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: