A LECTURE NOTE

ON

ASSEMBLY LANGUAGE PROGRAMMING (PART 1)

(CSC 303)

COURSE LECTURER:

DR. ONASHOGA S.A (MRS.)

COURSE OUTLINE

SECTION 1

1. **Introduction to Programming languages**
    - Machine Language
    - Low-Level Language
    - High-Level Language
2. **Data Representation & Numbering Systems**
    - Binary Numbering Systems
    - Octal Numbering Systems
    - Decimal Numbering Systems
    - Hexadecimal Numbering Systems
3. **Types of encoding**
    - American Standard Code for Information Interchange (ASCII)
    - Binary Coded Decimal (BCD)
    - Extended Binary Coded Decimal Interchange Code (EBCDIC)
4. **Mode of data representation**
    - Integer Representation
    - Floating Point Representation
5. **Computer instruction set**
    - Reduced Instruction Set Computer (RISC)
    - Complex Instruction Set Computer (CISC)

SECTION TWO

6. **Registers**
    - General Purpose Registers
    - Segment Registers
    - Special Purpose Registers

7. **80x86 instruction sets and Modes of addressing.**

- Addressing modes with Register operands
- Addressing modes with constants
- Addressing modes with memory operands
- Addressing mode with stack memory

8. **Instruction Sets**

- The 80x86 instruction sets
- The control transfer instruction
- The standard input routines
- The standard output routines
- Macros

9. **Assembly Language Programs**

- An overview of Assembly Language program
- The linker
- Examples of common Assemblers
- A simple Hello World Program using FASM
- A simple Hello World Program using NASMS

10. **Job Control Language**

- Introduction
- Basic syntax of JCL statements

- Types of JCL statements
- The JOB statement
- The EXEC statement
- The DD statement

CHAPTER ONE

1.0 INTRODUCTION TO PROGRAMMING LANGUAGES

Programmers write instructions in various programming languages, some directly understandable by computers and others requiring intermediate translation steps. Hundreds of computer languages are in use today. These can be divided into three general types:

a. Machine Language
b. Low Level Language
c. High level Language

1.1 MACHINE LANGUAGE

Any computer can directly understand its own machine language. Machine language is the "natural language" of a computer and such is defined by its hardware design. Machine languages generally consist of strings of numbers (ultimately reduced to 1s and 0s) that instruct computers to perform their most elementary operations one at a time. Machine languages are machine dependent (i.e a particular machine language can be used on only one type of computer). Such languages are cumbersome for humans, as illustrated by the following section of an early machine language program that adds overtime pay to base pay and stores the result in gross pay.

+1300042774

+1400593419

+1200274027

Advantages of Machine Language

i. It uses computer storage more efficiently
ii. It takes less time to process in a computer than any other programming language

Disadvantages of Machine Language

i. It is time consuming

ii. It is very tedious to write
iii. It is subject to human error
iv. It is expensive in program preparation and debugging stages

1.2 LOW LEVEL LANGUAGE

Machine Language were simply too slow and tedious for most programmers. Instead of using strings of numbers that computers could directly understand, programmers began using English like abbreviations to represent elementary operations. These abbreviations form the basis of **Low Level Language.** In low level language, instructions are coded using mnemonics. E.g. DIV, ADD, SUB, MOV. Assembly language is an example of a low level language.

An **assembly language** is a low-level language for programming computers. It implements a symbolic representation of the numeric machine codes and other constants needed to program a particular CPU architecture. This representation is usually defined by the hardware manufacturer, and is based on abbreviations (called mnemonics) that help the programmer remember individual instructions, registers, etc. An assembly language is thus specific to a certain physical or virtual computer architecture (as opposed to most high-level languages, which are usually portable).

A utility program called an **assembler** is used to translate assembly language statements into the target computer's machine code. The assembler performs a more or less isomorphic translation (a one-to-one mapping) from mnemonic statements into machine instructions and data. (This is in contrast with high-level languages, in which a single statement generally results in many machine instructions.)

Today, assembly language is used primarily for direct hardware manipulation, access to specialized processor instructions, or to address critical performance issues. The following section of an assembly language program also adds overtime to base pay and stores the result in gross pay:

```
Load basepay
Add overpay
Store grosspay
```

Advantages of Low Level Language

i. It is more efficient than machine language
ii. Symbols make it easier to use than machine language
iii. It may be useful for security reasons

Disadvantages of Low Level Language

i. It is defined for a particular processor
ii. Assemblers are difficult to get
iii. Although, low level language codes are clearer to humans, they are incomprehensible to computers until they are translated to machine language.

1.3 HIGH LEVEL LANGUAGE: Computers usage increased rapidly with the advent of assembly languages, but programmers still had to use many instructions to accomplish even the simplest tasks. To speed up the programming process, **high level language** were developed in which simple statements could be written to accomplish substantial tasks. Translator programs called **compilers** convert high level language programs into machine language. High level language allows programmers to write instructions that look almost like everyday English and contain commonly used mathematical notations. A payroll program written in high level language might contain a statement such as

grossPay=basePay + overTimePay

Advantages of High Level Language

i. Compilers are easy to get
ii. It is easier to use than any other programming language
iii. It is easier to understand compared to any other programming language

Disadvantages of High Level Language

i. It takes more time to process in a computer than any other programming language

CHAPTER TWO

1.0 DATA REPRESENTATION AND NUMBERING SYSTEMS

Most modern computer systems do not represent numeric values using the decimal system. Instead, they use a binary or two's complement numbering system. To understand the limitations of computer arithmetic, one must understand how computers represent numbers.

1.1 THE BINARY NUMBERING SYSTEM

Most modern computer systems (including the IBM PC) operate using binary logic. The computer represents values using two voltage levels (usually 0v and +5v). With two such levels we can represent exactly two different values. These could be any two different values, but by convention we use the values zero and one. These two values, coincidentally, correspond to the two digits used by the binary numbering system. Since there is a correspondence between the logic levels used by the 80x86 and the two digits used in the binary numbering system, it should come as no surprise that the IBM PC employs the binary numbering system.

The binary numbering system works just like the decimal numbering system, with two exceptions: binary only allows the digits 0 and 1 (rather than 0-9), and binary uses powers of two rather than powers of ten. Therefore, it is very easy to convert a binary number to decimal. For each "1" in the binary string, add in 2**n where "n" is the zero-based position of the binary digit. For example, the binary value 11001010 represents:

1*2**7 + 1*2**6 + 0*2**5 + 0*2**4 + 1*2**3 + 0*2**2 + 1*2**1 + 0*2**0
=128 + 64 + 8 + 2
=202 (base 10)

To convert decimal to binary is slightly more difficult. You must find those powers of two which, when added together, produce the decimal result. The easiest method is to work from the a large power of two down to 2**0. Consider the decimal value 1359:

- 2**10=1024, 2**11=2048. So 1024 is the largest power of two less than 1359. Subtract 1024 from 1359 and begin the binary value on the left with a "1" digit. Binary = "1", Decimal result is 1359 - 1024 = 335.

- The next lower power of two (2**9= 512) is greater than the result from above, so add a "0" to the end of the binary string. Binary = "10", Decimal result is still 335.

- The next lower power of two is 256 (2**8). Subtract this from 335 and add a "1" digit to the end of the binary number. Binary = "101", Decimal result is 79.

- 128 (2**7) is greater than 79, so tack a "0" to the end of the binary string. Binary = "1010", Decimal result remains 79.

- The next lower power of two (2**6 = 64) is less than79, so subtract 64 and append a "1" to the end of the binary string. Binary = "10101", Decimal result is 15.

- 15 is less than the next power of two (2**5 = 32) so simply add a "0" to the end of the binary string. Binary = "101010", Decimal result is still 15.

- 16 (2**4) is greater than the remainder so far, so append a "0" to the end of the binary string. Binary = "1010100", Decimal result is 15.

- 2**3(eight) is less than 15, so stick another "1" digit on the end of the binary string. Binary = "10101001", Decimal result is 7.

- 2**2 is less than seven, so subtract four from seven and append another one to the binary string. Binary = "101010011", decimal result is 3.

- 2**1 is less than three, so append a one to the end of the binary string and subtract two from the decimal value. Binary = "1010100111", Decimal result is now 1.

- Finally, the decimal result is one, which is2**0, so add a final "1" to the end of the binary string. The final binary result is "10101001111"

Binary numbers, although they have little importance in high level languages, appear everywhere in assembly language programs

1.8 THE OCTAL NUMBERING SYSTEM

Octal numbers are numbers to base 8. The primary advantage of the octal number system is the ease with which conversion can be made between binary and decimal numbers. Octal is often used as shorthand for binary numbers because of its easy conversion. The octal numbering system is shown below;

Decimal Number	Octal Equivalence
0	001
1	001
2	010
3	011
4	100
5	101
6	110
7	111

1.3 THE DECIMAL NUMBERING SYSTEM

The decimal (base 10) numbering system has been used for so long that people take it for granted. When you see a number like “123”, you don’t think about the value 123, rather, you generate a mental image of how many items this value represents in reality, however, the number 123 represents”

$1*10^2 + 2*10^1 + 3*10^0$ or 100+20+3

Each digit appearing to the left of the decimal point represents a value between zero and nine times an increasing power of ten. Digits appearing to the right of the decimal point represent a value between zero and nine times an increasing negative power of ten.

e.g. 123.456 means

$1*10^2 + 2*10^1 + 3*10^0 + 4*10^{-1} + 5*10^{-2} + 6*10^{-3}$

or 100 + 20 +3 +0.4 + 0.05 +0.006

1.4 THE HEXADECIMAL NUMBERING SYSTEM

A big problem with the binary system is verbosity. To represent the value 202 (decimal) requires eight binary digits. The decimal version requires only three decimal digits and, thus, represents numbers much more compactly than does the binary numbering system. This fact was not lost on the engineers who designed binary computer systems. When dealing with large values, binary numbers quickly become too unwieldy. Unfortunately, the computer thinks in binary, so most of the time it is convenient to use the binary numbering system. Although we can convert between decimal and binary, the conversion is not a trivial task. The hexadecimal (base 16) numbering system solves these problems. Hexadecimal numbers offer the two features we're looking for: they're very compact, and it's simple to convert them to binary and vice versa. Because of this, most binary computer systems today use the hexadecimal numbering system. Since the radix (base) of a hexadecimal number is 16, each hexadecimal digit to the left of the hexadecimal point represents some value times a successive power of 16. For example, the number 1234 (hexadecimal) is equal to:

1 * 16**3 + 2 * 16**2 + 3 * 16**1 + 4 * 16**0 or
4096 + 512 + 48 + 4 = 4660 (decimal).

Each hexadecimal digit can represent one of sixteen values between 0 and 15. Since there are only ten decimal digits, we need to invent six additional digits to represent the values in the range 10 through 15. Rather than create new symbols for these digits, we'll use the letters A through F. The following are all examples of valid hexadecimal numbers:

1234 DEAD BEEF 0AFB FEED DEAF

Since we'll often need to enter hexadecimal numbers into the computer system, we'll need a different mechanism for representing hexadecimal numbers. After all, on most computer systems you cannot enter a subscript to denote the radix of the associated value. We'll adopt the following conventions:

- All numeric values (regardless of their radix) begin with a decimal digit.
- All hexadecimal values end with the letter "h", e.g., 123A4h.
- All binary values end with the letter "b".
- Decimal numbers may have a "t" or "d" suffix.

Examples of valid hexadecimal numbers:

1234h 0DEADh 0BEEFh 0AFBh 0FEEDh 0DEAFh

As you can see, hexadecimal numbers are compact and easy to read. In addition, you can easily convert between hexadecimal and binary. Consider the following table:

Binary/Hex Conversion	
Binary	**Hexadecimal**
0000	**0**
0001	**1**
0010	**2**
0011	**3**
0100	**4**
0101	**5**
0110	**6**
0111	**7**

1000	**8**
1001	**9**
1010	**A**
1011	**B**
1100	**C**
1101	**D**
1110	**E**
1111	**F**

This table provides all the information you'll ever need to convert any hexadecimal number into a binary number or vice versa.

To convert a hexadecimal number into a binary number, simply substitute the corresponding four bits for each hexadecimal digit in the number. For example, to convert 0ABCDh into a binary value, simply convert each hexadecimal digit according to the table above:

0 A B C D Hexadecimal

0000 1010 1011 1100 1101 Binary

To convert a binary number into hexadecimal format is almost as easy. The first step is to pad the binary number with zeros to make sure that there is a multiple of four bits in the number. For example, given the binary number 1011001010, the first step would be to add two bits to the left of the number so that it contains 12 bits. The converted binary value is 001011001010. The next step is to separate the binary value into groups of four bits, e.g., 0010 1100 1010. Finally, look up these binary values in the table above and substitute the appropriate hexadecimal digits, e.g., 2CA. Contrast this with the difficulty of conversion between decimal and binary or decimal and hexadecimal!

Since converting between hexadecimal and binary is an operation you will need to perform over and over again, you should take a few minutes and memorize the table above. Even if you have a

calculator that will do the conversion for you, you'll find manual conversion to be a lot faster and more convenient when converting between binary and hex.

A comparison of the afore mentioned numbering systems is shown below;

binary	octal	decimal	Hexadecimal
0	0	0	0
1	1	1	1
10	2	2	2
11	3	3	3
100	4	4	4
101	5	5	5
110	6	6	6
111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F

CHAPTER THREE

3.0 TYPES OF ENCODING

When numbers, letters and words are represented by a special group of symbols, this is called "**Encoding"** and the group of symbol encoded is called a "**code".** Any decimal number can be represented by an equivalent binary number. When a decimal number is represented by its equivalent binary number, it is called "**straight binary coding**".

Basically, there are three methods of encoding and they are;

- American Standard Code for Information Interchange (ASCII)
- Binary Coded Decimal (BCD)
- Extended Binary Coded Decimal Interchange Code(EBCDIC)

3.1 ASCII CODING SYSTEM

In addition to numeric data, a computer must be able to handle non- numeric information. In order words, a computer should recognize codes that represents letters of the alphabets, punctuation marks, other special characters as well as numbers. These codes are called ***alphanumeric codes.*** The most widely used alphanumeric code is ASCII code (American Standard Code for Information Interchange). ASCII is used in most micro computers and mini computers and in many main frames. The ASCII code is a seven bit code, thus it has 2^7=128 possible code groups. In the 7 bits code, the first 3 bits represent the zone bits and the last 4 bits represent the numeric bits.

Despite some major shortcomings, ASCII data is *the* standard for data interchange across computer systems and programs. Most programs can accept ASCII data; likewise most programs can produce ASCII data. Since you will be dealing with ASCII characters in assembly language, it would be wise to study the layout of the character set and memorize a few key ASCII codes (e.g., "0", "A", "a", etc.).

The table below shows some commonly used ASCII codes

ZONE BITS					**NUMERIC BITS**			
011	100	101	110	111	8	4	2	1
0		P		p	0	0	0	0
1	A	Q	a	q	0	0	0	1
2	B	R	b	r	0	0	1	0
3	C	S	c	s	0	0	1	1
4	D	T	d	t	0	1	0	0
5	E	U	e	u	0	1	0	1
6	F	V	f	v	0	1	1	0
7	G	W	g	w	0	1	1	1
8	H	X	h	x	1	0	0	0
9	I	Y	i	y	1	0	0	1
	J	Z	j	z	1	0	1	0
	K		k		1	0	1	1

	L		l		1	1	0	0
	M		m		1	1	0	1
	N		n		1	1	1	0
	O		o		1	1	1	1

A summary of ASCII table is shown below;

Characters	Zonebits	Numeric bits
0-9	011	0000-1001
A-O	100	0001-1111
P-Z	101	0000-1010
a-o	110	0001-1111
p-z	111	0000-1010

Examples

1. Represent Bez in binary format

Since Bez contains an alphabets, the ASCII representation is suitable for this conversion

B- 100 0010

e- 110 0101

z- 111 1010

Answer: 100001011001011111010

2. Convert 1001000 1000101 1001100 1010000 to ASCII

1001000- H

1000101- E

1001100- L

1010000- P

Answer: HELP

3. Using ASCII representation, convert ***UNIVERSITY*** to binary

U- 1010101, N- 1001110, I- 1001001, V- 1000110, E- 1000101, R- 1010010, S- 1010011, I- 1001001,

T-1010100, Y- 1011001

ANSWER: 1010101100111010010011000110100010110100101010011100 1001

3.2 BINARY CODED DECIMAL

If each digit of a decimal number is represented by binary equivalence, this produces a code called Binary Coded Decimal. Since a decimal digit can be as large as 9, 4 bits are required to code each digit in the decimal number. E.g.

$874_{10} = 100001110100_2$

$943_{10} = 100101000011_2$

Only the four bits binary numbers from 0000 through 1001 are used for binary coded decimal. The BCD code does not use the numbers 10, 11, 12, 13, 14, 15. In other words, 10 of the 16 possible 4 bits binary codes are used. If any of these forbidden 4 bits number ever occurs in a machine using the BCD, it

is usually an indication that an error has occurred.

Comparison of BCD and Binary

It is important to realize that BCD is not another number system like binary, octal, hexadecimal and decimal. It is in fact the decimal system with each digit encoded in its binary equivalence. It is also important to understand that a BCD number is not the same as binary number.

A straight binary code takes the complete decimal number and represents it in binary while the BCD code converts each decimal digit to binary individually.

e.g.

137_{10} to straight binary coding is 10001001

137_{10} to BCD is 000100110111

The main advantage of BCD is the relative ease of converting to and from decimal. This ease of conversion is especially important from a hardware standpoint because in a digital system, it is the logic circuit that performs conversion to and from decimal.

BCD is used in digital machines whenever decimal information is either applied as input or displayed as output. e.g. digital voltmeter, frequency counters make use of BCD. Electronic calculators also make use of BCD because the input numbers are entered in decimal through the keyboard and the output is displayed in decimal.

BCD is not often used in modern high speed digital system for good 2 good reasons;

1. As it was already pointed out, the BCD code for a given decimal number requires more bits that the straight binary code and it is therefore less efficient. This is important in digital computers because the number of places in memory where the bits can be stored is limited.
2. The arithmetic processes for numbers represented in BCD code are more complicated than straight binary and thus requires more complex circuitry which contributes to a decrease in the speed at which arithmetic operations take place.

3.3 EBCDIC CODING SYSTEM

EBCDIC is an acronym for ***Extended Binary Coded Decimal Interchange Code***. IBM developed this code for use on its computers. In EBCDIC, eight bits are used to represent each character i.e 256 characters can be represented. IBM minicomputers and mainframe computers use the EBCDIC. The eight bits can as well be divided into two. The zonebits and the numeric bits each is represented by 4bits.

Zonebits				Numeric bits			
1111	1100	1101	1100	8	4	2	1
0				0	0	0	0
1	A	J	S	0	0	0	1
2	B	K	T	0	0	1	0
3	C	L	U	0	0	1	1
4	D	M	V	0	1	0	0
5	E	N	W	0	1	0	1
6	F	O	X	0	1	1	0
7	G	P	Y	0	1	1	1
8	H	Q	Z	1	0	0	0
9	I	R		1	0	0	1

Example: Convert **HELP** to binary using EBCDIC coding system.

Solution:

H- 11001000, E- 11000101, L- 11010011, P- 11010111

Answer

11001000110001011101001111010111

CHAPTER FOUR

4.0 MODES OF DATA REPRESENTATION

Most data structures are abstract structures and are implemented by the programmer with a series of assembly language instructions. Many cardinal data types (bits, bit strings, bit slices, binary integers, binary floating point numbers, binary encoded decimals, binary addresses, characters, etc.) are implemented directly in hardware for at least parts of the instruction set. Some processors also implement some data structures in hardware for some instructions — for example, most processors have a few instructions for directly manipulating character strings.

An assembly language programmer has to know how the hardware implements these cardinal data types. Some examples: Two basic issues are bit ordering (big endian or little endian) and number of bits (or bytes). The assembly language programmer must also pay attention to word length and optimum (or required) addressing boundaries. Composite data types will also include details of hardware implementation, such as how many bits of mantissa, characteristic, and sign, as well as their order.

4.1 INTEGER REPRESENTATION

Sign-magnitude is the simplest method for representing signed binary numbers. One bit (by universal convention, the highest order or leftmost bit) is the sign bit, indicating positive or

negative, and the remaining bits are the absolute value of the binary integer. Sign-magnitude is simple for representing binary numbers, but has the drawbacks of two different zeros and much more complicates (and therefore, slower) hardware for performing addition, subtraction, and any binary integer operations other than complement (which only requires a sign bit change).

In sign magnitude, the sign bit for positive is represented by 0 and the sign bit for negative is represented by 1.

Examples:

1. Convert +52 to binary using an 8 bits machine

Answer: The binary equivalence of 52 is 110100 but 0 is used to represent positive magnitude, hence 0 is added to the front of this binary equivalence. This makes a total of 7bits, since we are working on an eight bit machine, we have to pad the numbers with 0 so as to make it a total of 8bits. Thus the binary equivalence 0f 52 is 00110100.

2. Convert -52 to binary using an 8 bits machine

Answer: The binary equivalence of 52 is 110100 but 1 is used to represent positive magnitude, hence 1 is added to the front of this binary equivalence. This makes a total of 7bits, since we are working on an eight bit machine, we have to pad the numbers with 0 so as to make it a total of 8bits. In this case, the sign bit has to come first and the padded 0 follows. Thus the binary equivalence 0f -52 is 10110100.

Exercise:

a. Convert +47 to binary on an 8 bits machine
b. Convert -17 to binary on an 8 bits machine
c. Convert -567 on a 16 bits machine

In **one's complement** representation, positive numbers are represented in the "normal" manner (same as unsigned integers with a zero sign bit), while negative numbers are represented by complementing all of the bits of the absolute value of the number. Numbers are negated by complementing all bits. Addition of two integers is peformed by treating the numbers as unsigned integers (ignoring sign bit), with a carry out of the leftmost bit position being added to the least significant bit (technically, the carry bit is always added to the least significant bit, but when it is zero, the add has no effect). The ripple effect of adding the carry bit can almost double the time to do an addition. And there are still two zeros, a positive zero (all zero bits) and a negative zero (all one bits).

The I's complement form of any binary number is simply by changing each 0 in the number to a 1 and vice versa.

Examples

1. Find the 1's complement of -7

Answer: -7 in the actual representation without considering the machine bit is 1111. To change this to 1's complement, the sign bit has to be retained and other bits have to be inverted. Thus, the answer is: 1000. 1 denotes the sign bit.

2. Find the1's complement of -7.25

The actual magnitude representation of -7.25 is 1111.01 but retaining the sign bits and inverting the other bits gives: 1000.10

Exercises

1. Find the one's complement of -47
2. Find the one's complement of -467 and convert the answer to hexcadecimal.

In **two's complement** representation, positive numbers are represented in the "normal" manner (same as unsigned integers with a zero sign bit), while negative numbers are represented

by complementing all of the bits of the absolute value of the number and adding one. Negation of a negative number in two's complement representation is accomplished by complementing all of the bits and adding one. Addition is performed by adding the two numbers as unsigned integers and ignoring the carry. Two's complement has the further advantage that there is only one zero (all zero bits). Two's complement representation does result in one more negative number (all one bits) than positive numbers.

Two's complement is used in just about every binary computer ever made. Most processors have one more negative number than positive numbers. Some processors use the "extra" neagtive number (all one bits) as a special indicator, depicting invalid results, not a number (NaN), or other special codes.

2's complement is used to represent negative numbers because it allows us to perform the operation of subtraction by actually performing addition. The 2's complement of a binary number is the addition of 1 to the rightmost bit of its 1's complement equivalence.

Examples

1. Convert -52 to its 2's complement

The 1's complement of -52 is 11001011

To convert this to 2's complement we have

11001011

+ 1

11001100

2. Convert -419 to 2's complement and hence convert the result to hexadecimal

The sign magnitude representation of -419 on a 16 bit machine is 1000000110100011

The I's complement is 1111111001011100

To convert this to 2's complement, we have:

1111111001011100

+ 1

1111111001011101

Dividing the resulting bits into four gives an hexadecimal equivalence of $FE5D_{16}$

In general, if a number $a_1, a_2 \ldots\ldots a_n$ is in base b, then we by an form its b's complement by subtracting each digit of the number from b-1 and adding 1 to the result.

Example: Find the 8's complement of 7245_8

Answer:

7777

-7245

0532

+ 1

0533

Thus the 8's complement of 7245_8 is 0533

ADDITION OF NUMBERS USING 2'S COMPLEMENT

1. Add +9 and +4 for 5 bits machine

= 01001

00100

01101

01101 is equivalent to +13

2. Add +9 and -4 on a 5 bits machine

+9 in its sign magnitude form is 01001

-4 in the sign magnitude form is10100

Its 1's complement equivalence is 11011

Its 2's complement equivalence is 11100 (by adding 1 to its 1's complement)

Thus addition +9 and -4 which is also +9 + (-4)

This gives

01001

+ 11100

100101

Since we are working on a 5bits machine, the last leftmost bit is an off-bit, thus it is neglected. The resulting answer is thus 00101. This is equivalent to +5

3. Add -9 and +4

The 2's complement of -9 is 10111

The sign magnitude of +4 is 00100

This gives;

10111

+ 00100

11011

The sum has a sign bit of 1 indicating a negative number, since the sum is negative, it is in its 2's complement, thus the last 4 bits 1.e 1011 actually represent the 2's complement of the sum. To find the true magnitude of the sum, we 2's complement the sum

11011 to 1's complement is 10100

2's complement is 1

10101

This is equivalent to -5

Exercise:

1. Using the last example's concept add -9 and -4.

The expected answer is 11101

2. Add -9 and +9

The expected answer is 100000, the last leftmost bit is an off bit thus, it is truncated.

In **unsigned** representation, only positive numbers are represented. Instead of the high order bit being interpreted as the sign of the integer, the high order bit is part of the number. An unsigned number has one power of two greater range than a signed number (any representation) of the same number of bits. A comparison of the integer arithmetic forms is shown below;

bit pattern	*sign-mag.*	*one's comp.*	*two's comp*	*unsigned*
000	0	0	0	0
001	1	1	1	1
010	2	2	2	2
011	3	3	3	3
100	-0	-3	-4	4
101	-1	-2	-3	5
110	-2	-1	-2	6
111	-3	-0	-1	7

4.2 FLOATING POINT REPRESENTATIONS

Floating point numbers are the computer equivalent of "scientific notation" or "engineering notation". A floating point number consists of a fraction (binary or decimal) and an exponent (bianry or decimal). Both the fraction and the exponent each have a sign (positive or negative).

In the past, processors tended to have proprietary floating point formats, although with the development of an IEEE standard, most modern processors use the same format. Floating point numbers are almost always binary representations, although a few early processors had (binary coded) decimal representations. Many processors (especially early mainframes and early microprocessors) did not have any hardware support for floating point numbers. Even when commonly available, it was often in an optional processing unit (such as in the IBM 360/370 series) or coprocessor (such as in the Motorola 680x0 and pre-Pentium Intel 80x86 series).

Hardware floating point support usually consists of two sizes, called **single precision** (for the smaller) and **double precision** (for the larger). Usually the double precision format had twice as many bits as the single precision format (hence, the names single and double). Double precision floating point format offers greater range and precision, while single precision floating point format offers better space compaction and faster processing.

F_floating format (single precision floating), DEC VAX, 32 bits, the first bit (high order bit in a register, first bit in memory) is the sign magnitude bit (one=negative, zero=positive or zero), followed by 15 bits of an excess 128 binary exponent, followed by a normalized 24-bit fraction with the redundant most significant fraction bit not represented. Zero is represented by all bits being zero (allowing the use of a longword CLR to set a F_floating number to zero). Exponent values of 1 through 255 indicate true binary exponents of -127 through 127. An exponent value of zero together with a sign of zero indicate a zero value. An exponent value of zero together with a sign bit of one is taken as reserved (which produces a reserved operand fault if used as an operand for a floating point instruction). The magnitude is an approximate range of $.29*10^{-38}$ through $1.7*10^{38}$. The precision of an F_floating datum is approximately one part in 2^{23}, or approximately seven (7) decimal digits).

32 bit floating format (single precision floating), AT&T DSP32C, 32 bits, the first bit (high order bit in a register, first bit in memory) is the sign magnitude bit (one=negative, zero=positive or zero), followed by 23 bits of a normalized two's complement fractional part of the mantissa, followed by an eight bit exponent. The magnitude of the mantissa is always normalized to lie between 1 and 2. The floating point value with exponent equal to zero is reserved to represent the

number zero (the sign and mantissa bits must also be zero; a zero exponent with a nonzero sign and/or mantissa is called a “dirty zero” and is never generated by hardware; if a dirty zero is an operand, it is treated as a zero). The range of nonzero positive floating point numbers is N = [1 * 2^{-127}, [2-2^{-23}] * 2^{127}] inclusive. The range of nonzero negative floating point numbers is N = [-[1 + 2^{-23}] * 2^{-127}, -2 * 2^{127}] inclusive.

40 bit floating format (extended single precision floating), AT&T DSP32C, 40 bits, the first bit (high order bit in a register, first bit in memory) is the sign magnitude bit (one=negative, zero=positive or zero), followed by 31 bits of a normalized two’s complement fractional part of the mantissa, followed by an eight bit exponent. This is an internal format used by the floating point adder, accumulators, and certain DAU units. This format includes an additional eight guard bits to increase accuracy of intermediate results.

D_floating format (double precision floating), DEC VAX, 64 bits, the first bit (high order bit in a register, first bit in memory) is the sign magnitude bit (one=negative, zero=positive or zero), followed by 15 bits of an excess 128 binary exponent, followed by a normalized 48-bit fraction with the redundant most significant fraction bit not represented. Zero is represented by all bits being zero (allowing the use of a quadword CLR to set a D_floating number to zero). Exponent values of 1 through 255 indicate true binary exponents of -127 through 127. An exponent value of zero together with a sign of zero indicate a zero value. An exponent value of zero together with a sign bit of one is taken as reserved (which produces a reserved operand fault if used as an operand for a floating point instruction). The magnitude is an approximate range of .29*10^{-38} through 1.7*10^{38}. The precision of an D_floating datum is approximately one part in 2^{55}, or approximately 16 decimal digits).

CHAPTER FIVE

COMPUTER INSTRUCTION SET

5.0 An **instruction set** is a list of all the instructions, and all their variations, that a processor (or in the case of a virtual machine, an interpreter) can execute.

- Arithmetic such as **add** and **subtract**
- Logic instructions such as **and**, **or**, and **not**
- Data instructions such as **move**, **input**, **output**, **load**, and **store**
- Control flow instructions such as **goto**, **if ... goto**, **call**, and **return**.

An **instruction set**, or **instruction set architecture** (ISA), is the part of the computer architecture related to programming, including the native data types, instructions, registers,

addressing modes, memory architecture, interrupt and exception handling, and external I/O. An ISA includes a specification of the set of opcodes (machine language), the native commands implemented by a particular CPU design.

10.1 REDUCED INSTRUCTION SET

The acronym **RISC** (pronounced *risk*), for **reduced instruction set computing**, represents a CPU design strategy emphasizing the insight that simplified instructions that "do less" may still provide for higher performance if this simplicity can be utilized to make instructions execute very quickly. Many proposals for a "precise" definition have been attempted, and the term is being slowly replaced by the more descriptive **load-store architecture**. Well known RISC families include Alpha, ARC, ARM, AVR, MIPS, PA-RISC, Power Architecture (including PowerPC), SuperH, and SPARC.

Being an old idea, some aspects attributed to the first RISC-*labeled* designs (around 1975) include the observations that the memory restricted compilers of the time were often unable to take advantage of features intended to facilitate coding, and that complex addressing *inherently* takes many cycles to perform. It was argued that such functions would better be performed by sequences of simpler instructions, if this could yield implementations simple enough to cope with really high frequencies, and small enough to leave room for many registers, factoring out slow memory accesses. Uniform, fixed length instructions with arithmetics restricted to registers were chosen to ease instruction pipelining in these simple designs, with special *load-store* instructions accessing memory.

TYPICAL CHARACTERISTICS OF RISC

For any given level of general performance, a RISC chip will typically have far fewer transistors dedicated to the core logic which originally allowed designers to increase the size of the register set and increase internal parallelism.

Other features, which are typically found in RISC architectures are:

- Uniform instruction format, using a single word with the opcode in the same bit positions in every instruction, demanding less decoding;
- Identical general purpose registers, allowing any register to be used in any context, simplifying compiler design (although normally there are separate floating point registers);
- Simple addressing modes. Complex addressing performed via sequences of arithmetic and/or load-store operations;
- Few data types in hardware, some CISCs have byte string instructions, or support complex numbers; this is so far unlikely to be found on a RISC.

RISC designs are also more likely to feature a Harvard memory model, where the instruction stream and the data stream are conceptually separated; this means that modifying the memory where code is held might not have any effect on the instructions executed by the processor (because the CPU has a separate instruction and data cache), at least until a special synchronization instruction is issued. On the upside, this allows both caches to be accessed simultaneously, which can often improve performance.

5.2 COMPLEX INSTRUCTION SET COMPUTER

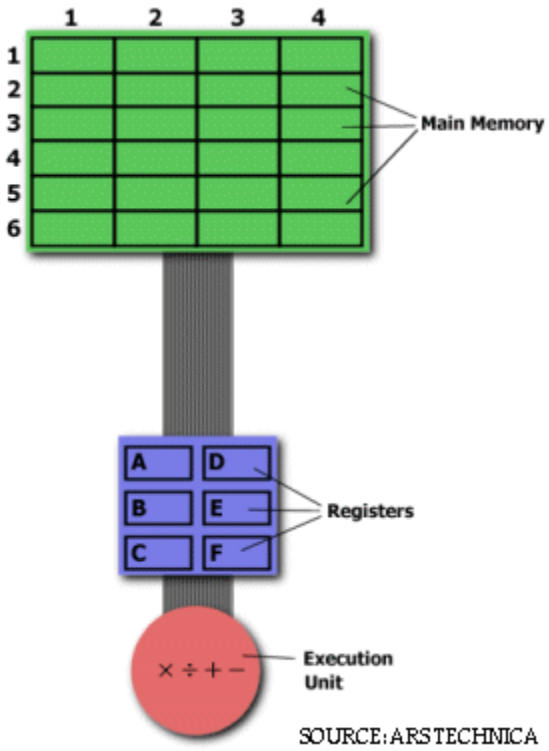
Complex Instruction Set Computers (CISC) have a large instruction set, with hardware support for a wide variety of operations. In scientific, engineering, and mathematical operations with hand coded assembly language (and some business applications with hand coded assembly language), CISC processors usually perform the most work in the shortest time. The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible. This is achieved by building processor hardware that is capable of understanding and executing a series of operations.

RISC VS. CISC

RISC and CISC architecture can be compared by examining the example below;

Multiplying Two Numbers in Memory

On the right is a diagram representing the storage scheme for a generic computer. The main memory is divided into locations numbered from (row) 1: (column) 1 to (row) 6: (column) 4. The execution unit is responsible for carrying out all computations. However, the execution unit can only operate on data that has been loaded into one of the six registers (A, B, C, D, E, or F). Let's say we want to find the product of two numbers - one stored in location 2:3 and another stored in location 5:2 - and then store the product back in the location 2:3.



The CISC Approach

For this particular task, a CISC processor would come prepared with a specific instruction (we'll call it "MULT"). When executed, this instruction loads the two values into separate registers, multiplies the operands in the execution unit, and then stores the product in the appropriate register. Thus, the entire task of multiplying two numbers can be completed with one instruction:

MULT 2:3, 5:2

MULT is what is known as a "complex instruction." It operates directly on the computer's memory banks and does not require the programmer to explicitly call any loading or storing functions. It closely resembles a command in a higher level language. For instance, if we let "a" represent the value of 2:3 and "b" represent the value of 5:2, then this command is identical to the C statement "a = a * b."

One of the primary advantages of this system is that the compiler has to do very little work to translate a high-level language statement into assembly. Because the length of the code is relatively short, very little RAM is required to store instructions. The emphasis is put on building complex instructions directly into the hardware.

THE RISC APPROACH

RISC processors only use simple instructions that can be executed within one clock cycle. Thus, the "MULT" command described above could be divided into three separate commands: "LOAD," which moves data from the memory bank to a register, "PROD," which finds the product of two operands located within the registers, and "STORE," which moves data from a register to the memory banks. In order to perform the exact series of steps described in the CISC approach, a programmer would need to code four lines of assembly:

```
LOAD A, 2:3
LOAD B, 5:2
PROD A, B
STORE 2:3, A
```

At first, this may seem like a much less efficient way of completing the operation. Because there are more lines of code, more RAM is needed to store the assembly level instructions. The compiler must also perform more work to convert a high-level language statement into code of this form.

RISC AND CISC COMPARISON

RISC	**CISC**
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	Single clock, reduced instruction only
Memory-Memory: “LOAD” and “STORE” incorporated in instructions	Register ro register: “LOAD” and “STORE”
Small code sizes, high cycles per second	Low cycles per second, large code sizes
Transistors used for storing complex instructions	Spends more transistors on memory registers

Separating the "LOAD" and "STORE" instructions actually reduces the amount of work that the computer must perform. After a CISC-style "MULT" command is executed, the processor automatically erases the registers. If one of the operands needs to be used for another computation, the processor must re-load the data from the memory bank into a register. In RISC, the operand will remain in the register until another value is loaded in its place.

The CISC approach attempts to minimize the number of instructions per program, sacrificing the number of cycles per instruction. RISC does the opposite, reducing the cycles per instruction at the cost of the number of instructions per program

SECTION TWO

CHAPTER SIX

REGISTERS

Registers are used for storing variables. Because registers are located inside the CPU, they are much faster than memory. Accessing a memory location requires the use of a system bus, so it takes longer time. Accessing data in a register usually takes no time. Therefore, variables should be often kept in registers. From the assembly language point of view, this chapter discusses the

80x86 register sets. The 80x86 processors provide a set of general purpose registers, segment registers and some special purpose registers. Certain members of the family provide additional registers, although typical applications do not use them. Register sets are very small and most registers have special purpose which limits their use as variables, but they are still an excellent place to store temporary data of calculations.

6.1 GENERAL PURPOSE REGISTERS

8086 CPU has eight 16 bit general purpose registers; each register has its own name:

AX- the accumulator register (divided into AH/AL)

BX- the base register (divided into BH/BL)

CX-the count register (divided into CH/CL)

DX- the data register (divided into DH/DL)

SI- source index register

DI- destination index register

BP- base pointer

SP- stack pointer

Despite the name of a register, it's the programmer who determines the usage for each general purpose register. The size of the above general purpose registers are 16bit, it's something like: 0011000000111001_b

Four general purpose registers (AX, BX, CX, and DX) are made of two separate 8 bit registers, e.g

If $AX=0011000000111001_{b,}$ then $AH=00110000_b$ and $AL=00111001_{b.}$ Therefore, when any of the 8 bit registers are modified, 16 bit register is also updated and vice versa. The same thing also applies for the other 3 registers (BX, CX and DX). "H" stands for high and "L" stands for low part.

While you can use many of these registers interchangeably in a computation, many instructions work more efficiently or absolutely require a specific register from this general purpose register group.

A brief explanation of the general purpose registers is given below;

The ax register: This is also called the accumulator. It is where most arithmetic and logical computations take place. Although, most arithmetic and logical operations can be done in other registers, it is often more efficient to use the ax register for such computations

The bx register: This is the base register. It has some special purpose too. It is commonly used to hold indirect addresses.

The cx register: This is the count register. As the name implies, it is used to count off the number of iterations in a loop or to specify the number of characters in a string.

The dx register: This is the data register. This register has two special purposes: It holds the overflow from certain arithmetic operations and it holds I/O addresses when accessing data on the 80x86 I/O bus.

The si and di register are respectively the source index and the destination index register. These registers can be used to indirectly access memory. These registers are also used with the 8086 string instructions when processing character strings.

The bp register: This is the base pointer. It is similar to bx register. It is used to access parameters and local variables in a procedure.

The sp register: This is the stack pointer. It has a very special purpose of maintaining the program stack.

6.2 SEGMENT REGISTERS

8086 CPU has four 16 bit general purpose registers; each register has its own name:

cs register stands for the code segment register

ds register stands for the data segment register

es register stands for the extra segment register

ss register stands foe the stack segment register

Segment registers points at the beginning of a segment in memory. Segments of memory on the 8086 can be no larger than 65,536 bytes long (64K). A brief explanation of the segment registers is given below;

The cs register points at the segment containing the currently executing machine instructions. Despite the 64K segment limitation, 8086 programs can be longer than 64K. This can be possible if multiple code segments are in memory.

The ds register generally points at global variables for the program. Again, one is limited to 65,536 bytes of data in the data segment but the value of ds can always be changed in order to access additional data in other segments.

The es register are often used by programs to gain access to segments when it is difficult or impossible to modify the other segment registers.

The ss register points at the segment containing the 8086 stack. The stack is where the 8086 stores important machine state information, subroutine return addresses, procedure parameters and local variables. In general, the stack segment is not always modified because too many things in the system depend upon it. Although, it is theoretically possible to store data in the segment registers, this is not a good idea.

The segment registers have a very special purpose- pointing at accessible blocks of memory. Any attempt to use the registers for any other purpose may result in considerable grief, especially if intended to move up to better CPU like 80386. Segment registers work together with general purpose register to access any memory value.

e.g If we would like to access memory at the physical address 12345_h(hexadecimal), we should set the DS=1230_h and SI=0045_h. This is good, since this way we can access much more information than with a single register that is limited to 16 bit.

CPU make a calculation of physical address by multiplying the segment register by 10_h and adding a general purpose register to it ($1230_h * 10_h + 45_h = 12345_h$)

The address formed with two registers is called an **effective address.** By default **BX, SI** and **DI** registers work with **DS** segment register; **BP** and **SP** work with **SS** segment register. Other general purpose registers cannot form an effective address. Also, although **BX** can form an effective address, **BH** and **BL** cannot.

10.2 SPECIAL PURPOSE REGISTERS

There are two types of special purpose registers on the 8086 CPU: the instruction pointer (ip) and the flags register. These registers are not accessed the same way other 8086 registers are accessed. Instead, the CPU generally manipulates these registers directly. A brief explanation of the special purpose registers is given below;

The ip register contains the address of the currently executing instruction. It is a 16 bit pointer which provides a pointer into the current code segment. IP works together with the CS segment register and it points to the currently executing instruction.

The flags register is unlike the other registers on the 8086. The other registers hold eight or 16 bit values. The flags register is simply an eclectic collection of on e bit values which help determine the current state of the processor. Flags register is modified automatically by CPU after mathematical operations, this allows to determine the type of the result and to determine the conditions to transfer control to other parts of the program. Generally, the flags register cannot be accessed directly. Although the flags register is 16 bits wide, the 8086 uses only nine of those bits. These bits are explained below;

The carry bits (C): It holds the carry after addition or after subtraction (or the borrow after subtraction). It also indicates an error condition.

The parity bits: it is a count of one in a number expressed as even or odd. It is used as error detection in some circuits.

The Auxiliary carry bit (A): This holds the half-carry after addition or the borrow after subtraction between bit positions 3 and 4 of the results. It is set by a carry out of the lowest nibble (1/2 byte) or a borrow into the lowest nibble.

The zero flag bit (Z): It is set when the result of the operation is zero.

The sign bit (S): This flag holds the arithmetic sign of the result after arithmetic or logic instruction is executed.

The trap flag (T): If the T flag is enabled, the microprocessor interrupts the flow of the program on conditions as indicated by the debug registers and control register. If T=0, the debugging feature is disabled.

The Interrupt flag bit (I): This flag determines whether an external interrupt are disabled or not.

The Direction flag bit (D): This flag determines the direction which string operations are performed. When cleared, string operations proceed from left to right or vice versa.

The overflow flag bit (O): Overflow occurs when signed numbers are added or subtracted. An overflow indicates that the result has exceeded the capacity of the machine.

Of all these flags, four flags are used most times and they are; zero, carry, sign and

CHAPTER SEVEN

ADDRESSING MODES

One approach to processors places an emphasis on flexibility of addressing modes. Some engineers and programmers believe that the real power of a processor lies in its addressing modes. Most addressing modes can be created by combining two or more basic addressing

modes, although building the combination in software will usually take more time than if the combination addressing mode existed in hardware (although there is a trade-off that slows down all operations to allow for more complexity). The x86 instructions use five different operand types: registers, constants, and three memories addressing schemes. Each form is called an addressing mode. The x86 processors support the register addressing mode, the immediate addressing mode, the indirect addressing mode, the indexed addressing mode and the direct addressing mode.

One approach to processors places an emphasis on flexibility of addressing modes. Some engineers and programmers believe that the real power of a processor lies in its addressing modes. Most addressing modes can be created by combining two or more basic addressing modes, although building the combination in software will usually take more time than if the combination addressing mode existed in hardware (although there is a trade-off that slows down all operations to allow for more complexity).

In a purely othogonal instruction set, every addressing mode would be available for every instruction. In practice, this isn't the case.

Virtual memory, memory pages, and other hardware mapping methods may be layered on top of the addressing modes.

7.1 ADDRESSING MODES WITH REGISTER OPERANDS

Register operands are the easiest to understand. Consider the following forms of the mov instruction:

```
mov ax,ax

mov ax,bx

mov ax,cx

mov ax, dx
```

In the above instructions, the first instruction is the destination register and the second operand is the source register. The first instruction does nothing. It copies the value from the ax register back into the ax register. The remaining three instructions copy the value of bx,cx, and dx into ax. Note that the original values of bx, cx and dx remain the same. The first operand (the destination register) is not limited to ax; you can move values to any of these registers. This mode of addressing is the ***register addressing mode***

7.2 ADDRESSING MODES WITH CONSTANTS

Constants are also pretty easy to deal with. Consider the following instructions:

```
mov ax,25

mov bx, 195

mov cx,2056

mov dx,1000
```

These instructions are all pretty straightforward; they load their respective registers with the specified hexadecimal constant. This mode of addressing is called the ***immediate addressing mode.***

7.3 ADDRESSING MODES WITH MEMORY STRUCTURES

There are three addressing modes which deal with accessing data in memory. There are three main addressing mode found in this category; the ***direct addressing mode,*** the ***indirect addressing mode*** and ***the index addressing mode.*** These addressing modes take the following forms:

```
mov ax, [1000]

mov ax, [bx]

mov ax, [1000+bx]
```

The first instruction uses the ***direct*** addressing mode to load ax with the 16 bit value stored in memory starting at location 1000hex.

The second instruction loads ax from the memory location specified by the contents of the bx register. This is an ***indirect*** addressing mode. Rather than using the value bx, the instruction accesses the memory location whose address appears in bx.

There are many cases where the use of indirection is faster, shorter and better. The last memory addressing mode is the ***indexed*** addressing mode. An example of this memory addressing mode is mov ax, [1000+bx]. The instruction adds the contents of bx with 1000 to produce the address of the memory value to fetch. This instruction is useful for accessing elements of arrays, records and other data structures.

7.4 ADDRESSING MODE WITH STACK MEMORY

The stack plays an important role in all microprocessors. It holds data temporarily and stores return addresses for procedures. The stack memory is a LIFO memory which describes the way data are stored and removed from the stack. Data are placed onto the stack with the PUSH instruction and removed with a POP instruction. The stack memory is maintained by two registers (the stack pointer SP or ESP and the stack segment)s. The stack pointer register always points to an area of memory located within the stack segment. The stack pointer adds to (ss*10h) to form the stack memory address in the real mode.

- PUSH ax: Copies ax into the stack
- POP cx: Removes a word from the stack and places it in cx
- PUSH dx: Copies dx into the stack
- PUSH 123: Copies 123 into the stack
- PUSH A: Copies the word content of all the registers into the stack (AX,BX,CX,DX,SI,DI,SP,BP)
- POP A: Removes data from the stack and places it in the 16 bit registers.

Example

```
.code
start:
        mov ax,23
        mov bx,44
        mov cx,13
        push ax; copies 23 into the stack
        push bx; copies 44 into the stack
        push cx; copies 13 into the stack
        pop cx; removes 13 from the stack and places it back into cx
        pop bx; removes 44 from the stack and places it back into bx
        pop ax; removes 23 from the stack and places it back into ax
```

CHAPTER EIGHT

INSTRUCTION SETS

Like any other programming language, there are going to be several instructions you use all the time, some you use occasionally, and some you will rarely, if ever, use. These are called the 80x86 instruction sets.

8.1 THE 80X86 INSTRUCTION SETS

80x86 instructions can be roughly divided into eight different classes;

1. Data movement instructions
    - mov, lea, les, push, pop, pushf, popf

2 Conversions

- cbw, cwd, xlat

3 Arithmetic instructions

- add,inc,sub,dec,cmp,neg,mul,imul,div,idiv

4 Logical shift

- and, or, xor, not, shl, shr, rcl, rcr

5 I/O instructions

- in,out

6 String instructions

- mov,stos,lods

7 Program flow instructions

- jmp, call, ret
- conditional jumps

8 Miscellaneous instructions

- clc, stc, cmc

The most commonly used of all these classes are the data movement instructions and the arithmetic instructions. The mov isnstruction is the most commonly used instruction of all the data movement instruction. The mov instruction is actually two instructions merged into the same instruction. The two forms of the mov instruction take the following forms;

mov reg, reg/memory/constant

mov reg, reg

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

The arithmetic and logical instruction take the following forms:

add reg, reg/memory/constant

sub reg, reg/memory/constant

cmp reg, reg/memory/constant

and reg, reg/memory/constant

or reg, reg/memory/constant

not reg/memory

The following sections describe some of the instructions in these groups and how they operate. The 80x86 instruction have simple semantics.

The ***add*** instruction adds the value of the second operand to the first (register) operand, leaving the sum in the first operand.

The ***sub*** instruction subtracts the value of the second operand from the first, leaving the difference in the first operand.

The ***cmp*** instruction subtracts the value of the second operand from the first, leaving the difference in the first operand.

The **and & or** instructions compute the corresponding bitwise logical operation on the two operands and store the result in the first operand.

The ***not*** instruction invert the bit in the single memory or register operand.

8.2 CONTROL TRANSFER INSTRUCTION

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

ja dest--jump if above

jae dest--jump if above or equal

jb dest--jump if below

jbe dest--jump if below or equal

je dest—jump if equal

jne dest—jum if not equal

jmp dest-unconditional jump

iret return from an interrupt

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

write integer values. Get will stop and prompt the user for a hexadecimal value and then store that value into the ax register. Put displays (in hexadecimal) the value of the ax register. The remaining instructions do not require any operands, they are **halt** and **brk.** Halt terminates program execution and brk stops the program in a state that it can be restarted.

8.3 THE STANDARD INPUT ROUTINES

While the standard library provides several input routines, there are three in particular that will be used be used most times.

- Getc (gets a character)
- Gets(gets a string)
- Getsm

Getc reads a single character from the keyboard and returns that character in a register. It does not modiffy any other registers. As usual, the carry flag returns the error status. You do not need to pass getc any values in th rgisters. Getc does not echo the input character to the display screen. You must explicitly print the character if you want it to appear on the output monitor.

The gets routine reads an entire line of text from the keyboard. It stores each successive character of the input line into a byte array whose base address in the es:di register pair. This array must have a room of 128bytes. The gets routine will read each character and place it in the array except for the carriage return character. Gets terminates the input line with a zero byte. Gets echoes each character you type to the display device, it also handles simple line editing functions such as backspace.

The getsm routine reads a string from the keyboard and returns a pointer to that string in nes:di pair register. The difference between gets an getsm is that you do not have to pass the address of an input buffer in es:di. Getsm automatically allocates storage on the heap with a call to malloc and returns a pointer to the buffer in es:di.

8.4 THE STANDARD OUTPUT ROUTINES

The basic standard output routines are; PUTC, PUTCR, PUTS, PUTH, PUTI, PRINT and PRINTF

Putc outputs a single character to the display device. It outputs the character appearing in the al register. It does not affect any registers unless there is an error on output (the carry flag denotes error/no error).

Putcr outputs a "newline" to the standard output.

Puts (put a string) routine prints the zero terminated string at which es:di points. Puts does not automatically output a new line after printing the string.

The puth routine prints the value in the al register as exactly two hexadecimal digits including a leading zero byte if the value is in the range (0..Fh).

The puti routine puts the value in the ax as a signed 16 bit integer

The print routine is one of the most often called procedures in the library. It prints the zero terminated string that immediately follows the call to the string.

Printf uses the escape character ("\") to print special characters in the fashion similar to, but not identical to C's printf.

8.5 MACROS

Many assemblers support *macros*, programmer-defined symbols that stand for some sequence of text lines. This sequence of text lines may include a sequence of instructions, or a sequence of data storage pseudo-ops. Once a macro has been defined using the appropriate pseudo-op, its name may be used in place of a mnemonic. When the assembler processes such a statement, it replaces the statement with the text lines associated with that macro, then processes them just as though they had appeared in the source code file all along (including, in better assemblers, expansion of any macros appearing in the replacement text).

Since macros can have 'short' names but expand to several or indeed many lines of code, they can be used to make assembly language programs appear to be much shorter (require less lines of source code from the application programmer - as with a higher level language). They can also be used to add higher levels of structure to assembly programs, optionally introduce embedded de-bugging code via parameters and other similar features.

Many assemblers have built-in macros for system calls and other special code sequences.

Macro assemblers often allow macros to take parameters. Some assemblers include quite sophisticated macro languages, incorporating such high-level language elements as optional parameters, symbolic variables, conditionals, string manipulation, and arithmetic operations, all usable during the execution of a given macros, and allowing macros to save context or exchange information. Thus a macro might generate a large number of assembly language instructions or data definitions, based on the macro arguments. This could be used to generate record-style data structures or "unrolled" loops, for example, or could generate entire algorithms based on complex parameters. An organization using assembly language that has been heavily extended using such a macro suite can be considered to be working in a higher-level language, since such programmers are not working with a computer's lowest-level conceptual elements.

CHAPTER NINE

ASSEMBLY LANGUAGE PROGRAM

A program written in assembly language consists of a series of *instructions*--mnemonics that correspond to a stream of executable instructions, when translated by an assembler, that can be loaded into memory and executed.

For example, an x86/IA-32 processor can execute the following binary instruction as expressed in machine language (see x86 assembly language):

- Binary: 10110000 01100001 (Hexadecimal: B0 61)

The equivalent assembly language representation is easier to remember (example in Intel syntax, more *mnemonic*):

```
MOV AL, #61h
```

This instruction means:

- Move the value 61h (or 97 decimal; the h-suffix means hexadecimal; the pound sign means move the immediate value, not location) into the processor register named "AL".

The mnemonic "mov" represents the opcode **1011** which *moves* the value in the second operand into the register indicated by the first operand. The mnemonic was chosen by the instruction set designer to abbreviate "move", making it easier for the programmer to remember. A comma-separated list of arguments or parameters follows the opcode; this is a typical assembly language statement.

9.1 OVERVIEW OF ASSEMBLY LANGUAGE PROGRAMMING

In practice many programmers drop the word *mnemonic* and, technically incorrectly, call "mov" an *opcode*. When they do this they are referring to the underlying binary code which it represents. To put it another way, a mnemonic such as "mov" is not an opcode, but as it

symbolizes an opcode, one might refer to "the opcode mov" for example when one intends to refer to the binary opcode it symbolizes rather than to the symbol--the mnemonic--itself. As few modern programmers have need to be mindful of actually what binary patterns are the opcodes for specific instructions, the distinction has in practice become a bit blurred among programmers but not among processor designers.

Transforming assembly into machine language is accomplished by an assembler, and the reverse by a disassembler. Unlike in high-level languages, there is usually a one-to-one correspondence between simple assembly statements and machine language instructions. However, in some cases, an assembler may provide *pseudoinstructions* which expand into several machine language instructions to provide commonly needed functionality. For example, for a machine that lacks a "branch if greater or equal" instruction, an assembler may provide a pseudoinstruction that expands to the machine's "set if less than" and "branch if zero (on the result of the set instruction)". Most full-featured assemblers also provide a rich macro language (discussed below) which is used by vendors and programmers to generate more complex code and data sequences.

Each computer architecture and processor architecture has its own machine language. On this level, each instruction is simple enough to be executed using a relatively small number of electronic circuits. Computers differ by the number and type of operations they support. For example, a new 64-bit machine would have different circuitry from a 32-bit machine. They may also have different sizes and numbers of registers, and different representations of data types in storage. While most general-purpose computers are able to carry out essentially the same functionality, the ways they do so differ; the corresponding assembly languages reflect these differences.

9.2 THE LINKER

The Linker is used for linking the output of an assembler to an executable file.

Basically a linker is used for large projects where there is more than one file to be assembled. Each of the assembly language files might contain references to code elements in the other files.

A linker is the program that ties all the loose ends together and makes a single program out of the pieces.

Thus, technically, an assembler converts assembly language files to *object files* (basically machine code with some loose ends), and a linker connects all the object files together and makes a single executable program out of them.

However, there are some assemblers that do not require the use of a linker. E.g Flat Assembler FASM.

Actually, a linker is not just specific to assembly language programing. It is used fairly much regardless of the language in which you are programming.

An example of a linker is the ALINK

9.3 COMMON ASSEMBLERS

Some commonly used assemblers are listed below;

Lazy Assembler (LZASM)
GoASM
Flat Assembler (FASM)
MASM32
Pass32
Netwide Assembler (NASM)

In this course, the Microsoft Micro Assembler (MASM) will be used.

9.4 A SIMPLE HELLO WORLD PROGRAM USING FLAT ASSEMBLER(FASM)

```
Include '%fasminc%/win32ax.inc'

.code

start:

        invoke MessageBox, HWND_DESKTOP, "Hello World! ", "Win32 Assembly",MB_OK

        invoke ExitProcess, 0

.endstart
```

The first line *includes* a special macro file win32ax.inc

The next line tells the assembler we are going to include some code now.

The line start: is simply a label. It just gives a name to one of the lines of our program so we can refer to it elsewhere. We'll use labels often, for example when writing loops or using other jump instructions. The label will simply allow us to refer to that particular location within our code, by name; say for example if we wanted to jump to that location from somewhere else.

The program is ended with the .end directive. It takes one parameter, the name of a label corresponding to the entry point of the program (which doesn't need to be at the start of the program). When the program is loaded into memory by the operating system, execution of the program will begin at this point.

9.5 A SIMPLE HELLO WORLD PROGRAM USING NETWIDE ASSEMBLY LANGUAGE (NASM)

```
 1:         %include'system.inc'
 2:
 3:         section    .data
```

```
 4:       hello    db       'Hello, World!', 0Ah
 5:       hbytes   equ      hello
 6:
 7:       section  .text
 8:       global   _start
 9:       _start:
10:       push     dword hbytes
11:       push     dword hello
12:       push     dword stdout
13:       sys.write
14:
15:       push     dword 0

16:    sys.exit
```

Here is what it does: Line 1 includes the defines, the macros, and the code from system.inc.

Lines 3-5 are the data: Line 3 starts the data section/segment. Line 4 contains the string "Hello, World!" followed by a new line (**0Ah**). Line 5 creates a constant that contains the length of the string from line 4 in bytes.

Lines 7-16 contain the code. Note that FreeBSD uses the *elf* file format for its executables, which requires every program to start at the point labeled **_start** (or, more precisely, the linker expects that). This label has to be global.

Lines 10-13 ask the system to write **hbytes** bytes of the **hello** string to **stdout**.

Lines 15-16 ask the system to end the program with the return value of **0**. The **SYS_exit** syscall never returns, so the code ends there.

CHAPTER TEN

JOB CONTROL LANGUAGE

10.1 INTRODUCTION

Job Control Language (**JCL**) is a scripting language used on IBM mainframe operating systems to instruct the system on how to run a batch job or start a subsystem. The term "Job Control Language" can also be used generically to refer to all languages which perform these functions.

JCL consists of control statements that do the following;

- introduce a computer job to the operating system
- request hardware devices
- direct the operating system on what is to be done in terms of running applications and scheduling resources

JCL is not used to write computer programs. Instead it is most concerned with input/output---telling the operating system everything it needs to know about the input/output requirements. It provides the means of communicating between an application program and the operating system and computer hardware.

JCL can be difficult because of the way it is used. A normal programming language, however difficult, soon becomes familiar through constant usage. This contrasts with JCL in which language features are used so infrequently that many never become familiar.

JCL can be difficult because of its design - JCL:

- consists of individual parameters, each of which has an effect that may take pages to describe
- has few defaults--must be told exactly what to do
- requires specific placement of commas and blanks
- is very unforgiving--one error may prevent execution

JCL is not necessarily difficult because most users only use a small set of similar JCL that never changes from job to job.

10.2 BASIC SYNTAX AND RULES OF JCL

//NAME	OPERATION	OPERAND,OPERAND,OPERAND	COMMENTS
|	|	|	|
name field	operation field	operand field	comment field

name field - identifies the statement so that other statements or the system can refer to it. The name field must begin immediately after the second slash. It can range from 1 to 8 characters in length, and can contain any alphanumeric (A to Z) or national (@ $ #) characters. operation field - specifies the type of statement: JOB, EXEC, DD, or an operand command. operand field - contains parameters separated by commas. Parameters are composites of prescribed words (keywords) and variables for which information must be substituted. comments field - optional. Comments can be extended through column 80, and can only be coded if there is an operand field.

General JCL Rules:

- Must begin with // (except for the /* statement) in columns 1 and 2
- Is case-sensitive (lower-case is just not permitted)

- NAME field is optional
- must begin in column 3 if used
- must code one or more blanks if omitted
- OPERATION field must begin on or before column 16
- OPERATION field stands alone
- OPERANDS must end before column 72
- OPERANDS are separated by commas

All fields, except for the operands, must be separated by one blank.

More on JCL

```
//LABEL     OPERATION     OPERAND,     OPERAND,
//                                OPERAND,OPERAND,
//                                     OPERAND,
//    OPERAND
```

When the total length of the fields on a control statement exceeds 71 columns, continue the fields onto one or more following statements.

- Interrupt the field after a complete operand (including the comma that follows it) at or before column 71
- Code // in columns 1 and 2 of the following line
- Continue the interrupted statement beginning anywhere in columns 4 to 16.

Commenting JCL

```
//* THIS IS A COMMENT LINE
```

JCL should be commented as you would any programming language. The comments statement contains //* in columns 1 to 3, with the remaining columns containing any desired comments. They can be placed before or after any JCL statements following the JOB statement to help

document the JCL. Comments can also be coded on any JCL statement by leaving a blank field after the operand field.

10.3 TYPES OF JCL STATEMENTS

JOB Identifies the beginning of a job

EXEC Indicates what work is to be done

DD Data Definition, i.e., Identifies what resources are needed and where to find them

10.4 THE JOB STATEMENT

The JOB statement informs the operating system of the start of a job, gives the necessary accounting information, and supplies run parameters. Each job must begin with a single JOB statement.//jobname JOB USER=userid

jobname - a descriptive name assigned to the job by the user which is the banner on your printout
- any name from 1 to 8 alphanumeric (A-Z,0-9) or national ($,@,#) characters
- first character must be alphabetic or national

JOB - indicates the beginning of a job

Userid - a 1 to 7 character user identification assigned to access the system

10.5 THE EXEC STATEMENT

Use the EXEC (execute) statement to identify the application program or cataloged or in-stream procedure that this job is to execute and to tell the system how to process the job.

//stepname EXEC procedure,REGION=####K

or

//stepname EXEC PGM=program,REGION=####K

stepname - an optional 1 to 8 character word used to identify the step
EXEC - indicates that you want to invoke a program or cataloged procedure
procedure - name of the cataloged procedure to be executed
program - name of the program to be executed
REGION=####K - amount of storage to allocate to the job

10.5 DATA DEFINITION (DD) STATEMENT

A DD (Data Definition) statement must be included after the EXEC statement for each data set used in the step. The DD statement gives the data set name, I/O unit, perhaps a specific volume to use, and the data set disposition. The system ensures that requested I/O devices can be allocated to the job before execution is allowed to begin.

The DD statement may also give the system various information about the data set: its organization, record length, blocking, and so on.

//ddname DD operand,operand,etc.

ddname - a 1 to 8 character name given to the DD statement

DD - DD statement identifier

operand - parameters used to define the input or output dataset

The DD Statement

- appears after an EXEC statement
- gives the system information on many things, including the dataset attributes, the disposition of the dataset when the job completes, and which input/output device(s) to use

Happy Lecture