Lecture note
Day 2
To be presented by group H&J

Introduction of data representation and its purpose

Representation of data is the base for any field of study. Whenever collection of data is started and the range of data increases rapidly, an efficient and convenient technique for representing data is needed. Several representation techniques have been developed in this concern. Data presentation and data representation are terms having similar meaning and importance. The purpose of data representation is for recording of the information and transmitting information across two or more ends.

Techniques for data representation:

Non graphical techniques: Tabular Form, Case Form
Graphical techniques: Pie Chart, Bar Chart, Line Graphs, Geometrical Diagrams

Numbering system and their representation

A number system is a technique used in representing numbers in the computer system architecture. The total number of digits used in a number system is called its base or radix. The base is written after the number as subscript; for instance 10001102 (1000110 base 2), 5610 (56 to base of 10), 718 (71 base 8) etc.

In the number system, each number is represented by its base. If the base is 2 it is a binary number, if the base is 8 it is an octal number, if the base is 10, then it is called decimal number system and if the base is 16, it is part of the hexadecimal number system.

Decimal Number System

In the **decimal number system**, the numbers are represented with base 10. The way of denoting the decimal numbers with base 10 is also termed as decimal notation. This number system is widely used in computer applications. It is also called the base-10 number system which consists of 10 digits, such as, 0,1,2,3,4,5,6,7,8,9. Each digit in the decimal system has a position and every digit is ten times more significant than the previous digit. Suppose, 25 is a decimal number, then 2 is ten times more than 5

Conversion from Other Bases to Decimal Number System

We can convert any other base system such as binary, octal and hexadecimal to the equivalent decimal number.

Binary to Decimal

In this conversion, a number with base 2 is converted into number with base 10. Each binary digit here is multiplied by decreasing power of 2. Let us see one example:

Example: Convert $(11011)_2$ to decimal number.
Solution: Given $(11011)_2$ a binary number.
We need to multiply each binary digit with the decreasing power of 2. That is;
$1\times2^4+1\times2^3+0x2^2+1\times2^1+1\times2^0$
$=16+8+0+2+1$
$=27$
Therefore, $(11011)_2 = (27)_{10}$

Octal to Decimal

In this conversion, a number with base 8 is converted into number with base 10. Each digit of octal number here is multiplied by decreasing power of 8. Let us see one example:

Example: Convert 121_8 into the equivalent decimal number.
Solution: Given $(121)_8$ is an octal number
Here, we have to multiply each octal digit with the decreasing power of 8, such as;
$1\times8^2+2\times8^1+1\times8^0$
$=64+16+1$
$=81$

Hexadecimal to Decimal

In this conversion, a number with base 16 is converted into number with base 10. Each digit of hex number here is multiplied by decreasing power of 16. Let us understand with the help of an example:

Example: Convert 12_{16} into a decimal number.
Solution: Given 12_{16}
Multiply each digit with decreasing power of 16 to obtain equivalent decimal number.
$1\times16^1+2\times16^0$
$=16+2$
$=18$

Decimal Number System to Other Bases

Earlier we learned about converting other base number systems into a decimal number, here we will learn how to convert a decimal number into different base numbers. Let us see one by one.

Decimal to Binary

To convert a decimal number into an equivalent binary number we have to divide the original number system by 2 until the quotient is 0, when no more division is possible. The remainder so obtained is counted for the required number in the order of LSB (Least significant bit) to MSB (most significant bit). Let us go through the example.

Example: Convert 26_{10} into a binary number**.**
Solution: Given 26_{10} is a decimal number.
Divide 26 by 2
26/2 = 13 Remainder →0 (MSB)
13/2 = 6 Remainder →1
6/2 = 3 Remainder →0
3/2 = 1 Remainder →1
½ = 0 Remainder →1 (LSB)
Hence, the equivalent binary number is $(11010)_2$

Decimal to Octal
Here the decimal number is required to be divided by 8 until the quotient is 0. Then, in the same way, we count the remainder from LSB to MSB to get the equivalent octal number.

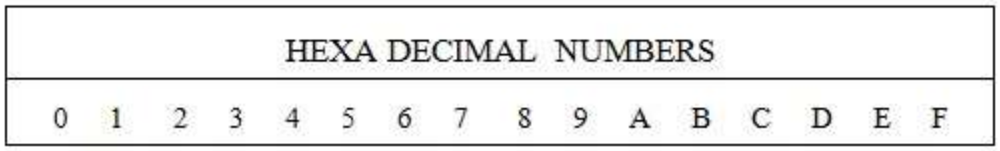
Example: Convert 65_{10} into an octal number.
Solution: Given 65_{10} is a decimal number.
Divide by 8
65/8 = 8 Remainder →1 (MSB)
8/8 = 1 Remainder →0
⅛ = 0 Remainder →1 (LSB)
Hence, the equivalent octal number is $(101)_8$

Decimal to Hexadecimal
The given decimal number here is divided by 16 to get the equivalent hex. The division of the number continues until we get the quotient 0.

Example: Convert 127_{10} to a hexadecimal number.
Solution: Given 127_{10} is a decimal number.
Divide by 16
127/16 = 7 Remainder →15
7/16 = 0 Remainder → 7
In the hexadecimal number system, alphabet E is considered as 15.
Hence, 127_{10} is equivalent to $7E_{16}$

Hexadecimal Numbering System

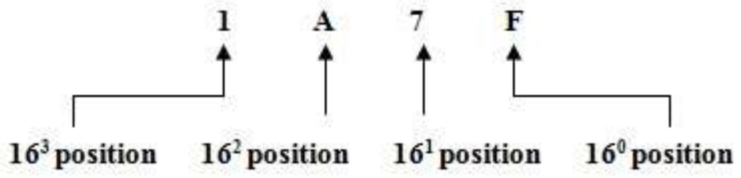
In the term Hexadecimal number system, the word ‘Hexa’ means 16. So as the name suggests, the Hexadecimal number system will have 16 values for representation of numeric that is – numbers from 0 to 9 and letters from A to F.



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

Note that each Hexa decimal number represents a group of four digits, called **“Nibble”.** In Hexa decimal number system, the position of digit is weighted as the powers of 16. This means that in a column, the value of a digit is sixteen times that of the digit to its right.

Ex: (10)16, (56)16, (3Fb1)16, (A51D0)16
Let’s see some more examples to understand this clearly



This Hexadecimal system is used in computer registers to store the addresses of the data. If we have to give a large number of binary strings, for suppose 101111011000101111010110001101, it is very much difficult and create a lot of confusion. So computer uses Hexadecimal numbers in representation of such strings.

Conversion of Hexadecimal Numbers
As we convert the binary number into decimal numbers and decimal numbers into binary numbers, we can also convert the Hexadecimal numbers into binary and decimal number systems.

Conversion from Binary to Hexadecimal
To convert the binary numbers into Hexadecimal numbers, we group the binary digits as a set of four. If necessary, add zeros to complete the set. Next, write the corresponding number to each group of the four digits.
Ex 1: Convert (0110101110001100)2 to Hexadecimal.
Given 0110101110001100, group the binary digits as a set of four digits.

```
0110 1011 1000 1100
  6    B    8    C
```

So (0110101110001100)2 = $(6B8C)_{16}$

Conversion from Hexadecimal to binary
To convert the Hexadecimal number into binary number, we follow the reverse steps explained above. First we write each digit of Hexadecimal number in binary form and then grouping the binary digits.
See some examples given below and then you will get a clear idea on this

Ex 1:   convert 5A9 into binary

    5   A   9

0101 1010 1001

So (5A9)16 = (10110101001)2

Conversion from Hexadecimal to decimal

To convert a Hexadecimal number to decimal number, we should write each digit / letter in the form of decimal number with base 10 and then the Hexadecimal ha to be written as the sum of powers of 16.

See the explained example,

Ex 1: Convert Hexadecimal number 1A9B into decimal.

1 A 9 B = 1 x 16^3 + A x 16^2 + 9 x 16^1 + B x 16^0 (writing powers of 16)

= 4096 + A (256) + 9 (16) + B (1)

= 4096 + 10 (256) + 144 + 11

= 6811

Therefore, (1A9B)16 = (6811)10

Conversion from decimal to Hexadecimal

The simple way to make a conversion from decimal to Hexadecimal number system is same as the conversion of decimal numbers to binary. There the repeated division process is done by 2 but as the Hexadecimal number is the base of 16, we should do the repeated division process by 16 instead of 2. The reminders are noted in the sequence of last to first.

Let's learn this with an example: even number

746 ÷ 16 Quotient 10, remainder 46 Result > A

46 ÷ 16 Quotient 14, remainder 2 Result > E A

2 ÷ 16 Quotient 0, remainder 2 Result > 2 E A

So (746)10 = (2EA) $_{16}$

Note:

Important and mandatory thing to remember in Hexadecimal number system, if we write (10)16 here, this means not 10. This is 1× (16) + 0× (16) in Hexadecimal. Similarly, if we write 19 or 32 in Hexadecimal, they are not like nineteen or Thirty-two. They are 1× (16) + 9× (16) and 3× (16) + 2× (16) in Hexadecimal system. Simply,

(10)10 is not equal to (10)16

(19)10 is not equal to (19)16

(32)10 is not equal to (32)16

The highest positive decimal in decimal number system is 255; in this Hexadecimal system the highest number formed using the Hexadecimal digits is FF. This is equal to 255 in decimal number system and 1111 1111 in binary number system.

The least 3-bit Hexadecimal number is 10016 (25610) & highest is FFF16 (409510). The maximum 4-digit Hexadecimal number is FFFF16 (65,53510).

Lecture note

Day 3

To be presented by group D, A & I

ENCODING

Encoding is the process of converting data from one form to another. While "encoding" can be used as a verb, it is often used as a noun, and refers to a specific type of encoded data. There are several types of encoding, including image encoding, audio and video encoding, and character encoding.

Types of Encoding

While we view text documents as lines of text, computers actually see them as binary data, or a series of ones and zeros. Therefore, the characters within a text document must be represented by numeric codes. In order to accomplish this, the text is saved using one of several types of character encoding.

American Standard Code for Information Interchange (ASCII)

ASCII character encoding provides a standard way to represent characters using numeric codes. These include upper and lower-case English letters, numbers, and punctuation symbols.

ASCII uses 7 bits to represent each character

Since ASCII uses 7 bits, it only supports 2^7, or 128 values. Therefore, the standard ASCII character set is limited to 128 characters. While this is enough to represent all Standard English letters, numbers, and punctuation symbols, it is not sufficient to represent all special characters or characters from other languages. Even Extended ASCII, which supports 8 bit values, or 256 characters, does not include enough characters to accurately represent all languages. Therefore, other character sets, such as Latin-1 (ISO-8859-1), UTF-8, and UTF-16 are commonly used for documents and webpages that require more characters.

A summary of ASCII table is shown below;

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

Examples: Represent HOME in binary format using ASCII

Since HOME is an alphabets, ASCII is suitable for this conversion

H – 1001000, O – 1001111, M – 1001101, E – 1000101
HOME is 1001000100111110011011000101

Binary Coded Decimal

Binary Coded Decimal (BCD) is a system of writing numerals that assigns a four-digit binary code to each digit 0 through 9 in a decimal (base-10) numeral. The four-bit BCD code for any particular single base-10 digit is its representation in binary notation, as follows:

0	1	2	3	4	5	6	7	8	9
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Numbers larger than 9, having two or more digits in the decimal system, are expressed digit by digit. For example, the BCD rendition of the base-10 number 1895 is

1895 in BCD is 0001 1000 1001 0101

The binary equivalents of 1, 8, 9, and 5, always in a four-digit format, go from left to right.

1895 in binary is 11101100111

Other bit patterns are sometimes used in BCD format to represent special characters relevant to a particular system, such as sign (positive or negative), error condition, or overflow condition.

However, arithmetic process for numbers represented in BCD code is more complicated than the straight binary

Extended Binary Coded Decimal Interchange Code

Extended binary coded decimal interchange code (EBCDIC) is an 8-bit binary code made for numeric and alphanumeric characters. It was developed and used by IBM. It is a coding representation in which symbols, letters and numbers are presented in binary language.

EBCDIC is an 8-bit character encoding widely used in IBM midrange and mainframe computers. This encoding was developed in 1963 and 1964. EBCDIC was developed to enhance the existing capabilities of binary-coded decimal code. This code is used in text files of S/390 servers and OS/390 operating systems of IBM

The table of EBDIC is shown below;

zone bits				Numeric bits			
111	110	110	110	8	4	2	1
1	0	1	0				
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 **HOME** to binary using EBCDIC coding system

H – 11001000, O – 11010110, M – 11010100 E – 11000101

HOME is 11001000110101101101010011000101 using EBDIC coding

MODES OF DATA REPRESENTATION

Computer uses a fixed number of bits to represent a piece of data, which could be a number, a character, or others. A n-bit storage location can represent up to 2^n distinct entities. For example, a 3-bit memory location can hold one of these eight binary patterns: 000, 001, 010, 011, 100, 101, 110, or 111. Hence, it can represent at most 8 distinct entities. You could use them to represent numbers 0 to 7, numbers 8881 to 8888, characters 'A' to 'H', or up to 8 kinds of fruits like apple, orange, banana; or up to 8 kinds of animals like lion, tiger, etc.

INTEGER REPRESENTATION

Integers are whole numbers or fixed-point numbers with the radix point fixed after the least-significant bit. They are contrast to real numbers or floating-point numbers, where the position of the radix point varies. It is important to take note that integers and floating-point numbers are treated differently in computers. They have different representation and are processed differently (e.g., floating-point numbers are processed in a so-called floating-point processor). Floating-point numbers will be discussed later.

Computers use a fixed number of bits to represent an integer. The commonly-used bit-lengths for integers are 8-bit, 16-bit, 32-bit or 64-bit. Besides bit-lengths, there are two representation schemes for integers:

Unsigned Integers:

Signed Integers:

UNSIGNED INTEGERS:

Unsigned integers can represent zero and positive integers, but not negative integers. The value of an unsigned integer is interpreted as "the magnitude of its underlying binary pattern".

Example 1: Suppose that n=8 and the binary pattern is 0100 0001, the value of this unsigned integer is 1×2^0 + 1×2^6 = 65.

Example 2: Suppose that n=16 and the binary pattern is 0001 0000 0000 1000, the value of this unsigned integer is 1×2^3 + 1×2^12 = 4104.

Example 3: Suppose that n=16 and the binary pattern is 0000 0000 0000 0000, the value of this unsigned integer is 0.

SIGNED INTEGERS:

Signed integers can represent zero, positive integers, as well as negative integers. Three representation schemes are available for signed integers:

Sign-Magnitude Representation

1's Complement Representation

2's Complement Representation

FLOATING POINTS REPRESENTATION

Fractional numbers such as and large 72.9100243 numbers like 987654321which fall outside the range of a d-bit word machine , say for instance 16-bit word machine are stored and processed in Exponential form. In exponential form these numbers have an embedded decimal point and are called floating point numbers or real numbers. The floating point representation of a real number x is x = M X 10^E where ‘M’ is called mantissa and ‘E’ is the exponent. So the floating - point representation of the fractional number 72.9100243is 0.729100243 X 10^2 and that of the large number 98765432 is 0.987654321 X 10^2.
Typically computers use a 32-bit representation for a floating point. The left most bit is reserved for the sign. The next seven bits are reserved for exponent and the last twenty four bits are used for mantissa.
The shifting of the decimal point to the left of the most significant digit is called normalization and the numbers represented in the normalized form are known as normalized floating point numbers.
For example, the normalized floating point form of the numbers 0.00695, 56.2547, -684.6 are:
0.00695 = 0.695 X 10^{-2} = .695E-2
56.2547 = 0.562547 X 10^2 = .562547E2
-684.6 = -0.6846 X 10^3 = -.6846E3

COMPUTER INSTRUCTION SET

The instruction set, also called ISA (instruction set architecture), is part of a computer that pertains to programming, which is basically machine language. The instruction set provides commands to the processor, to tell it what it needs to do.
An instruction set architecture is an abstract model of a computer. It is also referred to as architecture or computer architecture. A realization of an ISA, such as a central processing unit, is called an implementation.
An instruction set architecture is distinguished from a microarchitecture, which is the set of processor design techniques used, in a particular processor, to implement the instruction set. Processors with different microarchitectures can share a common instruction set. For example, the Intel Pentium and the Advanced Micro Devices Athlon implement nearly identical versions of the x86 instruction set, but have radically different internal designs
An example of an instruction set is the x86 instruction set, which is common to find on computers today. The instruction set can be classified into two
Reduced instruction set (RISC) & Complex instruction set (CISC)

REDUCED INSTRUCTION SET

RISC, or Reduced Instruction Set Computer is a type of microprocessor architecture that utilizes a small, highly-optimized set of instructions, rather than a more specialized set of instructions often found in other types of architectures.

A RISC computer has a small set of simple and general instructions, rather than a large set of complex and specialized ones. The main distinguishing feature of RISC is that the instruction set is optimized for a highly regular instruction pipeline flow. Another common RISC trait is their load/store architecture, in which memory is accessed through specific instructions rather than as a part of most instructions.
RISC processors only use simple instructions that can be executed within one clock cycle. Thus, the "MUL" command described above could be divided into three separate commands:

a. "LOAD," which moves data from the memory bank to a register,

b. "PROD," which finds the product of two operands located within the registers,
c. "STORE," which moves data from a register to the memory banks.
d. In order to perform the exact series of steps described in the CISC approach, a programmer would need to code four lines of assembly:

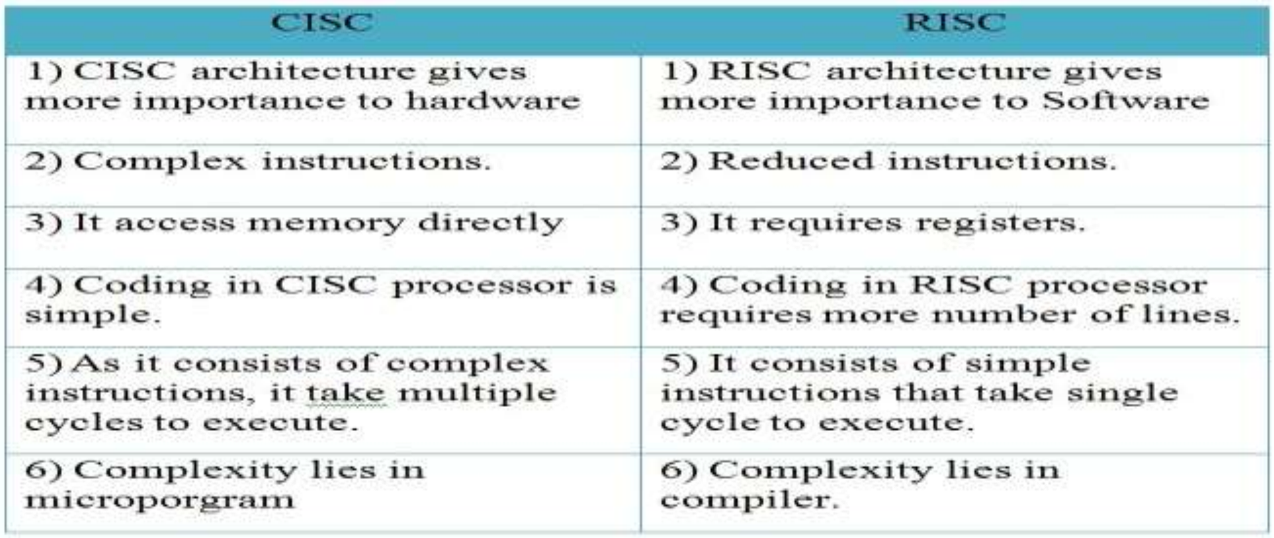
```
LOAD A, 2:3
LOAD B, 5:2
PROD A, B
STORE 2:3, A
```

CHARACTERISTICS OF REDUCED INSTRUCTION SET
Simpler instruction, hence simple instruction decoding.
Instruction come under size of one word.
Instruction take single clock cycle to get executed.
More number of general purpose register.
Simple Addressing Modes.

COMPLEX INSTRUCTION SET COMPUTER (CISC)

A complex instruction set computer is a computer in which single instructions can execute several low-level operations or are capable of multi-step operations or addressing modes within single instructions. The “best” way to design a CPU has been a subject of debate: should the low-level commands be longer and powerful, using less individual instructions to perform a complex task (CISC), or should the commands be shorter and simpler, requiring more individual instructions to perform a complex task (RISC). The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible. 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.
CHARACTERISTICS OF COMPLEX INSTRUCTION SET (CISC)
Complex instruction, hence complex instruction decoding.
Instruction are larger than one word size.
Instruction may take more than single clock cycle to get executed.
Less number of general purpose register as operation get performed in memory itself.
Complex Addressing Modes.

Comparison of CISC and RISC Architectures



CISC	RISC
1) CISC architecture gives more importance to hardware	1) RISC architecture gives more importance to Software
2) Complex instructions.	2) Reduced instructions.
3) It access memory directly	3) It requires registers.
4) Coding in CISC processor is simple.	4) Coding in RISC processor requires more number of lines.
5) As it consists of complex instructions, it take multiple cycles to execute.	5) It consists of simple instructions that take single cycle to execute.
6) Complexity lies in microporgram	6) Complexity lies in compiler.

Applications of RISC and CISC
RISC is used in high-end applications like video processing, telecommunications and image processing. CISC is used in low-end applications such as security systems, home automation, etc.
From the above comparison of RISC and CISC, finally, we can conclude that we cannot distinguish between RISC and CISC technology because both are apt at its precise application. Today, both RISC and CISC designers are doing all to get an edge on the competition. We hope that you have got a better understanding of this concept.

Day 4
To be presented by group F, B & G

REGISTERS

Registers are a type of computer memory used to quickly accept, store, and transfer data and instructions that are being used immediately by the CPU. The registers used by the CPU are often termed as Processor registers.

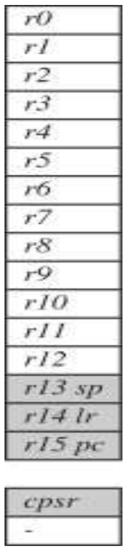
A processor register may hold an instruction, a storage address, or any data (such as bit sequence or individual characters).

The computer needs processor registers for manipulating data and a register for holding a memory address. The register holding the memory location is used to calculate the address of the next instruction after the execution of the current instruction is completed.

A register must be large enough to hold an instruction i.e in a 64-bits computer, a register must be 64 bits in length.

GENERAL PURPOSE REGISTERS

General-purpose registers hold either data or an address (memory location address). They are identified with the letter r prefixed to the register number. For example, register 4 is given the label r4. Figure below shows the active registers available in user mode—a protected mode normally used when executing applications. The processor can operate in seven different modes, which we will introduce shortly. All the registers shown are 32 bits in size.



SEGMENT REGISTERS

Segments are specific areas defined in a program for containing data, code and stack. There are three main segments −

Code Segment − It contains all the instructions to be executed. A 16-bit Code Segment register or CS register stores the starting address of the code segment.

Data Segment − It contains data, constants and work areas. A 16-bit Data Segment register or DS register stores the starting address of the data segment.

Stack Segment − It contains data and return addresses of procedures or subroutines. It is implemented as a 'stack' data structure. The Stack Segment register or SS register stores the starting address of the stack.

Apart from the DS, CS and SS registers, there are other extra segment registers - ES (extra segment), FS and GS, which provide additional segments for storing data.

In assembly programming, a program needs to access the memory locations. All memory locations within a segment are relative to the starting address of the segment. A segment begins in an address evenly divisible by 16 or hexadecimal 10. So, the rightmost hex digit in all such memory addresses is 0, which is not generally stored in the segment registers.

The segment registers stores the starting addresses of a segment. To get the exact location of data or instruction within a segment, an offset value (or displacement) is required. To reference any memory location in a segment, the processor combines the segment address in the segment register with the offset value of the location.

SPECIAL REGISTERS

A Special Function Register (or Special Purpose Register, or simply Special Register) is a register within a microprocessor, which controls or monitors various aspects of the microprocessor's function. Depending on the processor architecture, this can include, but is not limited to:

- I/O and peripheral control (such as serial ports or general-purpose IOs)
- timers

- stack pointer
- stack limit (to prevent overflows)
- program counter
- subroutine return address
- Processor status (servicing an interrupt, running in protected mode, etc.)
- condition codes (result of previous comparisons)

  Because special registers are closely tied to some special function or status of the processor, they might not be directly writable by normal instructions (such as adds, moves, etc.). Instead, some special registers in some processor architectures require special instructions to modify them. For example, the program counter is not directly writable in many processor architectures. Instead, the programmer uses instructions such as return from subroutine, jump, or branch to modify the program counter. For another example, the condition code register might not directly writable, instead being updated only by compare instructions.

The 80x86 Addressing Modes

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

When Intel designed the original 8086 processor, they provided it with a flexible, though limited, set of memory addressing modes. Intel added several new addressing modes when it introduced the 80386 microprocessor. Note that the 80386 retained all the modes of the previous processors; the new models are just an added bonus. If you need to write code that works on 80286 and earlier processors, you will not be able to take advantage of these new modes. However, if you intend to run your code on 80386sx or higher processors, you can use these new modes. Since many programmers still need to write programs that run on 80286 and earlier machines, it's important to separate the discussion of these two sets of addressing modes to avoid confusing them.

Addressing modes with register operands

Register operands refer to data stored in registers. The following examples show typical register operands:

```
mov    bx, 10       ; Load constant to BX
 add   ax, bx       ; Add BX to AX
 jmp   di           ; Jump to the address in DI
```

An offset stored in a base or index register often serves as a pointer into memory.
You can store an offset in one of the base or index registers, then use the register as an indirect memory operand. For example:

```
mov    [bx], dl ; Store DL in indirect memory operand
inc    bx      ; Increment register operand
mov    [bx], dl ; Store DL in new indirect memory operand
```

This example moves the value in DL twice to 2 consecutive bytes of a memory location pointed to by BX.
Example shows that changing BX register causes it to point to a different location in memory.

Addressing modes with constants

Immediate addressing mode is used to specify a constant operand. To do this, the instruction should reference:

- Source register r0, and
- Use As=3

The following code assumes that the data segment begins at address 0x1000, and the text segment begins at 0x2000.

```
 short i;          .data
               i:        .word          1000:    (unspecified)
 i = 3;            .text
                   mov #3, &i           2000: 40d2 0003 1000
```

Example: Constant values as source operand

Generally called immediate mode

- Constant stored immediately following instruction
- After instruction fetch, R0 (PC) references next instruction
- Source addressing mode 3: Indirect auto-increment
    - Fetch operand indirectly (stored in address referenced by R0)
    - Automatically increment R0 to next instruction (always increments by 2)

mov #0xdead, r6   ;; move the hexadecimal constant 0xdead into register 6
Is encoded as the following two words

opcode	src	mode	dest	immediate operand
4	0	3	6	Dead
0100	0000	0-0-11	0110	1101-1110-1010-1101

* 4: move instruction
* 0 source register (in indirect auto-increment mode, see below)
* 3: mode
** (0) desination register is in direct mode;
** (0) word (not byte)
** (3) source register is in indirect auto-increment mode
* 6: destination register (r6)
* dead: integer constant 0xdead

For this operation, the integer constant 0xdead is described as an immediate operand since its value, which immediately follows the instruction, is used as an operand for the specified operation.

Example: Immediate Mode
add.b #0x10,r5 ;;add 16 to low byte of r5
Is encoded as following:

5	0	7	5	0010
0101	0000	0-1-11	0101	0000-0000-0001-0000

5075 0010
*5: add instruction
*0: r0
*7: 0-Ad, 1-byte, 3-As
*5: r5

Addressing modes with memory operands

Source and destination operands in memory are referenced by means of a segment selector and an offset. On embedded operating systems, the segment selector often results in a base address of zero, particularly if virtual memory is used, so the memory address specified by the operand degenerates to being the offset value. The segment selector is automatically chosen by the processor, but can be overridden if needed. The following instruction moves the value in EAX to the address pointed by EBX, assuming the data segment selector contains zero. It is the simplest memory operand form.
MOV [EBX], EAX
The memory operand can also specify offsets to the base address specified in the memory operand. The offset is added to the base address (the general-purpose register) and can be made up from one or more of the following components:
Displacement—An 8-, 16-, or 32-bit immediate value.
Index—A value in a general-purpose register.
Scale factor—A value of 2, 4, or 8 that is multiplied by the index value.
So we have a memory operand that can consist of since the segment selector usually returns zero, the memory operand effective address becomes the following:
The compiler will make best use of these modes to de-reference data structures in memory or on the stack. The components of the offsets can be either positive or negative (two's complement values), providing excellent flexibility in the memory operand address generation.

Addressing modes with stack memory

Stack memory is a memory usage mechanism that allows the system memory to be used as temporary data storage that behaves as a first-in-last-out buffer. One of the essential elements of stack memory operation is a register called the Stack Pointer. The stack pointer indicates where the current stack memory location is, and is adjusted automatically each time a stack operation is carried out.
In common terms, storing data to the stack is called pushing (using the PUSH instruction) and restoring data from the stack is called popping (using the POP instruction). Depending on processor architecture, some processors perform storing of new data to stack memory using incremental address indexing and some use decrement address indexing. In the Cortex-M processors, the stack operation is based on a "full-descending" stack model. This

means the stack pointer always points to the last filled data in the stack memory, and the stack pointer predecrements for each new data store (PUSH)

PUSH and POP are commonly used at the beginning and at the end of a function or subroutine. At the beginning of a function, the current contents of the registers used by the calling program are stored onto the stack memory using PUSH operations, and at the end of the function, the data on the stack memory is restored to the registers using POP operations. Typically, each register PUSH operation should have a corresponding register POP operation; otherwise the stack pointer will not be able to restore registers to their original values. This can result in unpredictable behaviors, for example, function return to incorrect addresses.

The minimum data size to be transferred for each push and pop operations is one word (32-bit) and multiple registers can be pushed or popped in one instruction. The stack memory accesses in the Cortex-M processors are designed to be always word aligned (address values must be a multiple of 4, for example, 0x0, 0x4, 0x8,…) as this gives the best efficiency for minimum design complexity. For this reason, bit [1:0] of both stack pointers in the Cortex-M processors are hardwired to zeros and read as zeros.

In programming, the stack pointer can be accessed as either R13 or SP in the program codes. Depending on the processor state and the CONTROL register value, the stack pointer accessed can either be the MSP or the PSP. In many simple applications, only one stack pointer is needed and by default the MSP is used. The PSP is usually only required when an OS is used in the embedded application.

In a typical embedded application with an OS, the OS kernel uses the MSP and the application processes use the PSP. This allows the stack for the kernel to be separate from stack memory for the application processes. This allows the OS to carry out context switching quickly (switching from execution of one application process to another). Also, since exception handlers only use main stack, each of the stack spaces allocated to application tasks do not need to reserve space needed for exception handler, thus allow better memory usage efficiency.

Even though the OS kernel only uses the MSP as its stack pointer, it can still access the value in PSP by using special register access instructions (MRS and MSR)

Processor state	CONTROL[1] = 0 (default setting)	CONTROL[1] = 1 (OS has started)
Thread mode	Use MSP (R13 is MSP)	Use PSP (R13 is PSP)
Handler mode	Use MSP (R13 is MSP)	Use MSP (R13 is MSP)

Since the stack grows downward (full-descending), it is common for the initial value of the stack pointer to be set to the upper boundary of SRAM. For example, if the SRAM memory range is from 0x20000000 to 0x20007FFF, we can start the stack pointer at 0x20008000. In this case, the first stack PUSH will take place at address 0x20007FFC, the top word of the SRAM

The initial value of MSP is stored at the beginning of the program memory. Here we will find the exception vector table, which is introduced in the next section. The initial value of PSP is undefined, and therefore the PSP must be initialized by software before using it.

In many software development environments, the stack pointer can be set up again during the C start-up code (before entering "main ()"). This two-stage stack initialization sequence enables a system to boot up the system with the stack pointing to a small internal SRAM inside the chip, and then change the stack definition to a larger external memory space after the external memory controller has been initialized.

Instruction set

The instruction set is part of a computer that pertains to programming, which is basically machine language. The instruction set provides commands to the processor, to tell it what it needs to do. The instruction set consists of addressing modes, instructions, native data types, registers, memory architecture, interrupt, and exception handling, and external I/O. Example of an instruction set is the x86 instruction set, which is common to find on computers today. An instruction set can be built into the hardware of the processor, or it can be emulated in software, using an interpreter. The hardware design is more efficient and faster for running programs than the emulated software version.

Examples of Instruction set.

1)ADD - Add two numbers together

2) COMPARE - Compare numbers.

3). IN - Input information from a device, e.g., keyboard.

4). JUMP - Jump to designated RAM address.

5). JUMP IF - Conditional statement that jumps to a designated RAM address.

6). LOAD - Load information from RAM to the CPU.

7). OUT - Output information to device, e.g., monitor.

8). STORE - Store information to RAM.

x86

x86 is an Intel CPU architecture that originated with the 16-bit 8086 processor in 1978. Today, the term "x86" is used generally to refer to any 32-bit processor compatible with the x86 instruction set. The term may also be used to differentiate 32-bit hardware and software from their 64-bit counterparts in Windows PCs but it generally refers to the world's predominant personal computer CPU platform

To really understand what an x86 means, then you need to know that the x86 is an Intel, and intel is a microprocessor. So what is a processor then: processor is the logic circuitry that responds to and processes the basic instructions that drive a computer? The CPU is seen as the main and most crucial integrated circuitry (IC) chip in a computer, as it is responsible for interpreting most of computers commands. CPUs will perform most basic arithmetic, logic and I/O operations, as well as allocate commands for other chips and components running in a computer.

The control transfer instructions

The control transfer instructions control the flow of program Executive. It is categorized as conditional and unconditional control transfer instructions to direct the flow of execution. Conditional control transfers depend on the results of operations that affect the flag register. Unconditional control transfers are always executed.

Unconditional Transfer Instructions

JMP, CALL, RET, INT and IRET instructions transfer control from one code segment location to another. These locations can be within the same code segment (near control transfers) or in different code segments (far control transfers). If the model of memory organization used in a particular x86 application does not make segments visible to applications programmers, intersegment control transfers will not be used.

1). JMP (Jump) unconditionally transfers control to the target location. JMP is a one-way transfer of execution; it does not save a return address on the stack.

2). Call Instruction
CALL (Call Procedure) activates an out-of-line procedure, saving on the stack the address of the instruction following the CALL for later use by a RET (Return) instruction. CALL places the current value of EIP on the stack. The RET instruction in the called procedure uses this address to transfer control back to the calling program.
Return and Return-From-Interrupt Instruction
RET (Return from Procedure) terminates the execution of a procedure and transfers control through a back-link on the stack to the program that originally invoked the procedure. RET restores the value of EIP that was saved on the stack by the previous CALL instruction. RET instructions may optionally specify an immediate operand. By adding this constant to the new top-of-stack pointer, RET effectively removes any arguments that the calling program pushed on the stack before the execution of the CALL instruction.
IRET (Return from Interrupt) returns control to an interrupted procedure. IRET differs from RET in that it also pops the flags from the stack into the flags register. The flags are stored on the stack by the interrupt mechanism.
Interface is the path for communication between two components. Interfacing is of two types, memory interfacing and I/O interfacing.
Memory Interfacing : When we are executing any instruction, we need the microprocessor to access the memory for reading instruction codes and the data stored in the memory. For this, both the memory and the microprocessor requires some signals to read from and write to registers.
The interfacing process includes some key factors to match with the memory requirements and microprocessor signals. The interfacing circuit therefore should be designed in such a way that it matches the memory signal requirements with the signals of the microprocessor.

IO Interfacing
There are various communication devices like the keyboard, mouse, printer, etc. So, we need to interface the keyboard and other devices with the microprocessor by using latches and buffers. This type of interfacing is known as I/O interfacing.
Ways of Communication:
There are two ways of communication in which the microprocessor can connect with the outside world.
a). Serial Communication Interface
b). Parallel Communication interface
Serial Communication Interface − In this type of communication, the interface gets a single byte of data from the microprocessor and sends it bit by bit to the other system serially and vice-a-versa.
Parallel Communication Interface − In this type of communication, the interface gets a byte of data from the microprocessor and sends it bit by bit to the other systems in simultaneous (or) parallel fashion and vice-a-versa.

Macros

A Macro is a set of instructions grouped under a single unit. It is another method for implementing modular programming in the 8086 microprocessors (The first one was using Procedures). The Macro is different from the Procedure in a way that unlike calling and returning the control as in procedures, the processor generates the code in the program every time whenever and wherever a call to the Macro is made. A Macro can be defined

in a program using the following assembler directives: MACRO (used after the name of Macro before starting the body of the Macro) and ENDM (at the end of the Macro). All the instructions that belong to the Macro lie within these two assembler directives.
It is optional to pass the parameters in the Macro. If you want to pass them to your macros, you can simply mention them all in the very first statement of the Macro just after the directive:
The advantage of using Macro is that it avoids the overhead time involved in calling and returning (as in the procedures). Therefore, the execution of Macros is faster as compared to procedures. Another advantage is that there is no need for accessing stack or providing any separate memory to it for storing and returning the address locations while shifting the processor controls in the program.
But it should be noted that every time you call a macro, the assembler of the microprocessor places the entire set of Macro instructions in the mainline program from where the call to Macro is being made. This is known as Macro expansion. Due to this, the program code (which uses Macros) takes more memory space than the code which uses procedures for implementing the same task using the same set of instructions.
Hence, it is better to use Macros where we have small instruction sets containing less number of instructions to execute.

Day 5
To be presented by group E & C
ASSEMBLY LANGUAGE

Assembly language is a low-level programming language for a computer, or other programmable device specific to a particular computer architecture in contrast to most highlevel programming languages, which are generally portable across multiple systems. Assembly language is converted into executable machine code by a utility program referred to as an assembler like NASM, MASM etc.
Some advantage Assembly language includes.

- It allows complex Jobs to be done in a simpler way.
- It is memory efficient, as it requires less memory space.
- It is not required to keep track of memory location.
- It is low level embedded system.
- It is suitable for time-critical jobs.

Some disadvantage Assembly language includes.

- It is very complex and difficult to understand.
- The syntax is difficult to remember.
- It has lack of portability of programs between two computer architectures.

ASSEMBLERS
These are used to translate the assembly language into the machine language
The examples of assemblers include:
ARM;
A machine language encodes instructions as sequences of 0's and 1's; this binary encoding is what the computer's processor is built to execute. Writing programs using this encoding is unwieldy for human programmers, though. Thus, when programmers want to dictate the

precise instructions that the computer is to perform, they use an assembly language, which allows instructions to be written in textual form. An assembler translates a file containing assembly language code into the corresponding machine language.

MIPS;
MIPS assembly language simply refers to the assembly language of the MIPS processor. The term MIPS is an acronym which stands for Microprocessor without Interlocked Pipeline Stages, and it is a reduced-instruction set architecture which was developed by an organization called MIPS Technologies.
The MIPS assembly language is a very useful language to learn because many embedded systems run on the MIPS processor, and knowing how to code in the MIPS assembly language can bring about a deeper understanding of how these systems operate on a lower level.
x84;
This is a family of backward-compatible assembly languages, which provide some level of compatibility all the way back to the Intel 8008 introduced in April 1972. x86 assembly languages are used to produce object code for the x86 class of processors. Like all assembly languages, it uses short mnemonics to represent the fundamental instructions that the CPU in a computer can understand and follow. Compilers sometimes produce assembly code as an intermediate step when translating a high level program into machine code. Regarded as a programming language, assembly coding is machine-specific and low level. Assembly languages are more typically used for detailed and time critical applications such as small real-time embedded systems or operating system kernels and device drivers.

LINKERS

In computing, a linker or link editor is a computer system program that takes one or more object files generated by a complier or an and combines them into a single executable file, library file, or another 'object' file.
A simpler version that writes its output directly to memory is called the loader, though loading is typically considered a separate process.
Linkers can take objects from a collection called a library or runtime library. Most linkers do not include the whole library in the output; they include only the files that are referenced by other object files or libraries. Library linking may thus be an iterative process, with some referenced modules requiring additional modules to be linked, and so on. Libraries exist for diverse purposes, and one or more system libraries are usually linked in by default.
The linker also takes care of arranging the objects in a program's address space. This may involve relocating code that assumes a specific base address into another base. Since a compiler seldom knows where an object will reside, it often assumes a fixed base location (for example, zero). Relocating machine code may involve re-targeting of absolute jumps, loads and stores.
The executable output by the linker may need another relocation pass when it is finally loaded into memory (just before execution). This pass is usually omitted on hardware offering virtual memory: every program is put into its own address space, so there is no conflict even if all programs load at the same base address. This pass may also be omitted if the executable is a position independent executable.

Job Control Language

Job control language (JCL) is a set of statements that you code to tell the z/O S operating about the work you want to perform. Although this set of statements is quite large, most jobs can be run using a very small subset.

NB; z/O S is a 64-bit operating system for IBM mainframes, produced by IBM. It derives from

And is the successor to OS/390, which in turn followed as tringo fMVS versions. Like OS/390, z/O S combines a number of formerly separate, related products, some of which are still optional.

The main purpose of Job Control Language (JCL) is a name for scripting languages used on IBM mainframe operating systems to instruct the system on how to run a batch job or start a sub system.

JCL statement stell z/O S where to find the appropriate input, how to process that input (that is what program or programs to run), and what to do with the resulting output.

Job control language (JCL) is a scripting language executed on an IBM mainframe operating system. It consists of control statements that design a tea specific job for the operating system.

JCL provides a means of communication between the application programs, operating system and system hardware. JCL is considered to be one of the rude script languages run on IBM

OS/360 batch systems. It can define data set names, parameters and system output devices. One common feature in both DOS and OS JCL is the unit of work, which is called a job. A job

Consists of several small steps for running a specific program and is identified by cards called

Job cards, which indicate the beginning of the job and define exactly how the job is to be executed. Both DOS and OS operating systems use 71 characters per line. However, the maximum length is 80 characters. Characters 73-80 are used for locating the error are as reported by the OS.

When a JCL statement becomes too lengthy and exceeds the 71- character limit, it can be Extended using a continuation card. A statement can be continued to as many cards as necessary by ending all the JCL cards excluding the last card a tan instance where a comma is used, or by using(//) at the start of the continuation card in column one and using at least a one -space character.

THE JOB STATEMENT

The JOB statement is the first control statement in a job. It marks the beginning of a job and also specifies the name of the job. The JOB statement also might provide details and parameters that apply to all job steps within the job, such as accounting information and conditions for job termination.

This JCL example contain some JOB statement:

```
//JOB NUM 1 JOB 504, SMITH  PAY ROLL
//STEP 1  EXECPGM=PROGRAM 1
//DD 1 DD DSN=HLQ.OUTPUT
//
```

The name field contains the job name "JOB NUM1". In every JOB statement, the name field contains a one- through eight-character name that identifies the job so that other JCL statements or the operating system can refer to it. Be sure to assign a unique name for each job.

The parameter field defines information that applies to the entire job, contains an accounting number (504) and the programmer's name (SMITH). These parameters are positional and must appear in the order shown.

The comment field contains PAYROLL.

The end of a job is indicated by a null statement, which consists of only two forward Slashes (//), or is marked by the beginning of another JOB statement. In this sample, JOB NUM1 ends with a null statement.

JCL JOB statements are divided into Positional and frequently used parameters

In addition to the two positional parameters (job accounting information and Programmer name), the JOB statement also may contain over 20 keyword parameters.

CLASS

Use the CLASS parameter if your company uses classes to group jobs.

Grouping jobs helps to:

- Achieve a balance between different types of jobs. A good balance of job
- Class assignments helps to make the most efficient use possible of the system.

TIME

Use the TIME parameter to specify the maximum amount of time that a job may use the processor or to find out through messages how much

Processor time a job used. Using the TIME parameter prevents an err or in

Your program from causing it to run longer than necessary.

You can use the TIME parameter on a JOB statement to decrease the

Amount of processor time available to a job or job step below the default value. You cannot use the TIME parameter on a JOB statement to increase the amount of time available.

MSG LEVEL

The MSG LEVEL parameter control show the JCL, allocation messages, and

Termination messages are printed in the job's output listing (SYSOUT).

The MSG LEVEL parameter value consists of two sub parameters:

Statement

The statement sub parameter indicates which job control statements the system is to print on the job log.

Messages

The messages sub parameter indicates which messages the system is to

Print on the job log.

MSG CLASS

You can use the MSG CLASS keyword parameter to assign an output class

For your output listing (SYSOUT). Output classes are defined by the installation to design ate unit record devices, such as printers.

THE EXEC STATEMENT

The EXEC statement marks the beginning of a step within a job, and specifies the name of a program or cataloged procedure to be run.

Procedures are named collections of partial JCL, usually one or more EXEC

Statements and data definition (DD) statements that perform frequently used functions such as sorting data. Procedures are often called procs.

Programs and cataloged procedures are stored in specific datasets, which are called program or procedure libraries, respectively. This JCL example contains only one EXEC statement (and therefore, only one job step).

```
//JOBNUM1JOB504,SMITH PAYROLL
//STEP1 EXECPGM=PROGRAM1
//DD1 DD DSN=HLQ.INPUT
//
```

In this EXEC statement:

The name field contains the step name "STEP1".A step name is a one througheight-character name that identifies the job step so that other JCL Statements or the operating system can refer to it. The parameter field contains the positional parameter PGM, which identifies the program to be run (PROGRAM1).Also, the sample includes a DD statement that identifies the input dataset,

HLQ. INPUT, for the program. The JCL for a job step often contains several associated DD statements that define the program or procedure uses for input or output.

THE DD STATEMENT

Data definition (DD) statements define the datasets that a program or procedure uses when it runs. You must code one DD statement for each dataset that is used or

Created within a job step.

The order of DD statements within a jobstep is not usually significant.

This JCL example illustrates the format of a DD statement:

```
//PAY DD DSN=HLQ. PAY DS, DISP=NEW VENDOR PAYROLL
```

The name field contains a one-througheight- character name, known as a dd name, that identifies the DD statements that other JCL statements, programs, procedures, or the operating system can refer to it. The dd name of this DD statement is PAY.

The parameter field contains only two keyword parameters:

DSN, which is an accepted abbreviation for the parameter DS NAME, which identifies the real name of a dataset.

DISP, which identifies the dataset HLQ .PAYDS as a new dataset; that the system create when this job is submitted for processing.

The comment field contains the phrase VENDORPAYROLL.

A DD statement describes a dataset extensively, and can include the following information:

- The name that the program uses to refer to the dataset, known as the dd name
- The actual name of the dataset and its location
- Physical characteristics of the dataset, such as record format
- The initial and final status of the dataset, known as its disposition

You also can use DD statements to request I/O devices or specify storage allocation for New datasets.