number in ax
26                 cmp  ax, [bx+si+2]      ; compare with next number
27                 jbe  noswap             ; no swap if already in order
28
29                 xchg ax, [bx+si+2]      ; exchange ax with second number
30                 mov  [bx+si], ax        ; store second number in first
31                 mov  word [bp-2], 1     ; flag that a swap has been done
32
33   noswap:       add  si, 2              ; advance si to next index
34                 cmp  si, cx             ; are we at last index
35                 jne  innerloop          ; if not compare next two
36
37                 cmp  word [bp-2], 1     ; check if a swap has been done
38                 je   mainloop           ; if yes make another pass
39
40                 pop  si                 ; restore old value of si
41                 pop  cx                 ; restore old value of cx
42                 pop  bx                 ; restore old value of bx
43                 pop  ax                 ; restore old value of ax
44                 mov  sp, bp             ; remove space created on stack
45                 pop  bp                 ; restore old value of bp
46                 ret  4                  ; go back and remove two params
47
48   start:        mov  ax, data
49                 push ax                 ; place start of array on stack
50                 mov  ax, 10
51                 push ax                 ; place element count on stack
52                 call bubblesort         ; call our subroutine
53
54                 mov  ax, data2
55                 push ax                 ; place start of array on stack
56                 mov  ax, 20
57                 push ax                 ; place element count on stack
58                 call bubblesort         ; call our subroutine again
59
60                 mov  ax, 0x4c00         ; terminate program
61                 int  0x21
```

11 A word gap has been created for swap flag. This is equivalent to a dummy push. The registers are pushed above this gap.

23 The swapflag is accessed with [bp-2]. The parameters are accessed in the same manner as the last examples.

44 We are removing the hole that we created. The hole is removed by restoring the value of SP that it had at the time of snapshot or at the value it had before the local variable was created. This can be replaced with “add sp, 2” however the one used in the code is prefereed since it does not require to remember how much space for local variables was allocated in the start. After this operation SP points to the old value of BP from where we can proceed as usual.

We needed memory to store the swap flag. The fact that it is in the stack segment or the data segment doesn’t bother us. This will just change the addressing scheme.

EXERCISES

1. Replace the following valid instruction with a single instruction that has the same effect. Don't consider the effect on flags.

```
    push word L1
    jmp L2
L1:
```

2. Replace the following invalid instructions with a single instruction that has the same effect.

```
a.  pop  ip

b.  mov  ip, L5

c.  sub  sp, 2
    mov  [ss:sp], ax

d.  mov ax, [ss:sp]
    add sp, 2

e.  add  sp, 6
    mov  ip, [ss:sp-6]
```

3. Write a recursive function to calculate the fibonaccii of a number. The number is passed as a parameter via the stack and the calculated fibonaccii number is returned in the AX register. A local variable should be used to store the return value from the first recursive call. Fibonaccii function is defined as follows:

```
fibonaccii(0) = 0
fibonaccii(1) = 1
fibonaccii(n) = fibonaccii(n-1) + fibonaccii(n-2)
```

4. Write the above fibonaccii function iteratively.
   HINT: Use two registers to hold the current and the previous fibonaccii numbers in a loop.
5. Write a function switch_stack meant to change the current stack and will be called as below. The function should destroy no registers.

```
push word [new_stack_segment]
push word [new_stack_offset]
call switch_stack
```

6. Write a function "addtoset" that takes offset of a function and remembers this offset in an array that can hold a maximum of 8 offsets. It does nothing if there are already eight offsets in the set. Write another function "callset" that makes a call to all functions in the set one by one.
7. Do the above exercise such that "callset" does not use a CALL or a JMP to invoke the functions.
   HINT: Setup the stack appropriately such that the RET will execute the first function, its RET execute the next and so on till the last RET returns to the caller of "callset."
8. Make an array of 0x80 bytes and treat it as one of 0x400 bits. Write a function myalloc that takes one argument, the number of bits. It finds that many consecutive zero bits in the array, makes them one, and returns in AX the index of the first bit. Write another function myfree that takes two arguments, index of a bit in the array, and the number of bits. It makes that many consecutive bits zero, whatever their previous values are, starting from the index in the first argument.
9. [Circular Queue] Write functions to implement circular queues. Declare 16x32 words of data for 16 queues numbered from 0 to 15. Each queue has a front index, a rear index and 30 locations for data totaling to 32 words. Declare another word variable whose 16 bits correspond to the 16 queues and a 1 bit signals that the correspdoning queue is used and a 0 bit signals that it is free. Write a function "qcreate" that returns a queue number after finding a free queue or -1 if it failed. Write a function "qdestroy" that marks the queue as free. Write two other functions "qadd" and "qremove" that

can add and remove items from the circular queue. The two functions return 0 if they failed and 1 otherwise.

10. [Linked List] Declare 1024 nodes of four bytes each. The first 2 bytes will be used for data and the next 2 bytes for storing the offset of another node. Also declare a word variable “firstfree” to store the offset of the first free node. Write the following five functions:
    a. “init” chains all 1024 nodes into a list with offset of first node in firstfree, offset of the second node in the later two bytes of the first node and so on. The later two bytes of the last node contains zero.
    b. “createlist” returns the offset of the node stored in firstfree through AX. It sets firstfree to the offset stored in the later two bytes of that node, and it sets the later two bytes of that node to zero.
    c. “insertafter” takes two parameters, the offset of a node and a word data. It removes one node from freelist just like “createlist” and inserts it after the said node and updates the new node’s data part.
    d. “deleteafter” takes a node as its parameter and removes the node immediately after it in the linked list if there is one.
    e. “deletelist” takes a node as its parameters and traverses the linked list starting at this node and removes all nodes from it and add them back to the free list.

6
Display Memory

The debugger gives a very close vision of the processor. That is why every program written till now was executed inside the debugger. Also the debugger is a very useful tool in assembly language program development, since many bugs only become visible when each instruction is independently monitored the way the debugger allows us to do. We will now be using the display screen in character mode, the way DOS uses this screen. The way we will access this screen is specific to the IBM PC.

6.1. ASCII CODES

The computer listens, sees, and speaks in numbers. Even a character is a number inside the computer. For example the keyboard is labeled with characters however when we press ‘A’, a specific number is transferred from the keyboard to the computer. Our program interprets that number as the character ‘A’. When the same number comes on display, the Video Graphics Adapter (VGA) in our computer shows the shape of ‘A’. Even the shape is stored in binary numbers with a one bit representing a pixel on the screen that is turned on and a zero bit representing a pixel that is not glowing. This example is considering a white on black display and no colors. This is the way a shape is drawn on the screen. The interpretation of ‘A’ is performed by the VGA card, while the monitor or CRT (cathode ray tube) only glows the pixels on and turns them off. The keyboard has a key labeled ‘A’ and pressing it the screen shows ‘A’ but all that happened inside was in numbers.

An ‘A’ on any computer and any operating system is an ‘A’ on every other computer and operating system. This is because a standard numeric representation of all commonly used characters has been developed. This is called the ASCII code, where ASCII stands for American Standard Code for Information Interchange. The name depicts that this is a code that allows the interchange of information; ‘A’ written on one computer will remain an ‘A’ on another. The ASCII table lists all defined characters and symbols and their standardized numbers. All ASCII based computers use the same code. There are few other standards like EBCDIC and gray codes, but ASCII has become the most prevalent standard and is used for Internet communication as well. It has become the de facto standard for global communication. The character mode displays of our computer use the ASCII standard. Some newer operating systems use a new standard Unicode but it is not relevant to us in the current discussion.

Standard ASCII has 128 characters with assigned numbers from 0 to 127. When IBM PC was introduced, they extended the standard ASCII and defined 128 more characters. Thus extending the total number of symbols from 128 to 256 numbered from 0 to 255 fitting in an 8-bit byte. The newer characters were used for line drawing, window corners, and some non-English characters. The need for these characters was never felt on teletype terminals, but with the advent of IBM PC and its full screen display, these semi-graphics characters were the need of the day. Keep in mind that at that time there was no graphics mode available.

The extended ASCII code is just a de facto industry standard but it is not defined by an organization like the standard ASCII. Printers, displays, and all other peripherals related to the IBM PC understand the ASCII code. If the code for ‘A’ is sent to the printer, the printer will print the shape of ‘A’, if it is sent to the display, the VGA card will form the shape of ‘A’ on the CRT. If it is sent to another computer via the serial port, the other computer will understand that this is an ‘A’.

The important thing to observe in the ASCII table is the contiguous arrangement of the uppercase alphabets (41-5A), the lowercase alphabets (61-7A), and the numbers (30-39). This helps in certain operations with ASCII, for example converting the case of characters by adding or subtracting 0x20 from it. It also helps in converting a digit into its ASCII representation by adding 0x30 to it.

6.2. DISPLAY MEMORY FORMATION

We will explore the working of the display with ASCII codes, since it is our immediately accessible hardware. When 0x40 is sent to the VGA card, it will turn pixels on and off in such a way that a visual representation of ‘A’ appears on the screen. It has no reality, just an interpretation. In later chapters we will program the VGA controller to display a new shape when the ASCII of ‘A’ is received by it.

The video device is seen by the computer as a memory area containing the ASCII codes that are currently displayed on the screen and a set of I/O ports controlling things like the resolution, the cursor height, and the cursor position. The VGA memory is seen by the computer just like its own memory. There is no difference; rather the computer doesn’t differentiate, as it is accessible on the same bus as the system memory. Therefore if that appropriate block of the screen is cleared, the screen will be cleared. If the ASCII of ‘A’ is placed somewhere in that block, the shape of ‘A’ will appear on the screen at a corresponding place.

This correspondence must be defined as the memory is a single dimensional space while the screen is two dimensional having 80 rows and 25 columns. The memory is linearly mapped on this two dimensional space, just like a two dimensional is mapped in linear memory. There is one word per character in which a byte is needed for the ASCII code and the other byte is used for the character’s attributes discussed later. Now the first 80 words will correspond to the first row of the screen and the next 80 words will correspond to the next row. By making the memory on the video controller accessible to the processor via the system bus, the processor is now in control of what is displayed on the screen.

The three important things that we discussed are.

- One screen location corresponds to a word in the video memory
- The video controller memory is accessible to the processor like its own memory.
- ASCII code of a character placed at a cell in the VGA memory will cause the corresponding ASCII shape to be displayed on the corresponding screen location.

Display Memory Base Address

The memory at which the video controller’s memory is mapped must be a standard, so that the program can be written in a video card independent manner. Otherwise if different vendors map their video memory at different places in the address space, as was the problem in the start, writing software was a headache. BIOS vendors had a problem of dealing with various card vendors. The IBM PC text mode color display is now fixed so that system software can work uniformly. It was fixed at the physical memory location of B8000. The first byte at this location contains the ASCII for the character displayed at the top left of the video screen. Dropping the zero we can load the rest in a segment register to access the video memory. If we do something in this memory, the effect can be seen on the screen. For example we can write a virus that makes any character we write drop to the bottom of the screen.

Attribute Byte

The second byte in the word designated for one screen location holds the foreground and background colors for the character. This is called its video attribute. So the pair of the ASCII code in one byte and the attribute in the second byte makes the word that corresponds to one location on the screen. The lower address contains the code while the higher one contains the attribute. The attribute byte as detailed below has the RGB for the foreground and the background. It has an intensity bit for the foreground color as well thus making 16 possible colors of the foreground and 8 possible colors for the background. When bit 7 is set the character keeps on blinking on the screen. This bit has some more interpretations like background intensity that has to be activated in the video controller through its I/O ports.

7	6	5	4	3	2	1	0

7 – Blinking of foreground character
6 – Red component of background color
5 – Green component of background color
4 – Blue component of background color
3 – Intensity component of foreground color
2 – Red component of foreground color
1 – Green component of foreground color
0 – Blue component of foreground color

Display Examples

Both DS and ES can be used to access the video memory. However we commonly keep DS for accessing our data, and load ES with the segment of video memory. Loading a segment register with an immediate operand is not allowed in the 8088 architecture. We therefore load the segment register via a general purpose register. Other methods are loading from a memory location and a combination of push and pop.

```
mov  ax, 0xb800
mov  es, ax
```

This operation has opened a window to the video memory. Now the following instruction will print an ‘A’ on the top left of the screen in white color on black background.

```
mov  word [es:0], 0x0741
```

The segment override is used since ES is pointing to the video memory. Since the first word is written to, the character will appear at the top left of the screen. The 41 that goes in the lower byte is the ASCII code for ‘A’. The 07 that goes in the higher byte is the attribute with I=0, R=1, G=1, B=1 for the foreground, meaning white color in low intensity and R=0, G=0, B=0 for the background meaning black color and the most significant bit cleared so that there is no blinking. Now consider the following instruction.

```
mov  word [es:160], 0x1230
```

This is displayed 80 words after the start and there are 80 characters in one screen row. Therefore this is displayed on the first column of the second line. The ASCII code used is 30, which represents a ‘0’ while the attribute byte is 12 meaning green color on black background.

We take our first example to clear the screen.

Example 6.1

```
01      ; clear the screen
02      [org 0x0100]
03                    mov  ax, 0xb800         ; load video base in ax
04                    mov  es, ax             ; point es to video base
05                    mov  di, 0              ; point di to top left column
06
07      nextchar:     mov  word [es:di], 0x0720 ; clear next char on screen
08                    add  di, 2              ; move to next screen location
09                    cmp  di, 4000           ; has the whole screen cleared
10                    jne  nextchar           ; if no clear next position
11
12                    mov  ax, 0x4c00         ; terminate program
13                    int  0x21
```

07 The code for space is 20 while 07 is the normal attribute of low intensity white on black with no blinking. Even to clear the screen or put a blank on a location there is a numeric code.

08 DI is incremented twice since each screen location corresponds to two byte in video memory.

09 DI is compared with 80*25*2=4000. The last word location that corresponds to the screen is 3998.

Inside the debugger the operation of clearing the screen cannot be observed since the debugger overwrites whatever is displayed on the screen. Directly executing the COM file from the command prompt[*], we can see that the screen is cleared. The command prompt that reappeared is printed after the termination of our application. This is the first application that can be directly executed to see some output on the screen.

6.3. HELLO WORLD IN ASSEMBLY LANGUAGE

To declare a character in assembly language, we store its ASCII code in a byte. The assembler provides us with another syntax that doesn’t forces us to remember the ASCII code. The assembler also provides a syntax that simplifies declaration of consecutive characters, usually called a string. The three ways used below are identical in their meaning.

```
        db   0x61, 0x61, 0x63
        db   'a', 'b', 'c'
        db   'abc'
```

When characters are stored in any high level or low level language the actual thing stored in a byte is their ASCII code. The only thing the language helps in is a simplified declaration.

Traditionally the first program in higher level languages is to print “hello world” on the screen. However due to the highly granular nature of assembly language, we are only now able to write it in assembly language. In writing this program, we make a generic routine that can print any string on the screen.

Example 6.2

```
; hello world in assembly
[org 0x0100]
              jmp  start

message:      db   'hello world'      ; string to be printed
length:       dw   11                 ; length of the string

; subroutine to clear the screen
clrscr:       push es
              push ax
              push di
```

[*] Remember that if this example is run in a DOS window on some newer operating systems, a full screen DOS application must be run before this program so that screen access is enabled.

```
12
13                mov  ax, 0xb800
14                mov  es, ax             ; point es to video base
15                mov  di, 0              ; point di to top left column
16
17   nextloc:     mov  word [es:di], 0x0720 ; clear next char on screen
18                add  di, 2              ; move to next screen location
19                cmp  di, 4000           ; has the whole screen cleared
20                jne  nextloc            ; if no clear next position
21
22                pop  di
23                pop  ax
24                pop  es
25                ret
26
27   ; subroutine to print a string at top left of screen
28   ; takes address of string and its length as parameters
29   printstr:    push bp
30                mov  bp, sp
31                push es
32                push ax
33                push cx
34                push si
35                push di
36
37                mov  ax, 0xb800
38                mov  es, ax             ; point es to video base
39                mov  di, 0              ; point di to top left column
40                mov  si, [bp+6]         ; point si to string
41                mov  cx, [bp+4]         ; load length of string in cx
42                mov  ah, 0x07           ; normal attribute fixed in al
43
44   nextchar:    mov  al, [si]           ; load next char of string
45                mov  [es:di], ax        ; show this char on screen
46                add  di, 2              ; move to next screen location
47                add  si, 1              ; move to next char in string
48                loop nextchar           ; repeat the operation cx times
49
50                pop  di
51                pop  si
52                pop  cx
53                pop  ax
54                pop  es
55                pop  bp
56                ret  4
57
58   start:       call clrscr             ; call the clrscr subroutine
59
60                mov  ax, message
61                push ax                 ; push address of message
62                push word [length]      ; push message length
63                call printstr           ; call the printstr subroutine
64
65                mov  ax, 0x4c00         ; terminate program
66                int  0x21
```

05-06 The string definition syntax discussed above is used to declare a string “hello world” of 11 bytes and the length is stored in a separate variable.

09-25 The code to clear the screen from the last example is written in the form of a subroutine. Since the subroutine had no parameters, only modified registers are saved and restored from the stack.

29-35 The standard subroutine format with parameters received via stack and all registers saved and restored is used.

37-42 ES is initialized to point to the video memory via the AX register. Two pointer registers are used; SI to point to the string and DI to point to the top left location of the screen. CX is loaded with the length of the string. Normal attribute of low intensity white on black with no blinking is loaded in the AH register.

44-45 The next character from the string is loaded into AL. Now AH holds the attribute and AL the ASCII code of the character. This pair is

46-47	written on the video memory using DI with the segment override prefix for ES to access the video memory segment.
48	The string pointer is incremented by one while the video memory pointer is incremented by two since one char corresponds to a word on the screen.
50-56	The loop instruction used is equivalent to a combination of “dec cx” and “jnz nextchar.” The loop is executed CX times. The registers pushed on the stack are recovered in opposite order and the “ret 4” instruction removes the two parameters placed on the stack.
62	Memory can be directly pushed on the stack.

When the program is executed, screen is cleared and the greetings is displayed on the top left of the screen. This screen location and the attribute used were hard coded in the program and can also be made variable. Then we will be able to print anywhere on the screen.

6.4. NUMBER PRINTING IN ASSEMBLY

Another problem related to the display is printing numbers. Every high level language allows some simple way to print numbers on the screen. As we have seen, everything on the screen is a pair of ASCII code and its attribute and a number is a raw binary number and not a collection of ASCII codes. For example a 10 is stored as a 10 and not as the ASCII code of 1 followed by the ASCII code of 0. If this 10 is stored in a screen location, the output will be meaningless, as the character associate to ASCII code 10 will be shown on the screen. So there is a process that converst a number in its ASCII representation. This process works for any number in any base. We will discuss our examples with respect to the decimal base and later observe the effect of changing to different bases.

Number Printing Algorithm

The key idea is to divide the number by the base number, 10 in the case of decimal. The remainder can be from 0-9 and is the right most digit of the original number. The remaining digits fall in the quotient. The remainder can be easily converted into its ASCII equivalent and printed on the screen. The other digits can be printed in a similar manner by dividing the quotient again by 10 to separate the next digit and so on.

However the problem with this approach is that the first digit printed is the right most one. For example 253 will be printed as 352. The remainder after first division was 3, after second divison was 5 and after the third division was 2. We have to somehow correct the order so that the actual number 253 is displayed, and the trick is to use the stack since the stack is a Last In First Out structure so if 3, 5, and 2 are pushed on it, 2, 5, and 3 will come out in this order. The steps of our algorithm are outlined below.

- Divide the number by base (10 in case of decimal)
- The remainder is its right most digit
- Convert the digit to its ASCII representation (Add 0x30 to the remainder in case of decimal)
- Save this digit on stack
- If the quotient is non-zero repeat the whole process to get the next digit, otherwise stop
- Pop digits one by one and print on screen left to right

DIV Instruction

The divison used in the process is integer divison and not floating point divison. Integer divison gives an integer quotient and an integer remainder. A divison algorithm is now needed. Fortunately or unfortunely there is a DIV instruction available in the 8088 processor. There are two forms of the DIV instruction. The first form divides a 32bit number in DX:AX by its 16bit operand and stores the 16bit quotient in AX and the 16bit remainder in DX. The second form divides a 16bit number in AX by its 8bit operand and stores the 8bit quotient in AL and the 8bit remainder in AH. For example "DIV BL" has an 8bit operand, so the implied dividend is 16bit and is stored in the AX register and "DIV BX" has a 16bit operand, so the implied dividend is 32bit and is therefore stored in the concatentation of the DX and AX registers. The higher word is stored in DX and the lower word in AX.

If a large number is divided by a very small number it is possible that the quotient is larger than the space provided for it in the implied destination. In this case an interrupt is automatically generated and the program is usually terminated as a result. This is called a divide overflow error; just like the calculator shows an –E– when the result cannot be displayed. This interrupt will be discussed later in the discussion of interrupts.

DIV (divide) performs an unsigned division of the accumulator (and its extension) by the source operand. If the source operand is a byte, it is divided into the two-byte dividend assumed to be in registers AL and AH. The byte quotient is returned in AL, and the byte remainder is returned in AH. If the source operand is a word, it is divided into the two-word dividend in registers AX and DX. The word quotient is returned in AX, and the word remainder is returned in DX. If the quotient exceeds the capacity of its destination register (FF for byte source, FFFF for word source), as when division by zero is attempted, a type 0 interrupt is generated, and the quotient and remainder are undefined.

Number Printing Example

The next example introduces a subroutine that can print a number received as its only argument at the top left of the screen using the algorithm just discussed.

Example 6.3

```
; number printing algorithm
[org 0x0100]
              jmp  start

;;;;; COPY LINES 008-025 FROM EXAMPLE 6.2 (clrscr) ;;;;;

; subroutine to print a number at top left of screen
; takes the number to be printed as its parameter
printnum:     push bp
              mov  bp, sp
              push es
              push ax
              push bx
              push cx
              push dx
              push di

              mov  ax, 0xb800
              mov  es, ax             ; point es to video base
              mov  ax, [bp+4]         ; load number in ax
              mov  bx, 10             ; use base 10 for division
              mov  cx, 0              ; initialize count of digits

nextdigit:    mov  dx, 0              ; zero upper half of dividend
              div  bx                 ; divide by 10
              add  dl, 0x30           ; convert digit into ascii value
              push dx                 ; save ascii value on stack
              inc  cx                 ; increment count of values
              cmp  ax, 0              ; is the quotient zero
              jnz  nextdigit          ; if no divide it again

              mov  di, 0              ; point di to top left column

```

```
051  nextpos:      pop  dx                ; remove a digit from the stack
052                mov  dh, 0x07          ; use normal attribute
053                mov [es:di], dx        ; print char on screen
054                add  di, 2             ; move to next screen location
055                loop nextpos           ; repeat for all digits on stack
056
057                pop  di
058                pop  dx
059                pop  cx
060                pop  bx
061                pop  ax
062                pop  es
063                pop  bp
064                ret  2
065
066  start:        call  clrscr           ; call the clrscr subroutine
067
068                mov ax, 4529
069                push ax                ; place number on stack
070                call printnum          ; call the printnum subroutine
071
072                mov ax, 0x4c00         ; terminate program
073                int 0x21
```

026-033 The registers are saved as an essential practice. The only parameter received is the number to be printed.

035-039 ES is initialzed to video memory. AX holds the number to be printed. BX is the desired base, and can be loaded from a parameter. CX holds the number of digits pushed on the stack. This count is initialized to zero, incremented with every digit pushed and is used when the digits are popped one by one.

041-042 DX must be zeroed as our dividend is in AX and we want a 32bit divison. After the divison AX holds the quotient and DX holds the remainder. Actually the remainder is only in DL since the remainder can be from 0 to 9.

043-045 The remainder is converted into its ASCII representation and saved on the stack. The count of digits on the stack is incremented as well.

046-047 If the quotient is zero, all digits have been saved on the stack and if it is non-zero, we have to repeat the process to print the next digit.

049 DI is initialized to point to the top left of the screen, called the cursor home. If the screen location is to become a parameter, the value loaded in DI will change.

051-053 A digit is popped off the stack, the attribute byte is appended to it and it is displayed on the screen.

054-055 The next screen location is two bytes ahead so DI is incremented by two. The process is repeated CX times which holds the number of digits pushed on the stack.

057-064 We pop the registers pushed and “ret 2” to discard the only parameter on the stack.

066-070 The main program clears the screen and calls the printnum subroutine to print 4529 on the top left of the screen.

When the program is executed 4529 is printed on the top left of the screen. This algorithm is versatible in that the base number can be changed and the printing will be in the desired base. For example if “mov bx, 10” is changed to “mov bx, 2” the output will be in binary as 001000110110001. Similarly changing it to “mov bx, 8” outputs the number in octal as 10661. Printing it in hexadecimal is a bit tricky, as the ASCII codes for A-F do not consecutively start after the codes for 0-9. Inside the debugger observe the working of the algorithm is just as described in the above illustration. The digits are separated one by one and saved on the stack. From bottom to top, the stack holds 0034, 0035, 0032, and 0039 after the first loop is completed. The next loop pops them one by one and routes them to the screen.

6.5. SCREEN LOCATION CALCULATION

Until now our algorithms used a fixed attribute and displayed at a fixed screen location. We will change that to use any position on the screen and any attribute. For mapping from the two dimensional coordinate system of the screen to the one dimensional memory, we need to multiply the row number by 80 since there are 80 columns per row and add the column number to it and again multiply by two since there are 2 bytes for each character.

For this purpose the multiplication routine written previously can be used. However we introduce an instruction of the 8088 microprocessor at this time that can multiply 8bit or 16bit numbers.

MUL Instruction

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.

String Printing at Desired Location

We modify the string printing program to take the x-position, the y-position, and the attribute as parameters. The desired location on the screen can be calculated with the following formulae.

```
location = ( ypos * 80 + xpos ) * 2
```

Example 6.4

```
; hello world at desired screen location
[org 0x0100]
              jmp  start

message:      db   'hello world'      ; string to be printed
length:       dw   11                 ; length of the string

;;;;; COPY LINES 008-025 FROM EXAMPLE 6.2 (clrscr) ;;;;;

; subroutine to print a string at top left of screen
; takes x position, y position, string attribute, address of string
; and its length as parameters
printstr:     push bp
              mov  bp, sp
              push es
              push ax
              push cx
              push si
              push di

              mov  ax, 0xb800
              mov  es, ax             ; point es to video base
              mov  al, 80             ; load al with columns per row
              mul  byte [bp+10]       ; multiply with y position
              add  ax, [bp+12]        ; add x position
              shl  ax, 1              ; turn into byte offset
              mov  di,ax              ; point di to required location
              mov  si, [bp+6]         ; point si to string
              mov  cx, [bp+4]         ; load length of string in cx
              mov  ah, [bp+8]         ; load attribute in ah

nextchar:     mov  al, [si]           ; load next char of string
              mov  [es:di], ax        ; show this char on screen
              add  di, 2              ; move to next screen location
```

```
52                  add  si, 1             ; move to next char in string
53                  loop nextchar          ; repeat the operation cx times
54
55                  pop  di
56                  pop  si
57                  pop  cx
58                  pop  ax
59                  pop  es
60                  pop  bp
61                  ret  10
62
63  start:          call clrscr            ; call the clrscr subroutine
64
65                  mov  ax, 30
66                  push ax                ; push x position
67                  mov  ax, 20
68                  push ax                ; push y position
69                  mov  ax, 1             ; blue on black attribute
70                  push ax                ; push attribute
71                  mov  ax, message
72                  push ax                ; push address of message
73                  push word [length]     ; push message length
74                  call printstr          ; call the printstr subroutine
75
76                  mov  ax, 0x4c00        ; terminate program
77                  int  0x21
```

41	Push and pop operations always operate on words; however data can be read as a word or as a byte. For example we read the lower byte of the parameter y-position in this case.
43	Shifting is used for multiplication by two, which should always be the case when multiplication or divison by a power of two is desired.
61	The subroutine had 5 parameters so “ret 10” is used.
65-74	The main program pushes 30 as x-position, 20 as y-position meaning 30th column on 20th row. It pushes 1 as the attribute meaning low intensity blue on black with no blinking.

When the program is executed hello world is displayed at the desired screen location in the desired color. The x-position, y-position, and attribute parameters can be changed and their effect be seen on the screen. The important difference in this example is the use of MUL instruction and the calculation of screen location given the x and y positions.

EXERCISES

1. Replace the following valid instruction with a single instruction that has the same effect. Don’t consider the effect on flags.

   ```
   dec cx
   jnz L3
   ```

2. Write an infinite loop that shows two asterisks moving from right and left centers of the screen to the middle and then back. Use two empty nested loops with large counters to introduce some delay so that the movement is noticeable.
3. Write a function “printaddr” that takes two parameters, the segment and offset parts of an address, via the stack. The function should print the physical address corresponding to the segment offset pair passed at the top left of the screen. The address should be printed in hex and will therefore occupy exactly five columns. For example, passing 5600 and 7800 as parameters should result in 5D800 printed at the top left of the screen.
4. Write code that treats an array of 500 bytes as one of 4000 bits and for each blank position on the screen (i.e. space) sets the corresponding bit to zero and the rest to one.

5. Write a function “drawrect” that takes four parameters via the stack. The parameters are top, left, bottom, and right in this order. The function should display a rectangle on the screen using the characters + - and |.

7
String Instructions

7.1. STRING PROCESSING

Till now very simple instructions of the 8088 microprocessor have been introduced. In this chapter we will discuss a bit more powerful instructions that can process blocks of data in one go. They are called block processing or string instructions. This is the appropriate place to discuss these instructions as we have just introduced a block of memory, which is the video memory. The vision of this memory for the processor is just a block of memory starting at a special address. For example the clear screen operation initializes this whole block to 0741.

There are just 5 block processing instructions in 8088. In the primitive form, the instructions themselves operate on a single cell of memory at one time. However a special prefix repeats the instruction in hardware called the REP prefix. The REP prefix allows these instructions to operate on a number of data elements in one instruction. This is not like a loop; rather this repetition is hard coded in the processor. The five instructions are STOS, LODS, CMPS, SCAS, and MOVS called store string, load string, compare string, scan string, and move string respectively. MOVS is the instruction that allows memory to memory moves, as was discussed in the exceptions to the memory to memory movement rules. String instructions are complex instruction in that they perform a number of tasks against one instruction. And with the REP prefix they perform the task of a complex loop in one instruction. This causes drastic speed improvements in operations on large blocks of memory. The reduction in code size and the improvement in speed are the two reasons why these instructions were introduced in the 8088 processor.

There are a number of common things in these instructions. Firstly they all work on a block of data. DI and SI are used to access memory. SI and DI are called source index and destination index because of string instructions. Whenever an instruction needs a memory source, DS:SI holds the pointer to it. An override is possible that can change the association from DS but the default is DS. Whenever a string instruction needs a memory destination, ES:DI holds the pointer to it. No override is possible in this case. Whenever a byte register is needed, AL holds the value. Whenever a word register is used AX holds the value. For example STOS stores a register in memory so AL or AX is the register used and ES:DI points to the destination. The LODS instruction loads from memory to register so the source is pointed to by DS:SI and the register used is AL or AX.

String instructions work on a block of data. A block has a start and an end. The instructions can work from the start towards the end and from the end towards the start. In fact they can work in both directions, and they must be allowed to work in both directions otherwise certain operations with overlapping blocks become impossible. This problem is discussed in detail later. The direction of movement is controlled with the Direction Flag (DF) in the flags register. If this flag is cleared the direction is from lower addresses towards higher addresses and if this flag is set the direction is from higher addresses to lower addresses. If DF is cleared, this is called the auto-increment mode of string instruction, and if DF is set, this is called the auto-decrement mode. There are two instructions to set and clear the direction flag.

```
        cld     ; clear direction flag
        std     ; set direction flag
```

Every string instruction has two variants; a byte variant and a word variant. For example the two variants of STOS are STOSB and STOSW. Similarly the variants for the other string instructions are attained by appending a B or a W to the instruction name. The operation of each of the string instructions and each of the repetition prefixes is discussed below.

STOS

STOS transfers a byte or word from register AL or AX to the string element addressed by ES:DI and updates DI to point to the next location. STOS is often used to clear a block of memory or fill it with a constant.

The implied source will always be in AL or AX. If DF is clear, SI will be incremented by one or two depending of whether STOSB or STOSW is used. If DF is set SI will be decremented by one or two depending of whether STOSB or STOSW is used. If REP is used before this instruction, the process will be repeated CX times. CX is called the counter register because of the special treatment given to it in the LOOP and JCXZ instructions and the REP set of prefixes. So if REP is used with STOS the whole block of memory will be filled with a constant value. REP will always decrement CX like the LOOP instruction and this cannot be changed with the direction flag. It is also independent of whether the byte or the word variant is used. It always decrements by one; therefore CX has count of repetitions and not the count of bytes.

LODS

LODS transfers a byte or word from the source location DS:SI to AL or AX and updates SI to point to the next location. LODS is generally used in a loop and not with the REP prefix since the value previously loaded in the register is overwritten if the instruction is repeated and only the last value of the block remains in the register.

SCAS

SCAS compares a source byte or word in register AL or AX with the destination string element addressed by ES:DI and updates the flags. DI is updated to point to the next location. SCAS is often used to locate equality or in-equality in a string through the use of an appropriate prefix.

SCAS is a bit different from the other instructions. This is more like the CMP instruction in that it does subtraction of its operands. The prefixes REPE (repeat while equal) and REPNE (repeat while not equal) are used with this instruction. The instruction is used to locate a byte in AL in the block of memory. When the first equality or inequality is encountered; both have uses. For example this instruction can be used to search for a 0 in a null terminated string to calculate the length of the string. In this form REPNE will be used to repeat while the null is not there.

MOVS

MOVS transfers a byte or word from the source location DS:SI to the destination ES:DI and updates SI and DI to point to the next locations. MOVS is used to move a block of memory. The DF is important in the case of overlapping blocks. For example when the source and destination blocks overlap and the source is below the destination copy must be done upwards while if the destination is below the source copy must be done downwards. We cannot perform both these copy operations properly if the direction flag was not provided. If the source is below the destination and an upwards copy is used the source to be copied is destroyed. If however the copy is done downwards the portion of source destroyed is the one that has already been copied. Therefore we need the control of the direction flag to handle this problem. This problem is further detailed in a later example.

CMPS

CMPS subtracts the source location DS:SI from the destination location ES:DI. Source and Destination are unaffected. SI and DI are updated accordingly. CMPS compares two blocks of memory for equality or inequality of the block. It subtracts byte by byte or word by word. If used with a REPE or a REPNE prefix is repeats as long as the blocks are same or as long as they are different. For example it can be used for find a substring. A substring is a string that is contained in another string. For example “has” is contained in “Mary has a little lamp.” Using CMPS we can do the operation of a complex loop in a single instruction. Only the REPE and REPNE prefixes are meaningful with this instruction.

REP Prefix

REP repeats the following string instruction CX times. The use of CX is implied with the REP prefix. The decrement in CX doesn’t affect any flags and the jump is also independent of the flags, just like JCXZ.

REPE and REPNE Prefixes

REPE or REPZ repeat the following string instruction while the zero flag is set and REPNE or REPNZ repeat the following instruction while the zero flag is not set. REPE or REPNE are used with the SCAS or CMPS instructions. The other string instructions have nothing to do with the condition since they are performing no comparison. Also the initial state of flags before the string instruction does not affect the operation. The most complex operation of the string instruction is with these prefixes.

7.2. STOS EXAMPLE – CLEARING THE SCREEN

We take the example of clearing the screen and observe that how simple and fast this operation is with the string instructions. Even if there are three instructions in a loop they have to be fetched and decoded with every iteration and the time of three instructions is multiplied by the number of iterations of the loop. In the case of string instructions, many operations are short circuited. The instruction is fetched and decoded once and only the execution is repeated CX times. That is why string instructions are so efficient in their operation. The program to clear the screen places 0720 on the 2000 words on the screen.

Example 7.1

```
; clear screen using string instructions
[org 0x0100]
              jmp  start

; subroutine to clear the screen
clrscr:       push es
              push ax
              push cx
              push di

              mov  ax, 0xb800
              mov  es, ax             ; point es to video base
              xor  di, di             ; point di to top left column
              mov  ax, 0x0720         ; space char in normal attribute
              mov  cx, 2000           ; number of screen locations

              cld                     ; auto increment mode
              rep  stosw              ; clear the whole screen

              pop  di
```

```
021                 pop  cx
022                 pop  ax
023                 pop  es
024                 ret
025
026     start:      call clrscr            ; call clrscr subroutine
027
028                 mov  ax, 0x4c00        ; terminate program
029                 int  0x21
```

013 A space efficient way to zero a 16bit register is to XOR it with itself. Remember that exclusive or results in a zero whenever the bits at the source and at the destination are same. This instruction takes just two bytes compared to “mov di, 0” which would take three. This is a standard way to zero a 16bit register.

Inside the debugger the operation of the string instruction can be monitored. The trace into command can be used to monitor every repetition of the string instruction. However screen will not be cleared inside the debugger as the debugger overwrites its display on the screen. CX decrements with every iteration, DI increments by 2. The first access is made at B800:0000 and the second at B800:0002 and so on. A complex and inefficient loop is replaced with a fast and simple instruction that does the same operation many times faster.

7.3. LODS EXAMPLE – STRING PRINTING

The use of LODS with the REP prefix is not meaningful as only the last value loaded will remain in the register. It is normally used in a loop paired with a STOS instruction to do some block processing. We use LODS to pick the data, do the processing, and then use STOS to put it back or at some other place. For example in string printing, we will use LODS to read a character of the string, attach the attribute byte to it, and use STOS to write it on the video memory.

The following example will print the string using string instructions.

Example 7.2

```
001     ; hello world printing using string instructions
002     [org 0x0100]
003                 jmp  start
004
005     message:    db   'hello world'     ; string to be printed
006     length:     dw   11                ; length of string
007
008-027 ;;;;; COPY LINES 005-024 FROM EXAMPLE 7.1 (clrscr) ;;;;;
028
029     ; subroutine to print a string
030     ; takes the x position, y position, attribute, address of string and
031     ; its length as parameters
032     printstr:   push bp
033                 mov  bp, sp
034                 push es
035                 push ax
036                 push cx
037                 push si
038                 push di
039
040                 mov  ax, 0xb800
041                 mov  es, ax            ; point es to video base
042                 mov  al, 80            ; load al with columns per row
043                 mul  byte [bp+10]      ; multiply with y position
044                 add  ax, [bp+12]       ; add x position
045                 shl  ax, 1             ; turn into byte offset
046                 mov  di,ax             ; point di to required location
047                 mov  si, [bp+6]        ; point si to string
048                 mov  cx, [bp+4]        ; load length of string in cx
049                 mov  ah, [bp+8]        ; load attribute in ah
```

```
050
051                 cld                     ; auto increment mode
052   nextchar:     lodsb                   ; load next char in al
053                 stosw                   ; print char/attribute pair
054                 loop nextchar           ; repeat for the whole string
055
056                 pop  di
057                 pop  si
058                 pop  cx
059                 pop  ax
060                 pop  es
061                 pop  bp
062                 ret  10
063
064   start:        call clrscr             ; call the clrscr subroutine
065
066                 mov  ax, 30
067                 push ax                 ; push x position
068                 mov  ax, 20
069                 push ax                 ; push y position
070                 mov  ax, 1              ; blue on black attribute
071                 push ax                 ; push attribute
072                 mov  ax, message
073                 push ax                 ; push address of message
074                 push word [length]      ; push message length
075                 call printstr           ; call the printstr subroutine
076
077                 mov  ax, 0x4c00         ; terminate program
078                 int  0x21
```

051	Both operations are in auto increment mode.
052-053	DS is automatically initialized to our segment. ES points to video memory. SI points to the address of our string. DI points to the screen location. AH holds the attribute. Whenever we read a character from the string in AL, the attribute byte is implicitly attached and the pair is present in AX. The same effect could not be achieved with a REP prefix as the REP will repeat LODS and then start repeating STOS, but we need to alternate them.
054	CX holds the length of the string. Therefore LOOP repeats for each character of the string.

Inside the debugger we observe how LODS and STOS alternate and CX is only used by the LOOP instruction. In the original code there were four instructions inside the loop; now there are only two. This is how string instructions help in reducing code size.

7.4. SCAS EXAMPLE – STRING LENGTH

Many higher level languages do not explicitly store string length; rather they use a null character, a character with an ASCII code of zero, to signal the end of a string. In assembly language programs, it is also easier to store a zero at the end of the string, instead of calculating the length of string, which is very difficult process for longer strings. So we delegate length calculation to the processor and modify our string printing subroutine to take a null terminated string and no length. We use SCASB with REPNE and a zero in AL to find a zero byte in the string. In CX we load the maximum possible size, which is 64K bytes. However actual strings will be much smaller. An important thing regarding SCAS and CMPS is that if they stop due to equality or inequality, the index registers have already incremented. Therefore when SCAS will stop DI would be pointing past the null character.

Example 7.3

```
; hello world printing with a null terminated string
[org 0x0100]
```

```
003                jmp  start
004
005     message:      db   'hello world', 0    ; null terminated string
006
007-026 ;;;;; COPY LINES 005-024 FROM EXAMPLE 7.1 (clrscr) ;;;;;
027
028     ; subroutine to print a string
029     ; takes the x position, y position, attribute, and address of a null
030     ; terminated string as parameters
031     printstr:     push bp
032                   mov  bp, sp
033                   push es
034                   push ax
035                   push cx
036                   push si
037                   push di
038
039                   push ds
040                   pop  es                  ; load ds in es
041                   mov  di, [bp+4]          ; point di to string
042                   mov  cx, 0xffff          ; load maximum number in cx
043                   xor  al, al              ; load a zero in al
044                   repne scasb              ; find zero in the string
045                   mov  ax, 0xffff          ; load maximum number in ax
046                   sub  ax, cx              ; find change in cx
047                   dec  ax                  ; exclude null from length
048                   jz   exit                ; no printing if string is empty
049
050                   mov  cx, ax              ; load string length in cx
051                   mov  ax, 0xb800
052                   mov  es, ax              ; point es to video base
053                   mov  al, 80              ; load al with columns per row
054                   mul  byte [bp+8]         ; multiply with y position
055                   add  ax, [bp+10]         ; add x position
056                   shl  ax, 1               ; turn into byte offset
057                   mov  di,ax               ; point di to required location
058                   mov  si, [bp+4]          ; point si to string
059                   mov  ah, [bp+6]          ; load attribute in ah
060
061                   cld                      ; auto increment mode
062     nextchar:     lodsb                    ; load next char in al
063                   stosw                    ; print char/attribute pair
064                   loop nextchar            ; repeat for the whole string
065
066     exit:         pop  di
067                   pop  si
068                   pop  cx
069                   pop  ax
070                   pop  es
071                   pop  bp
072                   ret  8
073
074     start:        call clrscr              ; call the clrscr subroutine
075
076                   mov  ax, 30
077                   push ax                  ; push x position
078                   mov  ax, 20
079                   push ax                  ; push y position
080                   mov  ax, 1               ; blue on black attribute
081                   push ax                  ; push attribute
082                   mov  ax, message
083                   push ax                  ; push address of message
084                   call printstr            ; call the printstr subroutine
085
086                   mov  ax, 0x4c00          ; terminate program
                      int  0x21
```

039-040 Another way to load a segment register is to use a combination of push and pop. The processor doesn't match pushes and pops. ES is equalized to DS in this pair of instructions.

Inside the debugger observe the working of the code for length calculation after SCASB has completed its operation.

LES and LDS Instructions

Since the string instructions need their source and destination in the form of a segment offset pair, there are two special instructions that load a segment register and a general purpose register from two consecutive memory locations. LES loads ES while LDS loads DS. Both these instructions have two parameters, one is the general purpose register to be loaded and the other is the memory location from which to load these registers. The major application of these instructions is when a subroutine receives a segment offset pair as an argument and the pair is to be loaded in a segment and an offset register. According to Intel rules of significance the word at higher address is loaded in the segment register while the word at lower address is loaded in the offset register. As parameters segment should be pushed first so that it ends up at a higher address and the offset should be pushed afterwards. When loading the lower address will be given. For example “lds si, [bp+4]” will load SI from BP+4 and DS from BP+6.

7.5. LES AND LDS EXAMPLE

We modify the string length calculation subroutine to take the segment and offset of the string and use the LES instruction to load that segment offset pair in ES and DI.

Example 7.4

```
; hello world printing with length calculation subroutine
[org 0x0100]
              jmp  start

message:      db   'hello world', 0    ; null terminated string

;;;;; COPY LINES 005-024 FROM EXAMPLE 7.1 (clrscr) ;;;;;

; subroutine to calculate the length of a string
; takes the segment and offset of a string as parameters
strlen:       push bp
              mov  bp,sp
              push es
              push cx
              push di

              les  di, [bp+4]         ; point es:di to string
              mov  cx, 0xffff         ; load maximum number in cx
              xor  al, al             ; load a zero in al
              repne scasb             ; find zero in the string
              mov  ax, 0xffff         ; load maximum number in ax
              sub  ax, cx             ; find change in cx
              dec  ax                 ; exclude null from length

              pop  di
              pop  cx
              pop  es
              pop  bp
              ret  4

; subroutine to print a string
; takes the x position, y position, attribute, and address of a null
; terminated string as parameters
printstr:     push bp
              mov  bp, sp
              push es
              push ax
              push cx
              push si
              push di

              push ds                 ; push segment of string
              mov  ax, [bp+4]
              push ax                 ; push offset of string
              call strlen             ; calculate string length
```

```
065                 cmp  ax, 0              ; is the string empty
066                 jz   exit               ; no printing if string is empty
067                 mov  cx, ax             ; save length in cx
068
069                 mov  ax, 0xb800
070                 mov  es, ax             ; point es to video base
071                 mov  al, 80             ; load al with columns per row
072                 mul  byte [bp+8]        ; multiply with y position
073                 add  ax, [bp+10]        ; add x position
074                 shl  ax, 1              ; turn into byte offset
075                 mov  di,ax              ; point di to required location
076                 mov  si, [bp+4]         ; point si to string
077                 mov  ah, [bp+6]         ; load attribute in ah
078
079                 cld                     ; auto increment mode
080   nextchar:     lodsb                   ; load next char in al
081                 stosw                   ; print char/attribute pair
082                 loop nextchar           ; repeat for the whole string
083
084   exit:         pop  di
085                 pop  si
086                 pop  cx
087                 pop  ax
088                 pop  es
089                 pop  bp
090                 ret  8
091
092   start:        call clrscr             ; call the clrscr subroutine
093
094                 mov  ax, 30
095                 push ax                 ; push x position
096                 mov  ax, 20
097                 push ax                 ; push y position
098                 mov  ax, 0x71           ; blue on white attribute
099                 push ax                 ; push attribute
100                 mov  ax, message
101                 push ax                 ; push address of message
102                 call printstr           ; call the printstr subroutine
103
104                 mov  ax, 0x4c00         ; terminate program
105                 int  0x21
```

036 The LES instruction is used to load the DI register from BP+4 and the ES register from BP+6.

065 The convention to return a value from a subroutine is to use the AX register. That is why AX is not saved and restored in the subroutine.

Inside the debugger observe that the segment register is pushed followed by the offset. The higher address FFE6 contains the segment and the lower address FFE4 contains the offset. This is because we have a decrementing stack. Then observe the loading of ES and DI from the stack.

7.6. MOVS EXAMPLE – SCREEN SCROLLING

MOVS has the two forms MOVSB and MOVSW. REP allows the instruction to be repeated CX times allowing blocks of memory to be copied. We will perform this copy of the video screen.

Scrolling is the process when all the lines on the screen move one or more lines towards the top of towards the bottom and the new line that appears on the top or the bottom is cleared. Scrolling is a process on which string movement is naturally applicable. REP with MOVS will utilize the full processor power to do the scrolling in minimum time.

In this example we want to scroll a variable number of lines given as argument. Therefore we have to calculate the source address, which is 160 times the number of lines to clear. The destination address is 0, which is the top left of the screen. The lines that scroll up are discarded so the source pointer is placed after them. An equal number of lines at the bottom are cleared. These lines have actually been copied above.

Example 7.5

```
; scroll up the screen
[org 0x0100]
              jmp  start

; subroutine to scroll up the screen
; take the number of lines to scroll as parameter
scrollup:     push bp
              mov  bp,sp
              push ax
              push cx
              push si
              push di
              push es
              push ds

              mov  ax, 80              ; load chars per row in ax
              mul  byte [bp+4]         ; calculate source position
              mov  si, ax              ; load source position in si
              push si                  ; save position for later use
              shl  si, 1               ; convert to byte offset
              mov  cx, 2000            ; number of screen locations
              sub  cx, ax              ; count of words to move
              mov  ax, 0xb800
              mov  es, ax              ; point es to video base
              mov  ds, ax              ; point ds to video base
              xor  di, di              ; point di to top left column
              cld                      ; set auto increment mode
              rep  movsw               ; scroll up
              mov  ax, 0x0720          ; space in normal attribute
              pop  cx                  ; count of positions to clear
              rep  stosw               ; clear the scrolled space

              pop  ds
              pop  es
              pop  di
              pop  si
              pop  cx
              pop  ax
              pop  bp
              ret  2

start:        mov  ax,5
              push ax                  ; push number of lines to scroll
              call scrollup            ; call the scroll up subroutine

              mov  ax, 0x4c00          ; terminate program
              int  0x21
```

The beauty of this example is that the two memory blocks are overlapping. If the source and destination in the above algorithm are swapped in an expectation to scroll down the result is strange. For example if 5 lines were to scroll down, the top five lines of the screen are repeated on the whole screen. This is where the use of the direction flag comes in.

When the source is five lines below the destination, the first five lines are copied on the first five lines of the destination. However the next five lines to be copied from the source have been destroyed in the process; because they were the same as the first five lines of the destination. The same is the problem with every set of five lines as the source is destroyed during the previous copy. In this situation we must go from bottom of the screen towards the top. Now the last five lines are copied to the last five lines of the destination. The next five lines are copied to next five lines of the destination destroying the last five lines of source; but now these lines are no longer needed and have been previously copied. Therefore the copy will be appropriately done in this case.

We give an example of scrolling down with this consideration. Now we have to calculate the end of the block instead of the start.

Example 7.6

```
; scroll down the screen
[org 0x0100]
              jmp  start

; subroutine to scrolls down the screen
; take the number of lines to scroll as parameter
scrolldown:   push bp
              mov  bp,sp
              push ax
              push cx
              push si
              push di
              push es
              push ds

              mov  ax, 80             ; load chars per row in ax
              mul  byte [bp+4]        ; calculate source position
              push ax                 ; save position for later use
              shl  ax, 1              ; convert to byte offset
              mov  si, 3998           ; last location on the screen
              sub  si, ax             ; load source position in si
              mov  cx, 2000           ; number of screen locations
              sub  cx, ax             ; count of words to move
              mov  ax, 0xb800
              mov  es, ax             ; point es to video base
              mov  ds, ax             ; point ds to video base
              mov  di, 3998           ; point di to lower right column
              std                     ; set auto decrement mode
              rep  movsw              ; scroll up
              mov  ax, 0x0720         ; space in normal attribute
              pop  cx                 ; count of positions to clear
              rep  stosw              ; clear the scrolled space

              pop  ds
              pop  es
              pop  di
              pop  si
              pop  cx
              pop  ax
              pop  bp
              ret  2

start:        mov  ax,5
              push ax                 ; push number of lines to scroll
              call scrolldown         ; call scroll down subroutine

              mov  ax, 0x4c00         ; terminate program
              int  0x21
```

7.7. CMPS EXAMPLE – STRING COMPARISON

For the last string instruction, we take string comparison as an example. The subroutine will take two segment offset pairs containing the address of the two null terminated strings. The subroutine will return 0 if the strings are different and 1 if they are same. The AX register will be used to hold the return value.

Example 7.7

```
; comparing null terminated strings
[org 0x0100]
              jmp  start

msg1:         db   'hello world', 0
msg2:         db   'hello WORLD', 0
msg3:         db   'hello world', 0

;;;;; COPY LINES 028-050 FROM EXAMPLE 7.4 (strlen) ;;;;;

; subroutine to compare two strings
; takes segment and offset pairs of two strings to compare
```

```
035 ; returns 1 in ax if they match and 0 other wise
036 strcmp:       push bp
037               mov  bp,sp
038               push cx
039               push si
040               push di
041               push es
042               push ds
043
044               lds  si, [bp+4]        ; point ds:si to first string
045               les  di, [bp+8]        ; point es:di to second string
046
047               push ds                ; push segment of first string
048               push si                ; push offset of first string
049               call strlen            ; calculate string length
050               mov  cx, ax            ; save length in cx
051
052               push es                ; push segment of second string
053               push di                ; push offset of second string
054               call strlen            ; calculate string lenth
055               cmp  cx, ax            ; compare length of both strings
056               jne  exitfalse         ; return 0 if they are unequal
057
058               mov  ax, 1             ; store 1 in ax to be returned
059               repe cmpsb             ; compare both strings
060               jcxz exitsimple        ; are they successfully compared
061
062 exitfalse:    mov  ax, 0             ; store 0 to mark unequal
063
064 exitsimple:   pop  ds
065               pop  es
066               pop  di
067               pop  si
068               pop  cx
069               pop  bp
070               ret  8
071
072 start:        push ds                ; push segment of first string
073               mov  ax, msg1
074               push ax                ; push offset of first string
075               push ds                ; push segment of second string
076               mov  ax, msg2
077               push ax                ; push offset of second string
078               call strcmp            ; call strcmp subroutine
079
080               push ds                ; push segment of first string
081               mov  ax, msg1
082               push ax                ; push offset of first string
083               push ds                ; push segment of third string
084               mov  ax, msg3
085               push ax                ; push offset of third string
086               call strcmp            ; call strcmp subroutine
087
088               mov  ax, 0x4c00        ; terminate program
089               int  0x21
```

005-007	Three strings are declared out of which two are equal and one is different.
044-045	LDS and LES are used to load the pointers to the two strings in DS:SI and ES:DI.
070	Since there are 4 parameters to the subroutine “ret 8” is used.

Inside the debugger we observe that REPE is shown as REP. This is because REP and REPE are represented with the same prefix byte. When used with STOS, LODS, and MOVS it functions as REP and when used with SCAS and CMPS it functions as REPE.

EXERCISES

1. Write code to find the byte in AL in the whole megabyte of memory such that each memory location is compared to AL only once.

2. Write a far procedure to reverse an array of 64k words such that the first element becomes the last and the last becomes the first and so on. For example if the first word contained 0102h, this value is swapped with the last word. The next word is swapped with the second last word and so on. The routine will be passed two parameters through the stack; the segment and offset of the first element of the array.
3. Write a near procedure to copy a given area on the screen at the center of the screen without using a temporary array. The routine will be passed top, left, bottom, and right in that order through the stack. The parameters passed will always be within range the height will be odd and the width will be even so that it can be exactly centered.
4. Write code to find two segments in the whole memory that are exactly the same. In other words find two distinct values which if loaded in ES and DS then for every value of SI [DS:SI]=[ES:SI].
5. Write a function writechar that takes two parameters. The first parameter is the character to write and the second is the address of a memory area containing top, left, bottom, right, current row, current column, normal attribute, and cursor attribute in 8 consecutive bytes. These define a virtual window on the screen.
   The function writes the passed character at (current row, current column) using the normal attribute. It then increments current column, If current column goes outside the window, it makes it zero and increments current row. If current row gets out of window, it scrolls the window one line up, and blanks out the new line in the window. In the end, it sets the attribute of the new (current row, current column) to cursor attribute.
6. Write a function “strcpy” that takes the address of two parameters via stack, the one pushed first is source and the second is the destination. The function should copy the source on the destination including the null character assuming that sufficient space is reserved starting at destination.