Victoria University of Bangladesh
Name: Md. Ziaul Hoque “Sohel”
Student ID: 2221220031
Course Title: Computer Organization Assembly Programming
Course Code: CSE-233
Batch: 22nd (evening)
Semester: Spring-2024

Ans to the Que No 1(A)

Number of operands in assembly language:

The number of operands in assembly language instructions can vary depending on the specific instruction and architecture. However, many instructions typically have one, two, or three operands.

1. **Zero Operand Instructions**: Some instructions do not require any operands, as they operate directly on the CPU's registers or manipulate memory based on their opcode.

2. **One Operand Instructions**: These instructions typically have a source or destination operand, such as MOV (move) instructions which move data from one location to another.

3. **Two Operand Instructions**: These instructions typically involve operations between two operands, such as ADD (addition) or SUB (subtraction) instructions.

4. **Three Operand Instructions**: Less common but still present in some architectures, these instructions involve three operands, often seen in more complex arithmetic or logic operations.

The number of operands and their specific usage will depend on the assembly language syntax and the architecture for which the code is being written.

Ans to the Que No 1(B)

Difference between low-level language and high-level language:

Parameter	**High-Level Language**	**Low-Level Language**
Basic	These are programmer-friendly languages that are manageable, easy to understand, debug, and widely used in today's times.	These are machine-friendly languages that are very difficult to understand by human beings but easy to interpret by machines.
Ease of Execution	These are very easy to execute.	These are very difficult to execute.
Process of Translation	High-level languages require the use of a compiler or an interpreter for their translation into the machine code.	Low-level language requires an assembler for directly translating the instructions of the machine language.

Efficiency of Memory	These languages have a very low memory efficiency. It means that they consume more memory than any low-level language.	These languages have a very high memory efficiency. It means that they consume less energy as compared to any high-level language.
Portability	These are portable from any one device to another.	A user cannot port these from one device to another.
Comprehensibility	High-level languages are human-friendly. They are, thus, very easy to understand and learn by any programmer.	Low-level languages are machine-friendly. They are, thus, very difficult to understand and learn by any human.
Dependency on Machines	High-level languages do not depend on machines.	Low-level languages are machine-dependent and thus very difficult to understand by a normal user.
Debugging	It is very easy to debug these languages.	A programmer cannot easily debug these languages.
Maintenance	High-level languages have a simple and comprehensive maintenance technique.	It is quite complex to maintain any low-level language.
Usage	High-level languages are very common and widely used for programming in today's times.	Low-level languages are not very common nowadays for programming.
Speed of Execution	High-level languages take more time for execution as compared to low-level languages because these require a translation program.	The translation speed of low-level languages is very high.
Abstraction	High-level languages allow a higher abstraction.	Low-level languages allow very little abstraction or no abstraction at all.
Need of Hardware	One does not require a knowledge of hardware for writing programs.	Having knowledge of hardware is a prerequisite to writing programs.
Facilities Provided	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.
Ease of Modification	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.
Examples	Some examples of high-level languages include Perl, BASIC, COBOL, Pascal, Ruby, etc.	Some examples of low-level languages include the Machine language and Assembly language.

Ans to the Que No 2 (A)

Graphics 7 inches * 5 inches with 600dpi. The amount of memory required to store the graphic:

To calculate the amount of memory required to store a graphic, you first need to find out how many pixels the graphic contains.

Given that the graphic is 7 inches by 5 inches with a resolution of 600 dots per inch (dpi), you can calculate the number of pixels in each dimension:

Width in pixels = 7 inches * 600 dpi = 4200 pixels
Height in pixels = 5 inches * 600 dpi = 3000 pixels

Now,
to find out the total number of pixels in the graphic, you multiply the width by the height:

Total pixels = Width in pixels * Height in pixels
= 4200 pixels * 3000 pixels = 12,600,000 pixels

Since,

each pixel stores color information, you also need to consider the color depth. Let's assume the graphic is using 24 bits per pixel (which is common for images),

where each pixel is represented by 3 bytes (1 byte for red, 1 byte for green, and 1 byte for blue).

Memory required = Total pixels * Color depth
= 12,600,000 pixels * 24 bits/pixel = 302,400,000 bits

To convert bits to bytes, divide by 8:

Memory required in bytes = 302,400,000 bits / 8 = 37,800,000 bytes

To convert bytes to megabytes (MB), divide by 1024*1024:

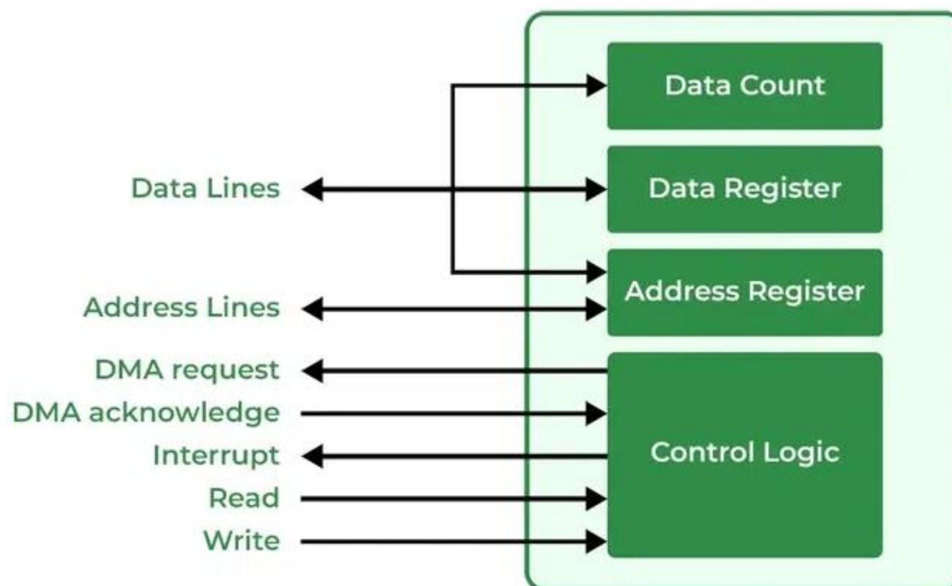
Memory required in MB ≈ 37,800,000 bytes / (1024 * 1024) ≈ 36.06 MB

So, approximately 36.06 megabytes of memory would be required to store the graphic.

Ans to the Que No 2 (A)

DMA Controllers:

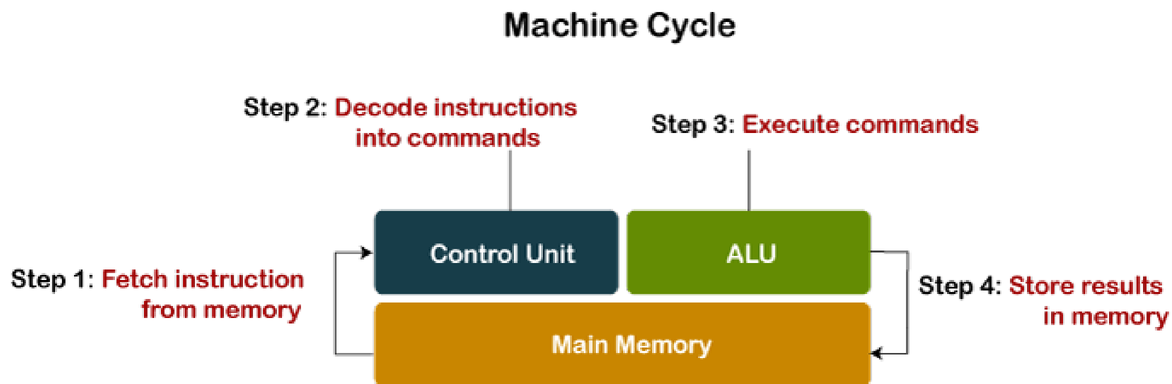
DMA Controller is a type of control unit that works as an interface for the data bus and the I/O Devices. As mentioned, DMA Controller has the work of transferring the data without the intervention of the processors, processors can control the data transfer. DMA Controller also contains an address unit, which generates the address and selects an I/O device for the transfer of data. Here we are showing the block diagram of the DMA Controller.



Ans to the Que No 2 (B)

Simple Unite of ALU (Arithmetic Logic Unit)

In the computer system, ALU is a main component of the central processing unit, which stands for arithmetic logic unit and performs arithmetic and logic operations. It is also known as an integer unit (IU) that is an integrated circuit within a CPU or GPU, which is the last component to perform calculations in the processor. It has the ability to perform all processes related to arithmetic and logic operations such as addition, subtraction, and shifting operations, including Boolean comparisons (XOR, OR, AND, and NOT operations). Also, binary numbers can accomplish mathematical and bitwise operations. The arithmetic logic unit is split into AU (arithmetic unit) and LU (logic unit). The operands and code used by the ALU tell it which operations have to perform according to input data. When the ALU completes the processing of input, the information is sent to the computer's memory.



Characteristics of ALU (Arithmetic Logic Unit)

The Arithmetic Logic Unit (ALU) is a crucial component of a computer's central processing unit (CPU), responsible for performing arithmetic and logic operations on data. Here are some key characteristics of the ALU:

1. **Arithmetic Operations**: The ALU performs basic arithmetic operations such as addition, subtraction, multiplication, and division. These operations are essential for numerical computations and data manipulation.
2. **Logic Operations**: In addition to arithmetic operations, the ALU also performs logical operations such as AND, OR, NOT, and XOR. These operations are fundamental for implementing conditional statements, bitwise manipulation, and boolean algebra.
3. **Bitwise Operations**: ALUs are capable of performing bitwise operations, which involve operations on individual bits of binary data. Bitwise operations include shifting bits left or right, setting or clearing specific bits, and performing bitwise logical operations.
4. **Speed**: The ALU is designed to perform operations at high speed, as it is one of the critical components responsible for the overall performance of the CPU. The speed of the ALU directly impacts the overall speed of computations and program execution.
5. **Parallelism**: Many modern ALUs are designed to execute multiple operations in parallel, either by incorporating multiple arithmetic and logic units or by implementing pipelining techniques to overlap the execution of multiple instructions.
6. **Data Width**: The ALU typically operates on data of a fixed width, determined by the architecture of the CPU. For example, a 32-bit ALU can perform operations on data with a width of 32 bits at a time.
7. **Registers**: The ALU interacts closely with the CPU's registers, which are small, high-speed storage locations used for holding data temporarily during processing. The ALU reads data from registers, performs operations on them, and writes the results back to registers.
8. **Control Signals**: The ALU receives control signals from the CPU's control unit, which determine the type of operation to be performed (e.g., addition, subtraction, logical AND) and the operands involved.
9. **Flags**: After performing arithmetic or logic operations, the ALU sets flags in the CPU's status register to indicate conditions such as overflow, carry, zero result, and negative result. These flags are used by the CPU to make decisions during program execution.
10. **Customization**: In some architectures, the ALU can be customized or extended with additional instructions or capabilities to suit specific applications or performance requirements.

Ans to the Que No 2 (C)

Convert 5G.AB216 to binary number:

To convert a hexadecimal number like 5G.AB216 into binary, you first need to convert each hexadecimal digit into its binary representation.

Here's the conversion:

So, 5G.AB216 in binary would be:
101.0011 01110100101111

Ans to the Que No 3 (A)

Uses of Assembly Language:

1. Assembly language uses easy-to-understand instructions to communicate with a computer's hardware, allowing programmers to control how the system works.
2. Assembly language is defined as a type of programming language designed to be used by developers to write programs that can run directly on a computer's central processing unit (CPU).
3. It is a low-level language, which means it is closer to the machine code the CPU can execute, making it more powerful than other higher-level languages such as C++, Java, or Python.
4. This article explains assembly language, its working, features, and key advantages.

Ans to the Que No 3 (B)

Input Output Assembly Language:

Input and output (I/O) operations in assembly language depend on the specific assembly language and the underlying hardware architecture. Here, I'll demonstrate basic I/O operations in assembly language using two different environments: x86 architecture with DOS interrupts (using NASM assembler) and ARM architecture.

x86 Assembly (using NASM and DOS interrupts)
In a DOS environment, you can use interrupts to handle I/O operations. Here's a simple program to read a character from the keyboard and display it on the screen using NASM syntax.
ARM Assembly
In an ARM environment, especially when working on a system like Raspberry Pi with Linux, you use system calls for I/O operations. Here's an example program to read a character from standard input and print it to standard output using ARM assembly.

Ans to the Que No 3 (C)

Types of Registers:

Registers are small, fast storage locations within a CPU that hold data and instructions temporarily during processing. They are essential for the efficient functioning of the processor. There are several types of registers, each serving specific purposes:

1. Accumulator (AC)
Function: Holds intermediate results of arithmetic and logical operations.
Usage: Central to the arithmetic logic unit (ALU), often involved in most operations.

2. Program Counter (PC)
Function: Holds the address of the next instruction to be executed.
Usage: Automatically increments after each instruction fetch to point to the next instruction.

3. Memory Address Register (MAR)
Function: Holds the memory address of data that needs to be accessed.
Usage: Used during the fetch phase to specify the address of the memory location.

4. Memory Data Register (MDR)
Function: Holds the data that is being transferred to or from the memory location specified by the MAR.
Usage: Acts as a buffer between the CPU and memory.

5. Instruction Register (IR)
Function: Holds the current instruction that is being executed.
Usage: Decoded by the control unit to generate appropriate control signals.

6. Stack Pointer (SP)
Function: Points to the top of the current stack in memory.
Usage: Used for managing function calls, returns, and local variables.

7. Base Register
Function: Holds the base address of a segment.
Usage: Used in segmented memory systems to facilitate address translation.

8. Index Register
Function: Used for indexed addressing modes.
Usage: Facilitates operations on arrays and tables by providing an offset value.

9. Flag Register (Status Register)
Function: Holds flags that indicate the status of the processor and the outcome of operations (e.g., zero, carry, overflow, sign flags).
Usage: Used by the CPU to make decisions based on the results of operations.

10. General Purpose Registers (GPRs)
Function: Used for a wide range of functions, including arithmetic, data movement, and address calculation.
Usage: Vary in number and function depending on the architecture (e.g., AX, BX, CX, DX in x86 architecture).

11. Floating Point Registers (FPRs)
Function: Hold floating-point numbers for arithmetic operations.
Usage: Used in systems that support floating-point calculations, particularly in scientific and mathematical applications.

12. Segment Registers
Function: Hold the starting address of a segment.
Usage: Used in segmented memory architectures to divide the memory into different segments (e.g., code, data, stack segments).

13. Control Registers
Function: Control various aspects of the CPU operation, including control and status information.
Usage: Includes registers like CR0, CR1, CR2, CR3 in x86 architecture, used for controlling paging, protection levels, and other system settings.

14. Special Purpose Registers
Function: Specific to certain operations and vary by architecture.
Usage: Include registers like the Model-Specific Registers (MSRs) in x86, used for performance monitoring, system configuration, etc.
These registers play crucial roles in the operation of a CPU, allowing for efficient processing and execution of instructions.

- END -