Module 1

Components of a Computer System:	
Hardware:	Physical components that can be seen or touched. Hardware affects the correctness and performance of the programs. CPU, GPU, APU...
Software	the stuff that controls the hardware
Computer	Electronic device that process data converting it into useful information. Used for converting a set of inputs into outputs
Steps to Start a Computer System	turn on- electrical signal sent to CPU- CPU starts executing instructions from a particular fixed address in the memory- instructions are executed one by one- instruction cycle, or the fetch execution cycle repeats forever.
Classes of Computers	Personal Computers, Server Computers, Embedded Computers, Super Computers
Signal	a transmission of data
Analog Waveform	ancient tech, involves using two conductors for each line (send and receive).

Module 1 (cont)

Digital Waveforms	discrete waveform. Represented by two possible voltages on a wire (on or off). We use binary because the technology is not advanced enough for a switch to reliably hold more than two possible states.
Physical Signals	Not discrete and continuous.

Module 1

Software	
Data / Information	all information is stored in the form of binary digits. Ex. a 1 or a 0 in a sequence of 8 bits
ASCII Code	is a set of 8 bits assigned together to. represent data. For each character, a unique byte-sized integer is assigned.
File	A collection of data. Text file can be user program written in any computer language or it can be just numbers and other facts
Binary File	collection of characters in only machine-readable form. Commonly used for the computer to read and execute. Ex operating system

Only thing that distinguishes different types of data is the context in which we view them

Software Categories	
Application Software	to perform a specific application
System Software	which is required to operate a hardware. Ex. operating system, computer languages, utility software etc...

Module 1 (cont)

Operating System	It is not possible for any computers to work without an operating system. Popular operating systems are Linux/Unix, Windows, iOS etc..The main bridge between the hardware and the user is the operating system
Processes	A process is the operating systems abstract for a running program. Multiple processes can run on the system, even on a single CPU core. The instructions for one process is interleaved with the instructions for another process, using context switching.
Threads	A process can consist of multiple execution units, called threads or lightweight processes, each running in the context of the process and sharing the same code and global variables, but different local variables. Multiple tabs of a browser are the threads of the browser process.

Module 1 (cont)

Process vs Threads	a process is an executing instance of a program, also termed "task". Always stored in the main memory, its an active entity; all processed are closed when the computer system is restarted. One program may consist of several processed and multiple processes can be executed in parallel, whenever possible. Ex. the MS Word software running a process. Thread is a subset if a process or a light weight process. The main difference is that threads execute with the context of a process and share the same resources allotted to the process by the kernal. Multiple threads leads to true parallelism, possible on multiprocessor systems. It works on the principle that all the threads running within a process share the same address space, file descriptors, stack, and other related attributes. Ex. in Word, when we type something, it automatically saves. SO editing and saving are hapeing in parallel in two threads

Module 1

Registers

Module 1 (cont)

General Purpose Registers	The x86 architecture contains eight 32-bit General Purpose Registers (GPRs). These registers are mainly used to perform address calculations, arithmetic, and logical calculations. Four of the GPRs can be treated as a 32-bit quantity, a 16-bit quantity or as two 8-bit quantities. They are the EAX, EBX, ECX, EDX. **R-registers Ex. RAX, RBX are just the 64-bit version of the E general purpose registers**
EFLAGS	are status registers which monitor the results produced from the execution of arithmetic instructions and then perform specific tasks based on the status report.
Sequence of Operations	Only one instruction in the main memory, which is pointed to by the PC, is read (called fetched) by the CPU and executed. This is called the instruction cycle. PC is increased after the instruction fetch so that the next instruction is pointed by PC and read by CPU.

Module 1

Programs and Compilation Systems	A high level C program is translated to a low-level machine language instruction, packaged into a form called an executable object program, and stored as a binary file. Object programs are also referred to as object files. A translator Ie, a Compiler or an interpreter is used to translate high-level language program to the machine language for execution.

Module 1 (cont)

Phases

Preprocessing Phase	The preprocessor (cpp) modifies the original C program according to directives that begins with the # character.
Compilation Phase	The compiler (cc1) translates the text file *hello.i* into the text file *hello.s* which contains an assembly language program. Each statement in an assembly language program exactly describes one low level machine language instruction in a standard text form. Assembly language is useful because it provides a common output language for different compilers for different high-level languages.
Assembly Phase	The assembler translates *hello.s* into machine language instruction, packages then into a form known as a relocatable object program and stored the result in the object file *hello.o*. The *hello.o* file is a binary file whose bytes are encoded machine language instructions rather than characters

Module 1 (cont)	
Linking Phase	Notice that our hello program calls the printf function, which is part of the standard C library. The printf function resides in a separate precompiled object file called *printf.o*, which must somehow be merged with our *hello.o* program. The linker (ld) handles merging, The result is the hello file, which is an executable object file that is ready to be loaded into memory and executed by the system.
Compilation System	Compilation systems try to produce correct efficient machine codes but list the errors that it could not understand.
Optimizing Program Performance	Not easy for them to optimize source code
Understanding Link-time Errors	Especially in the development of a large software system
Avoiding Security Holes	Run time errors
Memory	

Module 1 (cont)	
Cache Memories (SRAM)	It might take the CPU 10 times longer to read a file from the disk than from the main memory This is why the hello program needs to be loaded into memory before the file is executed by the CPU so that it may be processed faster.
Need for Cache Memory	Since fetch time is much longer than execution time, its a good idea to solve the processor-memory gap by the introduction of the cache memory or the CPU memory. Cache memory is very high-speed semi conductor memory which can speed up CPU, acting as a buffer between the CPU and main memory. It can also hold those parts of DATA and PROGRAM, which are most frequently used by the CPU. Those data and programs are the first transferred from disk to cache memory by the OS and the CPU can access them. It is mostly integrated directly with the CPU chip or it may be placed on a separate chip and can have a separate bus interconnect with the CPU.

Module 1 (cont)	
	The smaller faster and closely located storage device is called cache memory or simply cache. The cache is helpful as it speeds up the execution of the programs. Initially programs will be loaded into the main memory and then stored into cache memory or simply cache. The cache is helpful as it speeds up the execution of the programs. Initially, programs will be loaded into the main memory and then stored into cache memory when execution of the program starts. It can exist in multiple levels Ex. L1, L2, L3 ...
Memory Hierarchy	L0: Registers
	L1; L1 cache
	L2:L2 cache (SRAM)
	L3: L3 cache (SRAM)
	L4: Main Memory (DRAM)
	L5: Local Secondary Storage (Local Disk)
	L6: Remote Secondary Storage / Tertiary Storage (distributed file systems, web servers)

Module 1	
Hardware	
CPU	Central Processing Unit: main brain of the comuter also know as the processor or core. Newer computers have multiple cores.

Module 1 (cont)

GPU	Graphics Processing Unit: made to enhance the creation of images for a computer display, consists of thousands of smaller, more efficient cores designed to handle multiple tasks simultaneously. Main functions are texture mapping, image rotation, translation, and shading.
APU	Accelerated Processing Unit: is the main processor with additional functionalities. APU = CPU + GPU on a single chip
FPGA	Field Programmable Gate Array: is not a processor but is capable of creating one or multiple processors. The programmable hardware is completely separate from the CPU and its used for digital design and can be reconfigured repeatably. Can be used to help design specialized circuits for a specific application and can be modified for others,
I/O	Input / Output: keyboards, scanner, mouse etc..
Main Memory	
Primary Memory	
RAM	Random Access Memory: consists of two types: DRAM and SRAM

Module 1 (cont)

DRAM	Dynamic Random Access Memory: is a type of semi conductor memory where data is stored in the form of a charge. Each memory cell is made up of a transistor and a capacitor. Capacitors loose charge due to leakage, making DRAM volatile; consequently the device must be regularly refreshed to retain data
SRAM	Static Random Access Memory: (Cache): retains a value as long as power is supplied, SRAM is typically faster than DRAM. Each SRAM memory cell is comprised of 6 transistors; the cost per cell of SRAM is more than DRAM
ROM / PROM	non volatile, permanent. The different types are EEPROM and EAPROM
Secondary Memory	Floppy, hard disks
Buses	are cables used to carry data from one component to another. Each bus can transmit a fixed-size bytes known as a word that is of 4-bytes (32 bits) or 8 bytes (64 bits).
Registers	a word sized storage device in the main memory. PC (program counter) is a special register pointing at (contains the address of) some machine-language instruction stored in main memory increased after the fetch cycle.

Module 1

Data Structures	There are several ways memory can be organized and used for data storage:
Program Code and Data	Machine instructions, data to be processed as well as the processed recults
Shared Libraries	The library files (already written and translated programs) that are shared by multiple processes
Heap	Memory allocation as and when is required
Stack	Memory allocated for short term storage
Kernel	Virtual memory: this stores the part of the operating system
Network	A network is a set of hardware devices that are connected together physically or logically so that they may share or exchange information. The internet is an ideal example of a global network, also called a network of networks" The main advantages are; data sharing, connectivity, hardware and software sharing, data security and management, performance enhancement.
Protocol	A protocol is the defined set of rules, algorithms, messages, and other mechanisms that the software and hardware must follow to communicate effectively.
Nodes	Nodes are the connecting hardware on the network

Module 3 A

Elements of a C program	
A C development envionment includes	Systen libraries and headers; a set of standard libraries and their header files. For example see /usr / include and glibc.
	Application Source: application. source and header files.
	Compiler: converts source to object code for a specific function.
	Linker: Resolves external references and produces the executable module.
Why C?	Most system programs are written in C, not even C++, for fast execution. The kernels of most operating systems are written in C. A lot of projects use C
GNU C Compiler	GCC is GNU compiler collection. It is an integrated distribution of compilers for several major programming languages like C, C++, Java, Objective-C, Objective C++ etc...
	GCC also known as GNU Compiler is used for compiling C programs
Some features of GCC are:	language independence-possibility of sharing code among the compilers for all supported languages.

Module 3 A (cont)

	code optimization - Various optimization levels that may generate machine code for various processors.
How to compile	gcc hello.c -o hello
How to run	./a.out
Source and Header Files	
Header Files	(*h) Header files in C are templates that include function prototypes and definitions of variables, types, and macros. By including these files in your code, you're effectively enabling code reusability and modularization, making your code cleaner and easier to manage.
	Do not place source code (i.e. definitions) in the header file with a few exceptions: inline'd code, class definitions and const definitions
Standard Headers you should know	stdio.h : file and console
	stdlib.h: common utility functions: malloc, calloc etc..
	string.h: string and byte manipulation: strlen, strcpy etc...
Malloc	This is a function in C that reserves a specified amount of memory during runtime and returns a pointer to the beginning of the allocated block.

Module 3 A (cont)

Calloc	Similar to malloc, calloc also reserves a certain amount of memory during runtime but in addition, it initializes all the reserved memory to zero and returns a pointer to the start of it.
The Preprocessor	The preprocessor in C is a program that processes your code before it's compiled. It can include header files, define macros, conditionally compile sections of code, and handle other tasks that occur before actual code compilation.
C Preprocessor (cpp)	is used to insert common definitions into source files
	Commands begin with a #. #define(define a macros) #include(insert text from file).
C language	program file names end with ".c"
Primitive Data Types	char, int, short, long, float, double
Format Specifiers	%d: print as a decimal integer; %6d print as decimal int, at least 6 char wide; %6.2 print as floating point, at least 6 char wide and 2 after decimal.
	%o: ocatal; %x: hexa; %c: char; %s:string;%%: % itself
Precedence and Associativity	

Module 3 A (cont)

++ --	Binary operators -- Highest
+ -	Unary operators (negative, positive)
* / %	Math symbols
+ -	Math symbols -- lowest
Symbols Prescedence	
!	HIGHEST
>>=<<=	
== !=	
&&	
||	LOWEST
Comma operator	lowest precedence of all the operators, causes a sequence of operations "do this, then this"
Conditional operator	if-then-else. Exp1 ? Exp2: Exp3
Expression vs Statement	An expression in C is any valid combination of operators, constants, functions and variables. A statement is a valid expression followed by a semi colon;
Arrays	Collection of similar data items identified by a single name and stored in contiguous memory location
2D Arrays	type array_name[row-size][-column-size]
	int a [3][4] = {{0,1,2,3}, {4,5,6,7}, {8,9,10,11}; or int a[3][4]={0,1,2,3,4,5,6,7,8,9,1-0,11}

Module 3 A (cont)

String is a character array	sequence of characters in a character array that is terminated by null character '\0'
	C language does not support strings as a data type
	String is just a one-dimensional array of characters
	char name[10] = "Example Program"; or char name[10] = {'L','e','s','s','o','n','s','\0'}
	The length of the string is the number of bytes preceding the null character. The value of a string is the sequence of the values of the contained characters, in order.
	Series of characters treated as a single unit. String literal (string constant) = written in double quotes "hello";
Functions	Same as a method in Java
	return-type function-name(a-rgument declarations). Various parts may be absent
	Each function logically, should perform a specific task

Module 3 A (cont)

Parameter passing	Call by Value: In C, when you pass values to a function, a new copy of those values is created for the function to use. Changes made to these values in the function do not affect the originals.
	Call by Reference: This method passes the address of the variable to the function. Hence, any changes made to the variables in the function directly alter the original variables as they share the same memory location. Note that C doesn't directly support call by reference, but it can be simulated using pointers.
External Variables	Declared outside any function, usually with initial values
	permanent so they can retain values from one function invocation to the next.
Automatic (Local) Variables	Are internal to the function; they come into existence when the function is entered and disappear when it is left.

Module 3 A (cont)

Static Variables	Static variables in C are variables that retain their values between function calls. Unlike regular local variables, which are created and destroyed every time a function is called, static variables are initialized only once and exist until the program ends. User how is that different than an external varaible? External variables, or global variables, are declared outside any function and can be accessed by any function throughout the program. Unlike static variables, which are only visible within their own function or file, external variables are visible to all parts of the program, making them more universally accessible but also potentially increasing the risk of unintended modifications.
Register Variables	A register declaration advises the compiler that the variable in question will be heavily used. The idea is to keep them in registers which is much quicker to access
Scope Rules	

Module 2

Binary Number System	Base 2
	4 bits = nibble 8 bits = byte
Octal Number	Base 8
Hexadecimal System	Base 16
Decimal to Binary	divide the number by the base (=2). take the remainder (either 0 or 1) as a coefficient. Take the quotient and repeat the division.
	Ex. 13/2 = 6 (quotient), 1 (remainder)
	6/2 = 3 (quotient), 0 remainder
	3/2 = 1 (quotient), 1 (remainder)
	1/2 = 0 (quotient), 1 (remainder)
	13 = 1101 - remainders read from the bottom up.
Data Representation in Words	A word size is the number of bits processed by the computer in one go ie. typically 32 bits or 64 bits.
	Data bus size, instruction size, address size are usually multiples of the word size.
Addressing and Byte Ordering	A variable X of type int (allocated 4 bytes)
	If the address of x: 0x100 (means it starts storing from 0x100)

Module 2 (cont)

	This means the 4 bytes of x would be stored in memory locations 0x100, 0x101, 0x102, and 0x103.
	Lets assume x has the VALUE of 0x1234567, which needs to be stored in four bytes:
	Two conventions to store the values in the 4 consecutive byte memory locations. 0x01, 0x23, 0x45, and 0x67, or 0x67, 0x45, 0x23, and 0x01, depending on the CPU architecture
Little Endian	0x01, 0x23, 0x45, 0x67. **Little first** Refers to the byte order in which the least significant byte (LSB) is stored at the lowest memory address, and the most significant byte (MSB) is stored at the highest memory address.
Big Endian	0x67, 0x45, 0x23, 0x01 **Big first** Refers to Refers to the byte order in which the most significant byte (MSB) is stored at the lowest memory address, and the least significant byte (LSB) is stored at the highest memory address.

Module 2 (cont)

Integer Representation	
char, unsigned char	1 byte
Short, unsigned short	2 bytes
Int, unsigned int	4 bytes
Long, unsigned long	8 bytes
float	4 bytes
double	8 bytes
	max number able to store is bit size / 4
Unsigned	All 8 bits used for data storage, nos sign bit
unsigned char	Smallest number is 0, max number is 0xff
unsigned short	16 bits, smallest number is 0, max number is 0xffff - 65536 in decimal
unsigned int	32 bits, smallest number is 0. max number is 0xffffffff
	The ma number is 2^8 = 255, the min number is 0
unsigned long	64 bits, smallest number is 0, largest number is 0xffffffffffffffff.
Binary Addition	
1 + 1	= 0 carry the 1
0 + 1	= 1
Binary Subtraction	
1 -1	= 0
0 - 1	borrow a base when needed
Binary Multiplication	bit by bit
Representation of Negative Binaries in Memory	**Left bit is called the most significant bit**

Module 2 (cont)

	MSB = 0, non-negative integers, MSB = 1, negative integers
	Other 7 bits (for an int, 8 bits) is used to represent the integers
8 bit signed representation	Largest number is 127
	Smallest number is -128
2's Complement	Positive numbers and zero are represented as usual in binary. Negative numbers are represented by inverting all the bits of their positive counterpart and adding 1 to the result.
	This system allows simple binary addition to work for both positive and negative operands without needing separate subtraction hardware. The leftmost bit often acts as a sign bit, with 0 for non-negative values and 1 for negative values.
Fractional Number Representation	IEEE floating point representation
	Floating point is typically expressed in the scientific notation, with a fraction (F) and an exponent (E) of a certain radix (r), in the form of F x r ^ E
0.1	= 2^-1 = 0.5

Module 2 (cont)

0.01	= 2^-2 = 0.25
0.001	=2 ^ -3 = 0.125
Floats are stored in memory as follows	sign bit 's' = denoting positive or negative - 1 bit
	mantissa 'm' = the digits of your number - 23 bits
	exponent 'e' = 8 bits
Three different cases	
Normalized values	The bit pattern of exp is neither all zeros nor all ones
Denormalized values	It is the case where the exp is all 0's but the fraction is non-zero.
	The denormalized numbers gradually lose their precision as they get smaller because the left bits of the fraction become zeros

Module 2 (cont)

Special values	A final category of values occurs when the exponent field is all ones. When the fraction field is all zeros, the resulting values represent infinity, either +∞ when s = 0, or −∞ when s = 1. Infinity can represent results that overflow, as when we multiply two very large numbers, or when we divide by zero. When the fraction field is nonzero, the resulting value is called a “NaN,” short for “Not a Number.”
ASCII Character codes	Used to represent information sent as character based data
	uses 7 bits to represent 94 graphic printing characters 34 non printing characters
Boolean Expression	Any expression that evaluates to true or false.
Boolean Operators	AND(.), OR(+), NOT(!)

Module 3

Scope Rules	
There are three places where variables can be declared in C	Inside a function or a block which is called local variables
	Outside of all functions which is called global variables

Module 3 (cont)

	In the definition of function parameters which is called formal parameters
Pointers	Pointers in C are like arrows that point to a location in your computer's memory where data is stored. Instead of holding a value themselves, they tell you where to find the value.
	variable that contains the address of another variable

Module 3 (cont)

	Pointers and arrays in C are closely related because arrays are essentially a block of contiguous memory locations. The name of the array is a pointer to the first element of the array. So, if you have an array like int arr[5], you can access its elements using pointers. For example, *(arr + 2) will give you the third element of the array, because the pointer arr points to the start of the array and adding 2 moves the pointer to the third element. This relation gives you another way to access and manipulate arrays, making the use of arrays more flexible in C.
Pointer declaration	int *ptr; Declares a variable ptr that is a pointer to a data item that is an integer
Assignment to a pointer	ptr = &x; Assigns ptr to point to the address where x is stored.
To use the value pointed to by a pointer we use dereference (*)	Given a pointer, we can get the value it points to by using the * operator

Module 3 (cont)

	*ptr is the value at the memory address given by the value of ptr
	Ex. if ptr = &x then y = *ptr + 1 is the same as y = x +1
	If ptr = &y then y = *ptr + 1 is the same as y = y + 1
Structures	A way to have a single name referring to a group of related values.
	Structures provide a way of storing many different values in variables of potentially different types under the same name.
	Structs are useful when a lot of data needs to be grouped together,

Module 4

Types of memory	Primary memory: most part of memory clears out everytime a computer is restarted. This is also called temporary memory or volatile memory except ROM. Ex. RAM, ROM and Cache
	Secondary Memory: Data stored on this memory is permanent and retains even after computer is switched off. Ex. Harddisk, CD, USB

Primary Memeory

Module 4 (cont)

RAM	Random Access Memory: is the most common and accessible memory by the processor to process the data. This is traditionally packaged as a chip. Basic storage unit is normally a cell. Multiple RAM chips may be there on the computer board to form a memory.
Dynamic Ram (DRAM)	More commonly used in memory here each cell stores a bit wit ha capacitor. One transistor is used for accessing the data. Here values must be refreshed every 10-100ms to retain. Slower and cheaper than SRAM but is used as a simple scratch area for different data manipulations.
Static Ram (SRAM)	each cell stored a bit with four or six-transistor circuit. It may retain values indefinitely, as long as it is kept powered. This memory is faster and more expensive than DRAM and used as Cache memory.
Enhanced DRAM	
Synchronous DRAM (SDRAM)	this uses a conventional clock signal instead of asynchronous control

Module 4 (cont)

Double Data-Rate Synchronous DRAM (DDR SDRAM)	Double edge clocking to send two bits per cycle per pin. Different types are there that are distinguished by size of small prefetch buffer like DDR(2 bits), DDR2(4 bits), DDR4(8 bits)
ROM	Read only memory: it isprogrammed during production and can only be programmed once. There is special erasable PROM (EPROM) that can be bulk erased using electrical signals or UV or x-rays. A normal user cannot alter this memory. Main use is to store firmware programs life BIOS, controllers for disks, network cards, graphics accelerators, security subsystems etc...

Module 4 (cont)

Cache Memory	There is a problem of processor - memory bottleneck. If the processor is running at the same speed as the memory bus, both work fine, but since newer computers have high speed CPU's, the CPU will have to wait for information from the memory. Cache stores the info coming from memory and the processor can get the info from the cache must faster. Small fast storage device that acts as a staging area for a subset of the data from the larger, slower device like RAM. Can be multiple levels of cache L1, L2
	Cache Hit: When the ALU needs some data it looks for the data in the cache. If found, it is called a cache hit, else a cache miss
Cost of cache miss	Consider: Cache hit time of 1 cycle. Miss penalty of 100 cycles.
	97% hits: 1 cycle + 0.03 * 100 cycles = 4 cycles
	99% hits: 1 cycle + 0.01 * 100 cycles = 2 cycles

Module 4 (cont)

Miss Rate	Miss rate is the fraction of memory references that are not found in cache. (Misses / Accesses) = 1 hit rate. Typical hit percentages (3% to 10% for L1)
Hit Time	Hit time is the time to deliver a line in the cache to the processor. It also includes time to determine whether the line is in the cache. Typical hit times: 1-2 clock cycles for L1
Miss Penalty	Miss penalty is the additional time required because of a miss is typically 50 - 200 cycles.
Secondary Memory	This is the non-volatile part of the memory used for storing data for long term storage. Ex. Floppy, hard disk, usb
Hard Disk (HDD)	Most important and most commonly used secondary storage medium. This is mostly installed inside the CPU, but may also be an external hard disk that can be portable if required

Module 4 (cont)

	A hard disk consists of platters, each with two surfaces. Each surface consists of concentric rings called tracks. Each track consists of sectors separated by gaps. A sector is the block that is addressed to store a block of information.
Steps for disk access	The reading head is positioned above a track
	Counter clockwise rotation of the disk happens until it reaches the requireds ector
	Reads the data, once reached
Calculate Disk Capacity	We may measure the disk capacity using the following technology factors:
	Recording density (bits/in): number of bits that can be stored into a 1 inch segment of track
	Track Density (tacks/in):the number of tracks that can be squeezed into a 1 inch segment of the radius extending from the centre of the platter.
	Aerial Density (bits/in2): product of the recording density and track density

Module 4 (cont)

Disk Capacity Calculation	Capacity = (#bytes/sector) x (avg.# sectors/track) x (#tracks/surface) x (#surfaces/platter) x (#platters/disk)
Disk Access Time Calculations	
Average Time	Taccess = Tavg seek + Tavg rotation + Tavg transfer
Seek Time (Tavg seek):	Time to position heads over cylinder containing target sector. Typical avg seek is: 2-9ms
Rotaional Latency (Tavg rotation)	Time wasted for first bit of target sector to pass under r/w head. Tavg rotation = 1/2 x 1/RPMs x 60sec/1min . Typical is 720RPMs
Transfer Time (Tavg transfer)	Time to read the bits in the target sector. Tavg transfer = 1/(avg # sectors/track) x 1/RPM x 60secs / 1min
Solid State Disks (SSDs)	Device that uses integrated circuits to store data permanently. Since SSDs do not have mechanical components, these are typically more resistant to physical shock, run silently, have lower access time, and lower latency. More expensive

Module 4 (cont)

Principle of Locality	Programs tend to use data and instructions with addresses near or equal to those they have used recently. Two main localities are Spacial and Temporal
Spacial	Items near by addresses tend to be referenced close together in time
Temporal	Recently referenced items are likely to be references in the near future

Module 5

Linking	you can think of a linker as a person assembling a train set. Each car of the train is a piece of compiled code, and the linker's job is to connect these cars together in the correct order to form a complete train (the final executable program). If one car needs to connect to another in a specific way (such as a function in one piece of code calling another function in another piece), the linker ensures they're hooked together properly, so the entire train runs smoothly. The final result is a complete, operational program that's ready to run on your computer. A process of collecting and combining various pieces of code and data into a single file that can be loaded into the memory and executed. Linking can be done at compile time, or run time.

Module 5 (cont)

Role of Linkers	Symbol Resolution: Symbol definitions are stored (by compiler) in symbol table, an array of structs, in the .o files. Each entry includes name, size, and relative location of symbol. A linker associates each symbol reference with exactly one symbol definition. Symbols will be replaced with their relative locations in the .o files.
	Relocation A linker merges separate code and data sections in the .o files into single sections in the a.out file and relocates symbols from their relative locations in the .o files to their final absolute memory locations in the executable. A linker also updates all references to these symbols to reflect their new positions.:
Why do we use Linkers?	Programs can be written as a collection of smaller source files, rather than one monolithic mass and can build libraries of common funcitons

Module 5 (cont)

Efficiency is Implemented in two ways: 1. Time Efficiency - a separate compilation enables changes to one source file, compile, and then relink, therefore there is no need to recompile other source files again and again
2. Space efficiency: libraries, which are common functions that are aggregated into a single file, yet are still executable files and running memory images, contains only code for the functions they actually use

Module 5 (cont)

Compiler Drivers	think of the compiler driver as the conductor of an orchestra. The musicians in the orchestra each play their instruments, just as different parts of the compilation process (preprocessing, compiling, assembling, linking) handle specific tasks. The conductor coordinates all the musicians to create a harmonious piece of music, just as the compiler driver coordinates the various stages of the compilation process to produce a working executable. It makes sure everything happens in the right order and that all the necessary pieces come together, simplifying the complex process into a single, unified command. A compiler driver invokes the language pre processor, compiler, assembler, and linker as needed on behalf of the user. The name of the compiler driver on our Linux box is **gcc**

Module 5 (cont)

Process	In modular programming many methods/procedures are stored in a separate file. the file is called from the main program to call the methods defined in the external file.
The steps of execution are:	1. The driver first runs the C preprocessor **(cpp)**, which translates the C source file main.c into an ASCII intermediate file **main.i**
	2. Next, the driver runs the C compiler **(cc1)**, which translates **main.i** into ASCII assembly language file **main.s**
	3. then the driver runs the assembler **(as)**, which translates main.s into a relocatable object file **main.o**
	4. The driver goes through the same process to generate swap.o. Finally it runs the linker program **(ld)**, which combines main.o and swap.o, along with the necessary system object files, to create the executable object file.

Module 5 (cont)

Primary Memory Addressing	The primary memory of the computer is used as contiguous array of physical addresses. It has two parts: 1. Low Memory Area: stores resident operating system. 2. High Memory Area: user processes
Virtual Memory	virtual memory enables a computer to use more memory than it physically has by "borrowing" space from the hard drive. When the actual RAM gets full, less-used data is temporarily stored on the hard drive, making room for new data in the physical RAM. This process allows larger and more complex applications to run smoothly, even on systems with limited physical memory.

Module 5 (cont)

Memory Relocation Concept	memory relocation is like rearranging the contents of a bookshelf. If you need to make room for more books or organize them differently, you might shift some of the books to different spots on the shelf. Similarly, memory relocation moves data and code to different parts of the computer's memory to make more efficient use of space or to allow programs to run correctly, even if they weren't originally designed to be placed in those exact spots in memory.
Address Space	Address space is a set of addresses that a process can use to address memory. Each process has been given a unique address space that can be identified with base register and limit register combination. The data is moved between the process address space (Virtual space) and actual physical address for processing. This is called as swapping

Module 5 (cont)

Swapping	Swapping is like moving books between a small reading table (RAM) and large bookshelves (hard drive). If your table is full and you need a new book that's on the shelf, you'll put one book from the table back onto the shelf and take the new book you need. In a computer, when RAM is full and a new program or data needs to be loaded, the operating system puts some of the data or programs that are in RAM but not currently being used onto the hard drive, making space for the new information. This keeps the system running smoothly, even when working with more data than can fit in RAM at one time.

Module 5 (cont)

Linker	think of the linker as a puzzle master putting together a jigsaw puzzle. Each object file is like a section of the puzzle, and the linker's job is to fit them together in the correct way to form the complete picture (the executable program). If one piece refers to another (such as calling a function defined somewhere else), the linker makes sure they're connected properly, so the whole image makes sense. The final result is a complete program that's ready to run on your computer.
Static Linking	Symbol Resolution: Object files define and reference symbols. The purpose of symbol resolution is to associate each symbol reference with exactly one symbol definition

Module 5 (cont)

Relocation: Compilers and assemblers generate code and data sections that start at address 0. The linker relocates these sections by associating a memory location with each symbol definition, and then modifying all of the references to those symbols so that they point to this memory location.

Module 5 (cont)

Dynamic Linking	think of dynamic linking like a toy set with interchangeable parts. When you're playing with the toy, you might want to add special features or accessories, like wheels or wings. Instead of storing all these parts inside the main toy (which would make it big and heavy), you keep them in a separate box and snap them on as needed. In a computer, a dynamically linked program works the same way. It stays small and lightweight because it doesn't include everything inside itself. Instead, it reaches out and grabs the extra parts (like functions or variables) from shared libraries when it needs them, keeping things more efficient and flexible.

Types of Object Files

Module 5 (cont)

Relocatable Object files **(.o)**: Think of this as a puzzle piece that hasn't been fixed to the final picture yet. It contains compiled code, but it has references (like function calls) that aren't tied down to specific addresses. This allows the linker to move it around and fit it with other pieces when creating the final executable. It's a flexible, intermediate step in building a program.

Executable Object File **(a.out)**: This is like the completed puzzle, with all pieces fixed in place, forming a clear picture. An executable object file contains all the code, data, and references properly linked and ready to run on a computer. Everything is set, and it's ready to be launched and executed by the operating system.

Module 5 (cont)

Shared Object File **(.so)**: Imagine a special puzzle piece that can fit into multiple puzzles. A shared object file contains code or data that multiple programs can use simultaneously. Instead of including the same code in every program (which would take up more space), the code is stored in one place, and different programs can reach into it and use what they need. It's a way to share common functions or variables between different programs, making things more efficient.

Compilers and assemblers generate object files (including shared object files). Linkers generate executable object files.

Information in Object File

Header Information: info about the file such as the size of the code, name of the source file it was translated from, and creation date.

Object Code: Binary instructions and data generated by a compiler or assembler

Module 5 (cont)

Relocation: A list of the places in the object code that have to fixed up when the linker changes the addresses of the object code

Symbols:Global symbols defined in this module, symbols to be imported from other modules or defined by the linker.

Debugging Information: Other information about the object code not needed for linking but of use to a debugger. This includes source file and line number information, local symbols, descriptions of data structures used by the object code such as C structure definitions.

Module 5 (cont)

Executable and Linkable Format (ELF)	think of ELF as a standardized container used for shipping various goods. Different containers might be used for different types of goods, but this particular container (ELF) is designed in such a way that it can efficiently pack away different types of programming "goods," like the actual programs you run (executables), the building blocks used to create those programs (object code), or the shared pieces used by many programs (shared libraries). By having this standardized container, the system knows exactly how to handle, load, and run these various components, regardless of what's inside. Just as shipping containers have specific ways they can be lifted, stacked, and transported, ELF files have a specific structure that the operating system understands, allowing it to handle them in a consistent and efficient way.

Module 5 (cont)

ELF Header	Information to parse and interpret the object file; Word size, byte ordering, file type(.o, exec, .so) machine type, etc.
Segment Header table	For runtime execution: Page size, virtual addresses memory segments (sections), segment sizes.
	.text section: Instruction code
	.rodata section: Read only data: jump tables, ...
	.data section: Initialized global variables
	.bss section: Uninitialized global variables; it has a section header but occupies no space
	.symtab section: Symbol table; Procedure and static variable names; Section names and locations are used by a linker for code relocation
	.rel.text section: Is the relocation info for .text section, which addresses instructions that will need to be modified in the executable; Also instructions for modifying.

Module 5 (cont)

.rel.data section: Is the relocation info for .data section; it also addresses of pointer data that will need to be modified in the merged executable

.debug section: Info for symbolic debugging (gcc -g); Section header table used for linking and relocation: Offsets and sizes of each section

Types of ELF Files

Relocatable: files are created by compilers and assemblers but need to be processed by the linker before running

Executable: files have all relocation done and all symbols resolved except perhaps shared library symbols to be resolved at runtime

Shared Object: are shared libraries, containing both symbol information for the linker and directly runnable code for runtime

Symbols and Symbol Tables

Module 5 (cont)

	Global symbol: defined by module m that can be referenced by other modules. For example, non-static C functions and non-static global variables. (Note that static C functions and static global variables cannot be referred from other files.)
	External symbols: Global symbols that are referenced by module m but defined by some other module.
	Local symbols: Symbols defined and referenced exclusively by module m.
Strong and Weak Symbols	
	Strong: procedures and initialized globals
	Weak: uninitialized globals
Linkers Symbol Rules	Rule 1: Multiple strong symbols are not allowed; each item can be defined only once, otherwise the result is a linker error
	Rule 2: Given a strong symbol and multiple weak symbol, choose the strong symbol. References to the weak symbol resolve to the strong symbol.

Module 5 (cont)

	Rule 3: If there are multiple weak symbols, pick an arbitrary one. This can be overridden with gcc –fno-c-ommon
Global Variables;	Avoid global variables if possible, otherwise use static if possible; Initialize if you define a global variable. Use extern if you do use external global variables.
Relocation	Relocation consists of two steps:
	Relocating sections and symbol definitions
	Relocating symbol references within sections
Types of Libraries	
	Static Libraries: Concatenate related relocatable object files into a single file with an index (called an archive). ☐ Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives. ☐ If an archive member file resolves reference, link it into the executable.

Module 5 (cont)

Dynamic / Shared Libraries: Object files that contain code and data that are loaded and linked into an application dynamically, at either load-time or run-time. ☐ Also called: dynamic link libraries, DLLs, .so files ☐ Shared library routines can be shared by multiple processes. ☐ In shared libraries, the symbols for the code in shared libraries will be resolved with absolute addresses at either load-time or run-time.

Module 7

Performance Realities	To write an efficient code, you need to understand the basic facts: a. Constant factors o It is possible to improve the performance of the code, if it is properly written The optimization can be done in various ways: Adopting an appropriate algorithm, 👎 Selecting proper data representations, 👎 Following procedures and loops suitably and accurately

Module 7 (cont)

	The programmer must understand following ways the system to optimize performance: a. The way programs are compiled and executed. b. The way to measure program performance and identify bottlenecks. c. The way to improve performance without destroying code modularity and generality.
Optimizing Compilers	Compiler construction has always been an active research topic. Compiler developers have developed very advanced and optimization compilers that can automatically make the optimized codes by:
	Making proper register allocation ☐ Automatic code selection and ordering ☐ Performing dead code elimination automatically and ☐ Eliminating minor inefficiencies by itself
	Nevertheless, many places may not be optimized by compilers: ☐ Improving asymptotic efficiency ☐ Selecting best overall algorithm
	Compilers cannot overcome “optimization blockers” (this will be explained more fully later): ☐ Memory aliasing ☐ Procedure call side-effects
Limitations of optimizing compilers	The optimization compilers have numerous constraints; for example, they cannot cause any change in program behaviour and cannot change the algorithmic style of the programmers
	The compilers can: a. Often prevent it from making optimizations when that would only affect behaviour under pathological conditions.
	Not change behaviour that may be obvious to the programmer but can be obfuscated by languages and coding styles like data ranges may be more limited than variable types suggest
	Since the analysis is performed only within procedures, the whole-program analysis is too expensive in most cases. Most analysis is based only on static information, as compilers cannot anticipate run-time inputs. The main rule is that when in doubt, the compiler must be conservative and cannot do anything.
Optimizations for Programmers	Common optimizations: ☐ Take the same repeated task out of the loop (Code Motion) ☐ Replace mathematical operations with bitwise/shift operations wherever possible(reduction in strength) ☐ Share common sub-expressions ☐ Do not use functions as the loop condition checker (Optimization Blockers)

Module 7 (cont)	
Code Motion	First step is to reduce frequency with which computations are performed, wherever possible following the rule o it should still produce same result o moving computation code out of loop, that is repeated (n*i ☐repeated)
Reduction in Strength	Replace costly operation with simpler one. Shift operations are easier as compared to multiply and divide ☐ Use Shift operation instead of multiply or divide 16*x --> x << 4 ☐ Utility machine dependent ☐ Depends on cost of multiply or divide instruction o On Intel Nehalem, integer multiply requires 3 CPU cycles ☐ Recognize sequence of products
Share Common Sub-expressions	Reuse portions of expressions and write them once Compilers often not very sophisticated in exploiting arithmetic properties
	To calculate : p = q *r + m; x = q* r + n; It is always more advisable to do: s=q * r; p = s + m; x= s + n;

Module 7 (cont)	
Optimization Blocker #1: Procedure Calls	Procedure calls is an excellent concept of modularity but requires careful attention. The procedure call must not be used for checking the conditions: Please see lower1(). Here the issue is that strlen executed every iteration
How this works:	Strlen is the only way to determine length of string as scans the entire length, looking for null character. ☐ Overall performance of the program: o N calls to strlen o Require times N, N-1, N-2, …, 1 o Overall O(N2) performance
Improving Performance:	Take following steps to make it better: ☐ Move call to strlen outside of loop as function result does not change from one iteration to another ☐ Make the rest of the loop common Lower2() is the improvement function in the above figure

Module 7 (cont)	
Procedure Calls: Optimization blocker for the compiler	The optimization compiler will not be able move strlen out of inner loop, as: ☐ Procedure may have side effects ☐ May alter global state each time called ☐ The function may not return same value for given arguments ☐ May depends on other parts of global state ☐ Procedure lower could interact with strlen
The main reasons are:	Compiler treats procedure call as a black box ☐ Weakens the optimizations near them
Remedies for the programmer:	Use of inline functions o GCC does this with –O2 ☐ Do your own code motion as discussed above
Optimization Blocker #2: Memory Aliasing	Aliasing is the method of referring two different memory references specifying single location. This is very easy to have happen in C as it allows points and pointer arithmetic. This also supports direct access to storage structures.

Module 7 (cont)

	Remedy for the programmer: ☐ Get in habit of introducing local variables o Accumulating within loops o Your way of telling compiler not to check for aliasing
Removing non duplicating data processing	In the following example, code updates b[i] on every iteration. Therefore, we must consider the possibility that these updates will affect program behaviour
Instruction-Level Parallelism	Multiple data can process simultaneously. To understand that, we need a general understanding of modern processor design. As with multi-core systems, the CPU can execute multiple instructions in parallel. In this case, performance will be limited by data dependencies.

Module 7 (cont)

	But here too, some simple transformations can have dramatic performance improvement. These are done by the programmers as compilers often cannot make these transformations and because it is difficult to understand the associativity and distributives in floating-point arithmetic.
Cycles Per Element (CPE)	To see the impact of the optimization, we need to have a defined metrics. The number of cycles per element (CPE), is the measure that assumes the run time, measured in clock cycles, for an array of length n is a function of the form Cn + K where Cn is the CPE.
	It is a convenient way to express performance of program that operates on vectors or lists: Length = n In our case: CPE = cycles per OP T = CPE*n + Overhead CPE is slope of line

Module 7 (cont)

Superscalar Processor	A superscalar processor can issue and execute multiple instructions in one cycle. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.
	The benefit is that without programming effort, superscalar processor can take advantage of the instruction level parallelism that most programs have o Most CPUs since about 1998 are superscalar. o Intel: since Pentium Pro
Loop unrolling	Another way of implementing optimization applied to loops. This reduces the frequency of branches and loop maintenance instructions. The number of iterations is known prior to execution of the loop. Objective is to reduce the total number of times loop runs
Effect of Loop Unrolling	Helps integer multiply below latency bound ☐ Compiler does clever optimization ☐ Others don’t improve as it still has sequential dependency

Module 6

Phases of a Program	User programs in C (code time) [. C file]
	C Compiler (compile time)
	Assembler
	Hardware (run time) [executable file]
The time required to execute a program depends on:	program structure (as weitten in C, for instance)
	The compiler: Set of assembler instructions it translates the C program into
	The hardware implementation: Amount of time required to execute an instruction
	The instruction set architecture (ISA): Set of instructions it makes available to the compiler
ISA Instruction Set Architecture	ISA is used to define: The systems state (eg. registers, memory, program counter)
	The instructions the CPU can execute
	The effect that each of these instructions will have on the system state
ISA	is a protocol used to define the way a computer is used by the machine language programmer or compiler

Module 6 (cont)

The ISA describes the following	Memory model: how memory is accessed and referenced
	instruction format, types and modes - commands to be executed
	operand registers, types, and data addressing - data storage and processing locations
Assembly Language	Assembly Language is an Intermediate Language between absolute Machine code and High Level Language
Advantages include:	Machine code with a better human understanding, ease to write and debug, the use of mnemonics for instructions, and it reserves memory location for data
	High Level Language writes more efficient/optimized programs
Need for Assembly Language	The ability to read and understand assembly code is an important skill

Module 6 (cont)

	We can understand the optimization capabilities of the compiler and analyze the underlying inefficiencies in the code, to understand the function invocation mechanisms, and help ourselves understand how computer systems and operating systems run programs
	The programs written in high level languages usually does not run as fast as assembly language programs, so whenever execution speed is so critical, only assembly language routines can be useful.
	Knowledge of assembly enables the programmer to debug the higher level language code
	Programmers developing compilers must know assembly language
Assembly Programmers view of the System	Registers: fastest memory allocations that are nearest to ALU for processing
	Memory: Part of primary memory, where other data and program code is stored

Module 6 (cont)

	Opcodes: The assembly language commands that process the data using the set of registers
Registers	are small memory areas that are volatile and are used for all memory manipulations.
	There are 8 "general purpose" registers and 1 "instruction pointer" that points to the next instruction execute.
	Of the 8 registers, 6 are commonly used and the remaining two are rarely used.
EAX	Main and most commonly used register. Is is an accumulator like register, where all calculations occur. All systems are also called through the EAX register. Used to store the value returned from a function or as an accumulator to add the values

Module 6 (cont)

EBX	A general purpose register, that does not have a dedicated role. It is used as a base pointer form memory access and also used to store extra pointer or calculation step. Base pointer to the data section
ECX	Counter register for loops and strings. General purpose register but mainly used as the count register (for loops etc.). All the counting instructions use this register. The register counts downward rather than upwards. This also holds the data to be written on the port.
EDX	I/O pointer. This is the data register, that holds the size of the data.
ESI	source indicator
EDI	destination indicator
ESP	stack pointer
EBP	stack frame base pointer (where the stack starts for a specific function)
EIP	pointer to the next instruction to execute

Module 6 (cont)

EFLAGS register	a single register that may indicate different values through its different bits
Zero Flag (ZF)	sets if the result of the instruction is zero; cleared otherwise
Sign Flag (SF)	sets equal to the most significant bit of the result
Overflow Flag (OF)	indicates the overflow of a high-order bit (leftmost bit) of data after a signed arithmetic operation.
Direction Flag (DF)	determines left or right direction for moving comparing string data. When the DF value is 0, the string operation takes left-to-right direction
Interrupt Flag (IF)	determines whether the external interrupts like keyboard entry, etc, are to be ignored or processed. It disables the external interrupt when the value is 0 and enables interrupt when set to 1

Module 6 (cont)

Trap Flag (TF)	allows setting the operation of the processor in single-step mode. The DEBUG program we used sets the trap flag, so we could step through execution one instruction at a time.
Memory Segments	Assembly follows the segmented memory model, which divides the system memory into groups of independent segments referenced by pointers located in he segment registers
Data Segment	represented by .data section and the .bss section and is used to declare the memory region, where data elements are stored for the program.
	Code Segment: is represented by .text section. This defines an area in memory that stores the instruction codes. This is also a fixed area. CS register stores the starting address of the code segment is is pointed by CS(Code segment register)
	Stack: segment contains data values passed to functions and procedures within the program. SSR (Stack segment register stores the starting address of the stack). An extra segment is used to store Extra data. It is pointed by ES (Extra segment register)
Op Codes	also called Assembly Language commands, or mnemonic codes, are different categories of commands that makes the assembly language syntax
Three main categories:	data transfer instructions
	arithmetic instructions
	logical and program control instructions
Assembly Program Structure	.data section: declare variables
	.bss section: also declares variables
	.text section: has program codes
EAX	32 bit accumulator register
RAX	64 bit accumulator register
AX	16 bit accumulator register
Assembly Language Statements	[label]mnemonic[operands][;-comment]
Instructions	
NOP	does noting, no values, may be used for a delay
PUSH	push word, double word or quad word on the stack, it automatically decrements the stack pointer esp, by 4
POP	pops the data from the stack, sets the esp automatically, it would increment esp
EQU	sets a variable equal to some memory
HLT	to halt the program
Operation Suffixes	
b	byte (8 bit)
s	short (16 bit int) or single (32 bit floating point)
w	word (16 bit)
l	long (32 bit integer or 64 bit floating point)
q	quad (64 bit)
t	ten bytes (80-bit floating point)
Addressing Modes	Direct Memory Addressing(register) : register eax has the value 0x100
	Indirect Memory Addressing: register contains the value
	Offset Addressing (register, offset): register may calculate the memory reference for final data

Module 6 (cont)

	Offset Addressing (register, offset): register may calculate the memory reference for final data
Memory Addressing	
%eax	refers to the value in register
(%eax)	means use the value in register as address to point to data at that address
9(%eax, %edx)	means use the address in %eax+%edx*9 and use the value as addresss to refer to the data at that location
Three basic kinds of Instructions	Perform arithmetic function on register or memory data
	Transfer data between memory and register.-load data from memory into register.-store data into memory
	Transfer control - Unconditional jumps to/from procedures - conditional branches
MOV instruction	The syntax is: mov? Source, destination. movb = move byte; movw = move word...
movl $0x4050, %eax	Immediate--Register,4 bytes. : in plain English, this instruction means "move the 32-bit constant value 0x4050 (or 16464 in decimal) into the %eax register."

Module 6 (cont)

movw %bp, %sp	Register--Register, 2 bytes. :copy the 16-bit value from the base pointer register (%bp) into the stack pointer register (%sp).
movb (%edi, %ecx), %ah	Memory -- Register, 1byte. :So, in plain English, this instruction reads a byte from memory at the address formed by adding the values of the %edi and %ecx registers and stores it in the high byte of the %ax register, which is %ah.
Load Effective Address - leal	variant of the movl instruction. It has the form of an instruction that reads from memory to an register, but it does not reference memory at all. Its first operand appears to be a memory reference, but instead of reading from the designated location, the instruction copies the effective address to the destination

Module 6

Control	In addition to integer registers, the CPU maintains a set of single-bit condition code registers describing attributes of the most recent arithmetic or logical operation. These registers can then be tested to perform conditional branches

Module 6 (cont)

CF	Carry Flag: The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations
ZF	Zero Flag: The most recent operation yielded zero
SF	Sign Flag: The most recent operation yielded a negative value
OF	Overflow Flag:The most recent operation caused a two's complement overflow either negative of positive
Jump Instructions and their Encoding	Under normal execution, instructions follow each other in the order they are listed. A jump instruction can cause the execution to switch to a completely new position in the program. These jump destinations are generally indicated in assembly code by a label
jmp *%eax	Uses the value in register %eax as the jump target, and the instruction
jmp * (%eax)	reads the jump target from memory, using the value in %eax as the read address

Module 6 (cont)	
Procedures in Assembly	A procedure call involves passing both data (in the form of procedure parameters and return values) and control from one part of a program to another. ☐ The data passing happens using stack in the memory, that is shared by both main program and the procedure ☐ Within the function, the need is to also allocate space for the local variables defined in the procedure on entry and deallocate them on exit. ☐ Most machines, including IA32, provide only simple instructions for transferring control to and from procedures. ☐ The part of the program that is needed to be done many times is defined in the procedure ☐ Each procedure is identified by a name ☐ The procedure is defined as a label but after the execution of the procedure, the execution returns to the same place from where it has been called when ret (return) statement is executed ☐ The procedure may flow along multiple labels as well

Module 6 (cont)	
Stack	Stack plays an important role when we use the procedures When a program starts executing, a certain contiguous section of memory is set aside for the program called the stack.
	The stack implementation has some special features, which are: ☐ The stack can only hold words or doublewords, not a byte. ☐ The stack grows in the reverse direction, i.e., toward the lower memory address ☐ The top of the stack points to the last item inserted in the stack; it points to the lower byte of the last word inserted.
	The stack pointer is the register that contains the top of the stack and base pointer is the register having the address of the bottom of the stack.