Final Assessment │Summer 2022

Md. Shafayet Hossain

CSE- 21st Batch

Computer Organization & Assembly Programming

Code: CSE 233│ID: 2121210071

Answer to the Question no- 01 (a)

Operands in Assembly Language:

Each assembly language statement is split into an opcode and an operand. The opcode is the instruction that is executed by the CPU and the operand is the data or memory location used to execute that instruction.

Number of Operands in Assembly Language:

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 multipled 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.

Answer to the Question no- 01 (b)

➢ Programming Language:

Programming languages define and compile a set of instructions for the CPU (Central Processing Unit) for performing any specific task. Every programming language has a set of keywords along with syntax- that it uses for creating instructions.

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

- Low-level language
- High-level language

The primary difference between low and high-level languages is that any programmer can understand, compile, and interpret a high-level language feasibly as compared to the machine. The machines, on the other hand, are capable of understanding the low-level language more feasibly compared to human beings.

➢ Differences Between High-Level & Low-Level Languages-

❖ High-Level Languages:

- One can easily interpret and combine these languages as compared to the low-level languages.
- They are very easy to understand.
- Such languages are programmer-friendly.
- Debugging is not very difficult.
- They come with easy maintenance and are thus simple and manageable.
- One can easily run them on different platforms.
- They require a compiler/interpreter for translation into a machine code.
- A user can port them from one location to another.
- Such languages have a low efficiency of memory. So it consumes more memory than the low-level languages.
- They are very widely used and popular in today's times.
- Java, C, C++, Python, etc., are a few examples of high-level languages.

❖ Low-Level Languages:

- They are also called machine-level languages.
- Machines can easily understand it.
- High-level languages are very machine-friendly.
- Debugging them is very difficult.
- They are not very easy to understand.
- All the languages come with complex maintenance.
- They are not portable.
- These languages depend on machines. Thus, one can run it on various platforms.
- They always require assemblers for translating instructions.
- Low-level languages do not have a very wide application in today's times.

Answer to the Question no- 01 (c)

Graphic 7 inches x 5 inches with 600dpi. Calculate 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 (Answer)

Answer to the Question no- 02 (a)

DMA Controllers in a computer System:

Definition: DMA or Direct Memory Access Controller is an external device that controls the transfer of data between I/O device and memory without the involvement of the processor. It holds the ability to directly access the main memory for read or write operation.

DMA controller was designed by Intel, to have the fastest data transfer rate with less processor utilization.

Typical examples are disk controllers, Ethernet controllers, USB controllers, and video controllers. Usually the DMA controller built into these devices can only move data between the device itself and main memory

Answer to the Question no- 02 (b)

Number of Operand:

The operand is the object that is being worked on by an operation. Operations can be mathematical ones such as multiplication or addition, or they can be more sophisticated functions. A basic example of an operand would be a variable declared in a program that would change value because of operations.

A basic example of an operand would be a variable declared in a program that would change value because of operations. For example, a programmer can create a variable x. He can set the value of x at anything, for example, one. Then, that value can be changed using an operator, for example, by entering something like x=x +3. The value of x then becomes 4.

Different operators continue to change this operand for programming and computing purposes.

Calling an operand an 'object' also shows how the evolution of computer programming has treated this principle.

What is an arithmetic-logic unit (ALU)?

The ALU can perform two types of operations: arithmetic and logic.

The set of arithmetic operations supported by an ALU may be limited to addition and subtraction, or it may include multiplication, division, trigonometry functions such as sine, cosine, and so on, as well as square roots. Some can only operate on whole numbers (integers), whereas others, albeit with limited precision, use floating point to represent real numbers. Any computer capable of performing only the simplest operations, on the other hand, can be programmed to break down more complex operations into simple steps that it can perform. As a result, any computer can be programmed to perform any arithmetic operation—though doing so will take longer if the ALU does not directly support the operation. An ALU can also compare numbers and return boolean truth values (true or false) based on whether one is equal to, greater than, or less than the other (is 64 greater than 65?).

AND, OR, XOR, and NOT are examples of logic operations that use Boolean logic. These can be useful for creating complex conditional statements and processing boolean logic.Superscalar computers may contain multiple ALUs, allowing them to process multiple instructions at the same time. ALUs that can perform arithmetic on vectors and matrices are frequently found in graphics processors and computers with SIMD and MIMD features.

Answer to the Question no- 03 (a)

Current usage of Assembly Programming:

Assembly language is used in specific use cases. It is still the most common in specific use cases in embedded systems software however it is still present in other low level projects like Linux kernel. It is used here and there because there are certain things then can be done only in assembly and because assembly results in a faster and smaller code. One real life example is to make it more clear that ome critical parts of firmware for smart cards is written in assembly in order to protect software execution against different types of attacks and assembly is the only language where fully control binary representation of the program. Compiler might leave security hole in the firmware which might be an entry point for the attacker.

There are few reasons to use assembler for device-level programming. Systems-level languages such as C and C++ can run on stand-alone, bare-metal systems with few resources and no OS. In the past is was common to resort to assembler to implement time critical code sections (often with little evidence of necessity in my experience), but with modern optimizing compilers that is seldom necessary, and the compiler encapsulates instruction set expertise that would take a human many hours to learn for just a single architecture - so the compiler will often beat a human on performance.

Here are few example engineering positions requiring advanced assembly language knowledge:

- Compiler Engineer.
- Firmware Engineer.
- Secure Firmware Engineer.
- Malware Analyst.

Answer to the Question no- 03 (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 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 taken

Answer to the Question no- 03 (c)

Different Types of Registers:

A register capable of shifting its binary contents either to the left or to the right is called a shift register. The shift register permits the stored data to move from a particular location to some other location within the register. Registers can be designed using discrete flip-flops (S-R, J-K, and D-type). The data in a shift register can be shifted in two possible ways:

(a) Serial shifting and

(b) Parallel shifting.

The serial shifting method shifts one bit at a time for each clock pulse in a serial manner, beginning with either LSB or MSB. On the other hand, in parallel shifting operation, all the data (input or output) gets shifted simultaneously during a single clock pulse. Hence, we may say that parallel shifting operation is much faster than serial shifting operation. There are two ways to shift data into a register (serial or parallel) and similarly two ways to shift the data out of the register. This leads to the construction of four basic types of registers as shown in below. All of the four configurations are commercially available as TTL MSI/LSI circuits. They are:

1. Serial in/Serial out (SISO) – 54/74L91, 8 bits

2. Serial in/Parallel out (SIPO) – 54/74164, 8 bits

3. Parallel in/Serial out (PISO) – 54/74265, 8 bits

4. Parallel in/Parallel out (PIPO) – 54/74198, 8 bits

Serial-In-Serial-Out Register:

From the name itself it is obvious that this type of register accepts data serially, i.e., one bit at a time at the single input line. The output is also obtained on a single output line in a serial fashion. The data within the register may be shifted from left to right using shift-left register, or may be shifted from right to left using shift-right register.

Serial-In–Parallel-Out Register:

In this type of register, the data is shifted in serially, but shifted out in parallel. To obtain the output data in parallel, it is required that all the output bits are available at the same time. This can be accomplished by connecting the output of each flip-flop to an output pin. Once the data is stored in the flip-flop the bits are available simultaneously.

Parallel-In–Serial-Out Register:

In the preceding two cases the data was shifted into the registers in a serial manner. We now can develop an idea for the parallel entry of data into the register. Here the data bits are entered into the flip-flops simultaneously, rather than a bit-by-bit basis.

Parallel-In–Parallel-Out Register:

There is a fourth type of register already before, which is designed such that data can be shifted into or out of the register in parallel. The parallel input of data has already been discussed in the preceding section of parallel-in–serial-out shift register. Also, in this type of register there is no interconnection between the flip-flops since no serial shifting is required. Hence, the moment the parallel entry of the data is accomplished the data will be available at the parallel outputs of the register.