2022

Victoria University of Bangladesh

58/11/A, Panthapath, Dhaka, Bangladesh

Computer Organization & Assembly Programming

CSE-233

Final Assessment

Submitted By:

ARS Nuray Alam Parash

ID # 2120180041

18th Batch, BSc in CSE

E-mail: chatokparash@gmail.com

Mobile: 01922339393

Submission Date: 6 October 2022

Submitted To:

Umme Khadiza Tithi

Lecturer

Department of Computer Science & Engineering

Victoria University of Bangladesh (VUB)

Answer to the Question No- 1 (a)

Number of Operands: An x86 instruction can have zero to three operands. Operands are separated by commas (,) (ASCII 0x2C). For instructions with two operands, the first (left-hand) operand is the source operand, and the second (righthand) operand is the destination operand (that is, source->destination).

Operands can be immediate (that is, constant expressions that evaluate to an inline value), register (a value in the processor number registers), or memory (a value stored in memory). An indirect operand contains the address of the actual operand value. Indirect operands are specified by prefixing the operand with an asterisk (*) (ASCII 0x2A). Only jump and call instructions can use indirect operands.

- Immediate operands are prefixed with a dollar sign ($) (ASCII 0x24)
- Register names are prefixed with a percent sign (%) (ASCII 0x25)
- Memory operands are specified either by the name of a variable or by a register that contains the address of a variable. A variable name implies the address of a variable and instructs the computer to reference the contents of memory at that address. Memory references have the following syntax:segment:offset(base, index, scale).
    - Segment is any of the x86 architecture segment registers. Segment is optional: if specified, it must be separated from offset by a colon (:). If segment is omitted, the value of %ds (the default segment register) is assumed.

- Offset is the displacement from segment of the desired memory value. Offset is optional.
- Base and index can be any of the general 32–bit number registers.
- Scale is a factor by which index is to be multiplied before being added to base to specify the address of the operand. Scale can have the value of 1, 2, 4, or 8. If scale is not specified, the default value is 1.

Some examples of memory addresses are:

movl var, %eax

Move the contents of memory location var into number register %eax.

movl %cs:var, %eax

Move the contents of memory location var in the code segment (register %cs) into number register %eax.

movl $var, %eax

Move the address of var into number register %eax.

movl array_base(%esi), %eax

Add the address of memory location array_base to the contents of number register %esi to determine an address in memory. Move the contents of this address into number register %eax.

movl (%ebx, %esi, 4), %eax

Multiply the contents of number register %esi by 4 and add the result to the contents of number register %ebx to produce a memory reference. Move the contents of this memory location into number register %eax.

movl struct_base(%ebx, %esi, 4), %eax

Multiply the contents of number register %esi by 4, add the result to the contents of number register %ebx, and add the result to the address of struct_base to produce an address. Move the contents of this address into number register %eax.

Answer to the Question No- 1 (b)

Programming Language: Programming languages provide various ways of specifying programs for computers to run. Unlike natural languages, programming languages are designed to permit no ambiguity and to be concise. They are purely written languages and are often difficult to read aloud. They are generally either translated into machine code by a compiler or an assembler before being run, or translated directly at run time by an interpreter.

On the basis of this level of abstraction, there are two types of programming languages:

- Low-level language
- High-level language

Here is the difference between low-level & higher-level language:

Low-level language	High-Level Language
It is considered as a machine-friendly language.	It can be considered as a programmer-friendly language.
It requires an assembler that would translate instructions.	It requires a compiler/interpreter to be translated into machine code.
It is not portable.	It can be ported from one location to another.
It is difficult to understand.	It is easy to understand.
It is difficult to debug.	It is easy to debug.
It consumes less memory.	It is less memory efficient, i.e., it consumes more memory in comparison to low-level languages.

Answer to the Question No- 1 (c)

Here,

Graphic 7 inches X 5 inches with 600 dpi.

$= (7 \text{ x } 600) \text{ x } (5 \text{ x } 600)$

$= 4200 \text{ x } 3000\ pixels$

$= 12600000\ pixels\ (\div 8)$

$= 1575000\ bytes\ (\div 1024)$

$= 1538.08\ KB\ (\div 1024)$

$= \mathbf{1.5\ MB}$

So, the 1.5 MB memory is required to store the graphic.

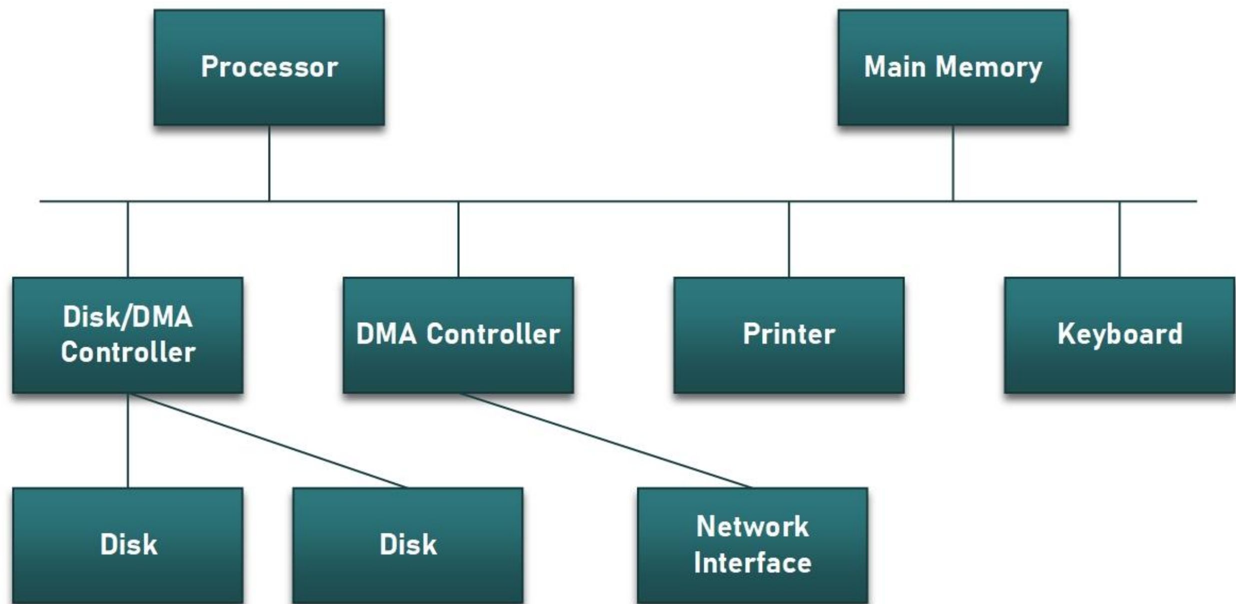
Answer to the Question No- 2 (a)

DMA Controller: DMA Controller is a hardware device that allows I/O devices to access memory with less participation from the processor directly. DMA controller needs the same old circuits of an interface to communicate with the CPU and input/output devices.

DMA controller is a control circuit that performs DMA transfers and is a part of the I/O device interface. It performs functions that generally be carried out by the processor. DMA controller must increment the memory address and keep track of the number of transfers. The operations of DMA controller must be under the control of a program executed by the processor. To initiate the transfer of a block of words, the processor sends the starting address, the number of words in the block, and the transfer's direction. On receiving this information, the DMA controller transfers the entire block and informs the processor by raising an interrupt signal. While a DMA transfer is taking place, the processor can be used to execute another program. After the DMA transfer is completed, and the processor can return to the program that requested the transfer.

Three registers in a DMA interface are:

- Starting address
- Word count
- Status and control flag

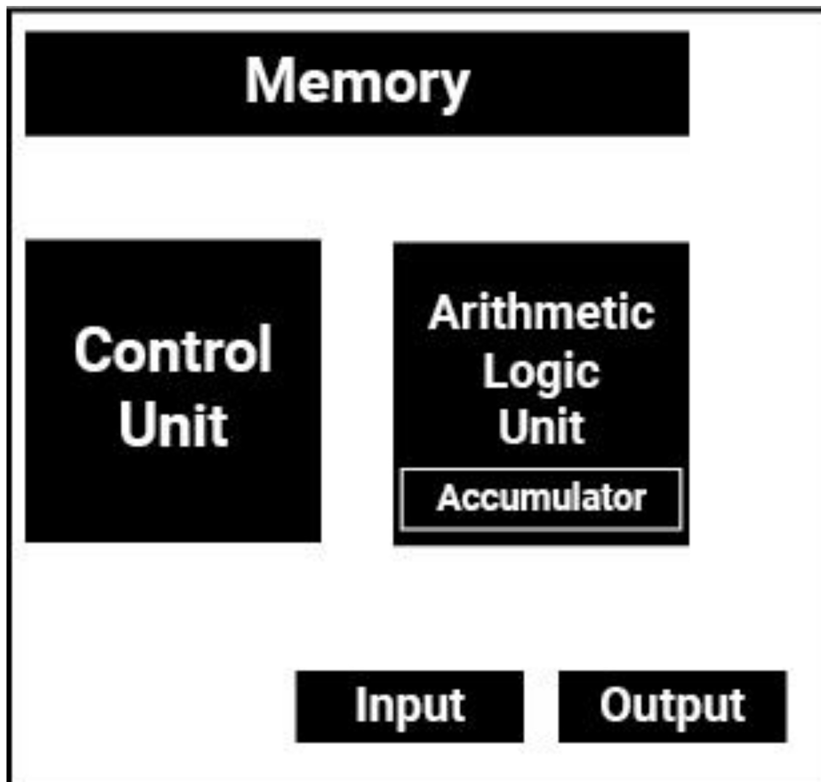


A conflict may arise if both the processor and a DMA controller or two DMA controllers try to use the bus at the same time to access the main memory. To resolve this, an arbitration procedure is implemented on the bus to coordinate the activities of all devices requesting memory transfers.

Answer to the Question No- 2 (b)

Design of Simple Units of ALU: In ECL, TTL and CMOS, there are available integrated packages which are referred to as arithmetic logic units (ALU). The logic circuitry in these units is entirely combinational (i.e. consists of gates with no feedback and no flip-flops). The ALU is an extremely versatile and useful device since, it makes available, in a single package, a facility for performing many different logical and arithmetic operations. The arithmetic Logic Unit (ALU) is a critical component of a microprocessor and is the core component of a central processing unit.

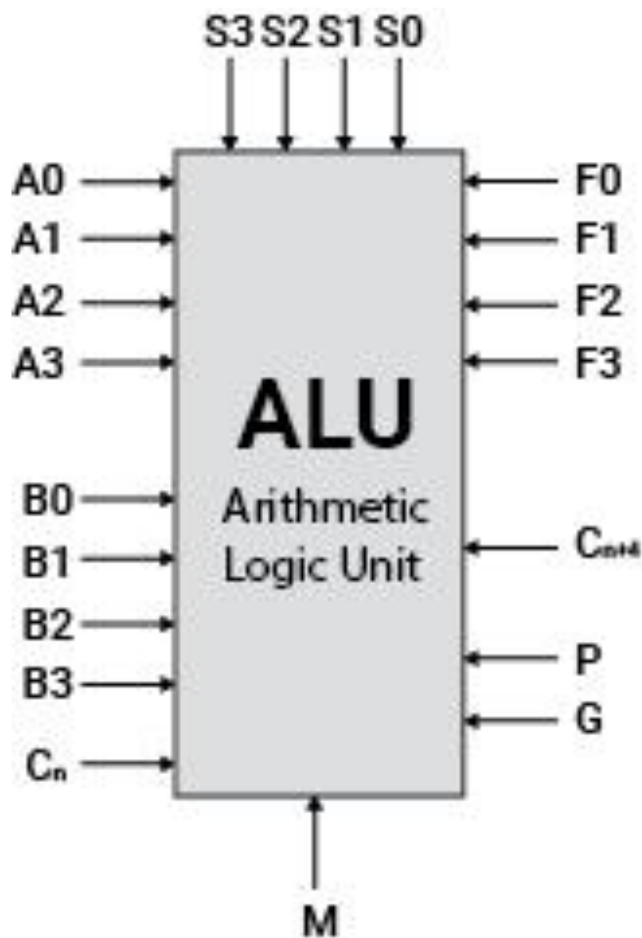
Central Processing Unit (CPU)-



ALU’s comprise the combinational logic that implements logic operations such as AND, OR, and arithmetic operations, such as ADD, SUBTRACT.

Functionally, the operation of a typical ALU is represented as shown in the diagram below,

Functional representation of Arithmetic Logic Unit-



Characteristics Of ALU: The ALU is responsible for performing all logical and arithmetic operations.

- Some of the arithmetic operations are as follows: addition, subtraction, multiplication, and division.
- Some of the logical operations are as follows: comparison between numbers, letters, and or special characters.
- The ALU is also responsible for the following conditions: Equal-to conditions, Less-than conditions, and greater than conditions.

Answer to the Question No- 2 (c)

Here,

The hexadecimal number given is $(5F.AB2)_{16}$

Now,

4-bit binary equivalent is

5 = 0101

F = 1111

A = 1010

B = 1011

2 = 0010

Hence, the $(5F.AB2)_{16}$hexadecimal number's equivalent binary number is $(01011111.101010110010)_2$

Answer to the Question No- 3 (a)

Current Usage of Assembly Language: There have always been debates over the usefulness and performance of assembly language relative to high-level languages. Assembly language has specific niche uses where it is important; see below. But in general, modern optimizing compilers are claimed to render high-level languages into code that can run as fast as hand-written assembly, despite the counterexamples that can be found. The complexity of modern processors and memory sub-systems makes effective optimization increasingly difficult for compilers, as well as assembly programmers. Moreover, and to the dismay of efficiency lovers, increasing processor performance has meant that most CPUs sit idle most of the time, with delays caused by predictable bottlenecks such as I/O operations and paging. This has made raw code execution speed a non-issue for many programmers.

There are some situations in which developers might choose to use assembly language:

- A stand-alone executable of compact size is required that must execute without recourse to the run-time components or libraries associated with a high-level language; this is perhaps the most common situation. For example, firmware for telephones, automobile fuel, and ignition systems, air-conditioning control systems, security systems, and sensors.
- Code that must interact directly with the hardware, for example in device drivers and interrupt handlers.

- Programs that need to use processor-specific instructions are not implemented in a compiler. A common example is the bitwise rotation instruction at the core of many encryptions' algorithms.
- Programs that create vectorized functions for programs in higher-level languages such as C. In the higher-level language, this is sometimes aided by compiler intrinsic functions which map directly to SIMD mnemonics but nevertheless result in a one-to-one assembly conversion specific for the given vector processor.
- Programs requiring extreme optimization, for example, an inner loop in a processor-intensive algorithm. Game programmers take advantage of the abilities of hardware features in systems, enabling games to run faster. Also, large scientific simulations require highly optimized algorithms, e.g. linear algebra with BLAS or discrete cosine transformation (e.g. SIMD assembly version from x264)
- Situations where no high-level language exists, on a new or specialized processor, for example.
- Programs that need precise timing such as-
    - real-time programs such as simulations, flight navigation systems, and medical equipment. For example, in a fly-by-wire system, telemetry must be interpreted and acted upon within strict time constraints. Such systems must eliminate sources of unpredictable delays, which may be created by (some) interpreted languages, automatic garbage collection, paging operations, or preemptive multitasking. However, some higher-level languages incorporate run-time components and operating system interfaces that can introduce such delays. Choosing assembly or lower-

level languages for such systems gives programmers greater visibility and control over processing details.

- cryptographic algorithms that must always take strictly the same time to execute, preventing timing attacks.

Answer to the Question No- 3 (b)

Input/output in assembly Language Program: Input/Output (I/O) instructions are used to input data from peripherals, output data to peripherals, or read/write input/output controls. Early computers used special hardware to handle I/O devices. The trend in modern computers is to map I/O devices in memory, allowing the direct use of any instruction that operates on memory for handling I/O.

- **IN** Input; MIX; initiate transfer of information from the input device specified into consecutive locations starting with M, block size implied by unit.
- **OUT** Output; MIX; initiate transfer of information from consecutive locations starting with M to the output device specified, block size implied by unit
- **IOC** Input-Output Control; MIX; initiate I/O control operation to be performed by a designated device
- **JRED** Jump Ready; MIX; Jump if the specified unit is ready (completed previous IN, OUT, or IOC operation); if jump occurs, J-register loaded with the address of the instruction which would have been next if the jump had not been taken

- **JBUS** Jump Busy; MIX; Jump if the specified unit is not ready (not yet completed previous IN, OUT, or IOC operation); if jump occurs, J-register loaded with the address of the instruction which would have been next if the jump had not been taken

Answer to the Question No- 3 (c)

Registers are normally measured by the number of bits they can hold, for example, an "8-bit register" or a "32-bit register". A processor often contains several kinds of registers, that can be classified accordingly to their content or instructions that operate on them:

- **User-Accessible Registers-** The most common division of user-accessible registers is into data registers and address registers.
- **Data Registers-** It can hold numeric values such as integer and floating-point values, as well as characters, small-bit arrays, and other data. In some older and low-end CPUs, a special data register, known as the accumulator, is used implicitly for many operations.
- **Address Registers-** It holds addresses and are used by instructions that indirectly access primary memory.
    - Some processors contain registers that may only be used to hold an address or only to hold numeric values (in some cases used as an index register whose value is added as an offset from some address); others

allow registers to hold either kind of quantity. A wide variety of possible addressing modes, used to specify the effective address of an operand, exist.

  - The stack pointer is used to manage the run-time stack. Rarely, other data stacks are addressed by dedicated address registers, see stack machine.

- **Conditional Registers** hold truth values often used to determine whether some instruction should or should not be executed.
- **General Purpose Registers (GPRs)** can store both data and addresses, i.e., they are combined Data/Address registers, and rarely the register file is unified to include floating point as well.
- **Floating Point Registers (FPRs)** store floating point numbers in many architectures.
- **Constant Registers** hold read-only values such as zero, one, or pi.
- **Vector Registers** hold data for vector processing done by SIMD instructions (Single Instruction, Multiple Data).
- **Special Purpose Registers (SPRs)** hold program states; they usually include the program counter (aka instruction pointer) and status register (aka processor status word). The aforementioned stack pointer is sometimes also included in this group. Embedded microprocessors can also have registers corresponding to specialized hardware elements.
  - **Instruction Registers** store the instruction currently being executed.
- In some architectures, **Model-Specific Registers** (also called machine-specific registers) store data and settings related to the processor itself.

Because their meanings are attached to the design of a specific processor, cannot be expected to remain standard between processor generations.

- **Control and Status Registers**- There are three types: program counter, instruction registers, and program status word (PSW).
- Registers related to fetching information from RAM, a collection of storage registers located on separate chips from the CPU (unlike most of the above, these are generally not architectural registers):
    - Memory buffer register (MBR)
    - Memory data register (MDR)
    - Memory address register (MAR)
    - Memory Type Range Registers (MTRR)

>> END <<