Victoria University
of Bangladesh

Assessment Topic:

Final Assessment

Course Title: Computer Organization & Assembly Language

Course Code: CSE-233

Submitted To:

Umme Khadiza Tithi

Lecturer, Department of Computer Science & Engineering

Victoria University of Bangladesh

Submitted By:

Ruhul Amin

ID: 2120180051

Department: CSE

Semester: Summar-2022

Batch: 18th

Submission Date: 6th October, 2022

Question 01 a): Answer:

Operands in assembly language: Instruction sets may be categorized by the maximum number of operands explicitly specified in instructions.
(In the examples that follow, a, b, and c are (direct or calculated) addresses referring to memory cells, while reg1 and so on refer to machine registers.)

- 0-operand (zero-address machines), so called stack machines: All arithmetic operations take place using the top one or two positions on the stack: **push** a, **push** b, **add**, **pop** c. For stack machines, the terms "0-operand" and "zero-address" apply to arithmetic instructions, but not to all instructions, as 1-operand push and pop instructions are used to access memory.
- 1-operand (one-address machines), so called accumulator machines, include early computers and many small microcontrollers: most instructions specify a single right operand (that is, constant, a register, or a memory location), with the implicit accumulator as the left operand (and the destination if there is one): load a, add b, store c. A related class is practical stack machines which often allow a single explicit operand in arithmetic instructions: **push** a, **add** b, **pop** c.
- 2-operand — many CISC and RISC machines fall under this category:
    - CISC — often **load** a, reg1; **add** reg1, b; **store** reg1, c on machines that are limited to one memory operand per instruction; this may be load and store at the same location
    - CISC — **move** a->c; **add** c+=b.
    - RISC — Requiring explicit memory loads, the instructions would be: **load** a, reg1; **load** b, reg2; **add** reg1, reg2; **store** reg2, c
- 3-operand, allowing better reuse of data:
    - CISC — It becomes either a single instruction: **add** a, b, c, or more typically: **move** a,reg1; **add** reg1,b,c as most machines are limited to two memory operands.
    - RISC — arithmetic instructions use registers only, so explicit 2-operand load/store instructions are needed: **load** a,reg1; **load** b,reg2; **add** reg1+reg2->reg3; **store** reg3,c; unlike 2-operand or 1-operand, this leaves all three values a, b, and c in registers available for further reuse
- more operands—some CISC machines permit a variety of addressing modes that allow more than 3 operands (registers or memory accesses), such as the VAX "POLY" polynomial evaluation instruction.

Due to the large number of bits needed to encode the three registers of a 3-operand instruction, RISC processors using 16-bit instructions are invariably 2-operand machines, such as the Atmel AVR, the TI MSP430, and some versions of the ARM Thumb. RISC processors using 32-bit instructions are usually 3-operand machines, such as processors implementing the Power Architecture, the SPARC architecture, the MIPS architecture, the ARM architecture, and the AVR32 architecture.
Each instruction specifies some number of operands (registers, memory locations, or immediate

values) explicitly. Some instructions give one or both operands implicitly, such as by being stored on top of the stack or in an implicit register. If some of the operands are given implicitly, fewer operands need be specified in the instruction. When a "destination operand" explicitly specifies the destination, an additional operand must be supplied. Consequently, the number of operands encoded in an instruction may differ from the mathematically necessary number of arguments for a logical or arithmetic operation (the arity). Operands are either encoded in the "opcode" representation of the instruction, or else are given as values or addresses following the instruction.

Question 01 b): Answer:

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. Sometimes programs are executed by a hybrid method of the two techniques. Programming language can be

Low-level languages or

Higher-level languages

Differences between low-level languages and higher-level languages:

High-Level Language	Low-level language
It can be considered as a programmer-friendly language.	It is considered as a machine-friendly language.
It requires a compiler/interpreter to be translated into machine code.	It requires an assembler that would translate instructions.
It can be ported from one location to another.	It is not portable.
It is easy to understand.	It is difficult to understand.
It is easy to debug.	It is difficult to debug.
It is less memory efficient, i.e., it consumes more memory in comparison to low-level languages.	It consumes less memory.
It can run on any platform.	It is machine-dependent.
It is simple to maintain.	It is complex to maintain comparatively.
It is used widely for programming.	It is not commonly used now-a-days in programming.
It is programmer friendly language.	It is a machine friendly language.
High-level languages do not provide various facilities at the hardware level.	Low-level languages are very close to the hardware. They help in writing various programs at the hardware level.

The process of modifying programs is very difficult with high-level programs. It is because every single statement in it may execute a bunch of instructions.	The process of modifying programs is very easy in low-level programs. Here, it can directly map the statements to the processor instructions.

Question 01 c): Answer:

Graphic 7 inches x 5 inches with 600dpi. Calculation given below for the amount of memory required to store the graphic.

(7 x 600) x (5 x 600) = 4200 x 3000 pixels

= 12600000 pixels (÷ 8)

= 1575000 bytes (÷ 1024)

= 1538.08 Kb (÷ 1024)

= 1.5 Mb

So, we need 1.5 Mb of memory to store the graphic.

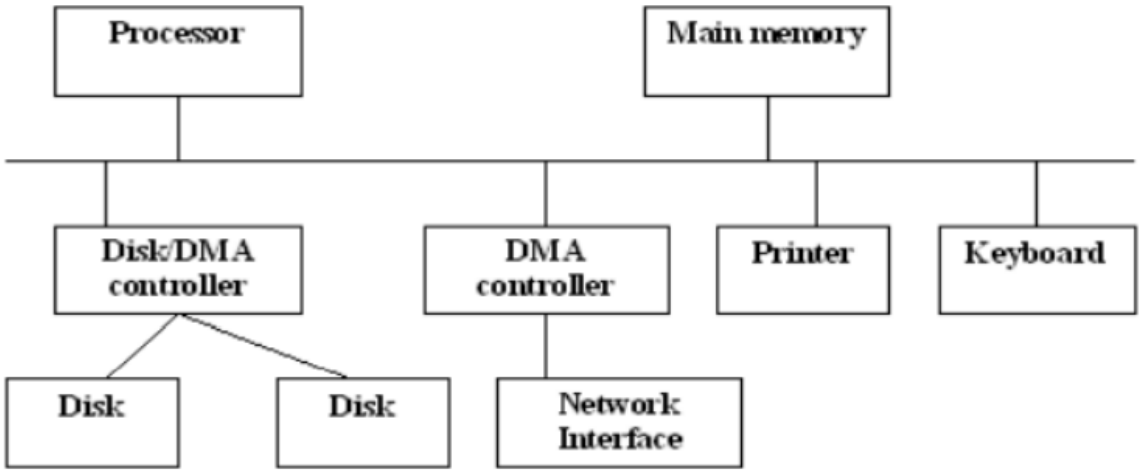
Question 02 a): Answer:

DMA Controller: Basically, for high speed I/O devices, the device interface transfer data directly to or from the memory without informing the processor. When interrupts are used, additional overhead involved with saving and restoring the program counter and other state information. To transfer large blocks of data at high speed, an alternative approach is used. A special control unit will allow transfer of a block of data directly between an external device and the main memory, without continuous intervention by the processor.

DMA controller is a control circuit that performs DMA transfers, is a part of the I/O device interface. It performs functions that normally 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 block of words, the processor sends the starting address, the number of words in the block and the direction of the transfer. On receiving this information, 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, 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



Use of DMA controllers in a computer system

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.

Question 02 b): Answer:

Arithmetic logic unit: Short for Arithmetic Logic Unit, ALU is one of the many components within a computer processor. The ALU performs mathematical, logical, and decision operations in a computer and is the final processing performed by the processor. After the information has been processed by the ALU, it is sent to the computer memory. In some computer processors, the ALU is divided into two distinct parts, the AU and the LU. The AU performs the arithmetic operations and the LU performs the logical operations.

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 this 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 single package, facility for performing many different logical and arithmetic operations.

Arithmetic Logic Unit (ALU) is a critical component of a microprocessor and is the core component of central processing unit.

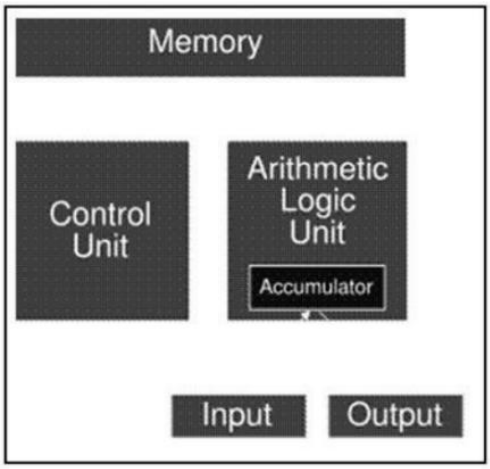


Fig.1 *Central Processing Unit (CPU)*

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 typical ALU is represented as shown in diagram below

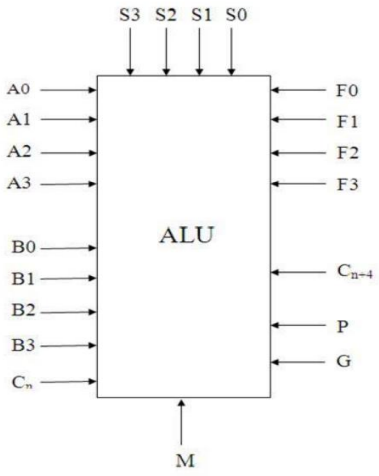


Fig.2 *Functional representation of Arithmetic Logic Unit*

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, letter and or special characters.

- The ALU is also responsible for the following conditions: Equal-to conditions, Less-than condition and greater than condition.

Question 02 c): Answer:

Note: According to the question paper there is a printing mistake because 5G.AB216 is not a valid number. I already talked to Tithi ma'am and the question will be

Convert $5E.AB2_{16}$ into an equivalent binary number. [page number 27, example 1.22]

Solution: The hexadecimal number given is 5 E.A B 2

4-bit binary equivalent

5				E				A				B				2			
0	1	0	1	1	1	1	0	1	0	1	0	1	0	1	1	0	0	1	0

Hence the equivalent binary number is $(01011110.101010110010)_2$

Table of Hex to Binary Number

Hex	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Question 03 a): Answer:

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 highlevel 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 not implemented in a compiler. A common example is the bitwise rotation instruction at the core of many encryption 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 processorintensive 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

Question 03 b): Answer:

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 designated device
- JRED Jump Ready; MIX; Jump if 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 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 take

Question 03 c): Answer:

Different types of Registers: In computer architecture, a processor register is a small amount of storage available as part of a CPU or other digital processor. Such registers are (typically) addressed by mechanisms other than main memory and can be accessed more quickly. Almost all computers, load-store architecture or not, load data from a larger memory into registers where it is used for arithmetic, manipulated, or tested, by some machine instruction. Manipulated data is then often stored back in main memory, either by the same instruction or a subsequent one. Modern processors use either static or dynamic RAM as main memory, the latter often being implicitly accessed via one or more cache levels. A common property of computer programs is locality of reference: the same values are often accessed repeatedly and frequently used values held in registers improves performance. This is what makes fast registers (and caches) meaningful.

Processor registers are normally at the top of the memory hierarchy, and provide the fastest way to access data. The term normally refers only to the group of registers that are directly encoded as part of an instruction, as defined by the instruction set. However, modern high-performance CPUs often have duplicates of these "architectural registers" in order to improve performance via register renaming, allowing parallel and speculative execution. Modern x86 is perhaps the most well-known example of this technique.

Allocating frequently used variables to registers can be critical to a program's performance. This register allocation is either performed by a compiler, in the code generation phase, or manually, by an assembly language programmer.

Categories of registers

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** 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** hold 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 state; 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, they 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)
  - Hardware registers are similar, but occur outside CPUs.

Some examples

The table shows the number of registers of several mainstream architectures. Note that in x86-compatible processors the stack pointer (ESP) is counted as an integer register, even though there are a limited number of instructions that may be used to operate on its contents. Similar caveats apply to most architectures.

x86 FPUs have 8 80-bit stack levels in legacy mode, and at least 8 128-bit XMM registers in SSE modes.

Although all of the above listed architectures are different, almost all are a basic arrangement known as the Von Neumann architecture, first proposed by mathematician John von Neumann.

Architecture	**Integer Registers**	**FP Registers**	**Notes**
x86-16 x86-32	8 8	8 8	8086/8088, 80186/80188, 80286, with 8087, 80187 or 80287 for floating-point 80386 required 80387 for floating-point
x86-64	16	16	
IBM/360 z/Architecture	16 16	4 16	
Itanium	128	128	And 64 1-bit predicate registers and 8 branch registers. The FP registers are 82 bit.
SPARC IBM Cell IBM POWER	31 4~16 32	32 1~1 32	Global register 0 is hardwired to 0. Uses register windows. Each SPE contains a 128-bit, 128-entry unified register file. And 1 link and 1 count register.
Power Architecture	32	32	And 32 128-bit vector registers, 1 link and 1 count register.
Alpha 6502	32 3	32 0	
W65C816S	5 2	0	

PIC microcontroller AVR microcontroller ARM 32-bit	1 32 16	0 0 Varies (up to 32)	Of which, register r15 is the program counter and r8-r14 can be switched out for others (banked) on a processor mode switch.
ARM 64-bit	31	32	In addition, register r31 is the stack pointer or hardwired to 0
MIPS Epiphany	31 64 (per core)	32	Register 0 is hardwired to 0. Each instruction controls whether registers are interpreted as integers or single precision floating point. 16 or 64 cores.