Victoria University
of Bangladesh

Final Assessment

Md Bakhtiar Chowdhury

ID: 2121210061

Department: CSE

Semester: Summer -2022

Batch: 21th

Course Title: Computer Organization & Assembly Programming

Course Code: CSE 233

Submitted To:

Umme Khadiza Tithi

Lecturer, Department of Computer Science & Engineering

Victoria University of Bangladesh

Submission Date: 08 October, 2022

Answer to the question no 1(a)

 # Write Down number of operands in assembly language?
Answer:

Number of operands

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.

Answer to the question no 1(b)

 # Write Down Difference between programming, Low-level, Higher -level language?

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.

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)

 # Graphic 7 inches X 5 inches with 600dpi. Calculate the amount of memory required to store the graphic?

Answer to the question no 1(c)

Answer: Graphic 7 inch × 5 inches with 600 dpi. calculate the am Amount of memory required to store the graphic.

Answer:-

$(7 \times 600) \times (5 \times 600)$

$= 4200 \times 3000$ pixels

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

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

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

$= 1.5$ MB

Answer to the question no 2(a)

 # Define DMA controllers in a computer System?

Answer:

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

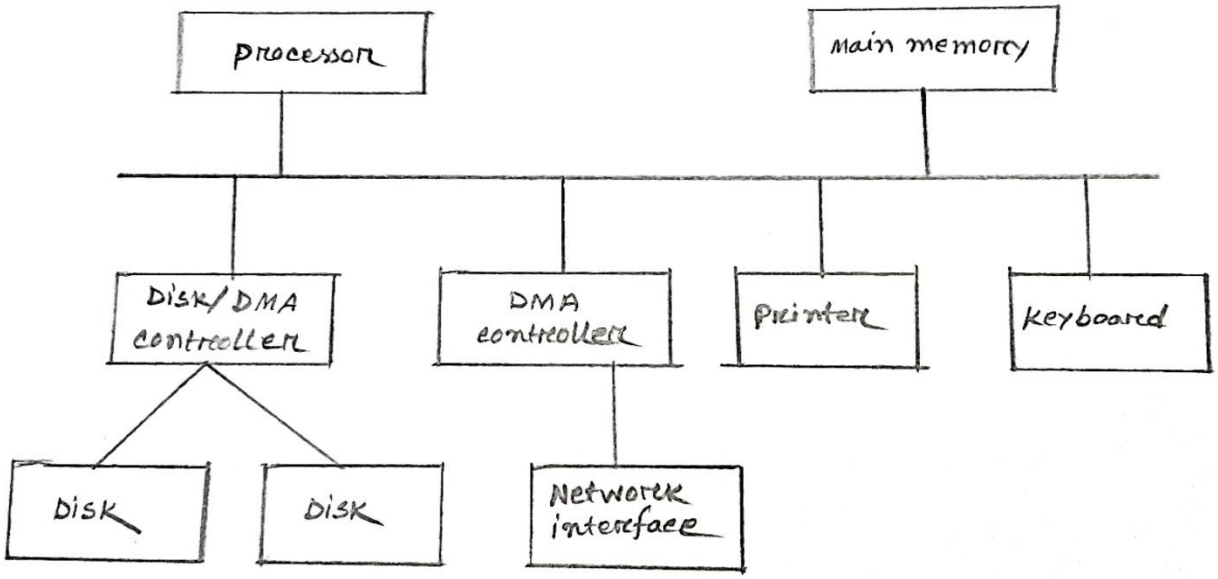


Fig:- DMA controllers in a computer system

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.

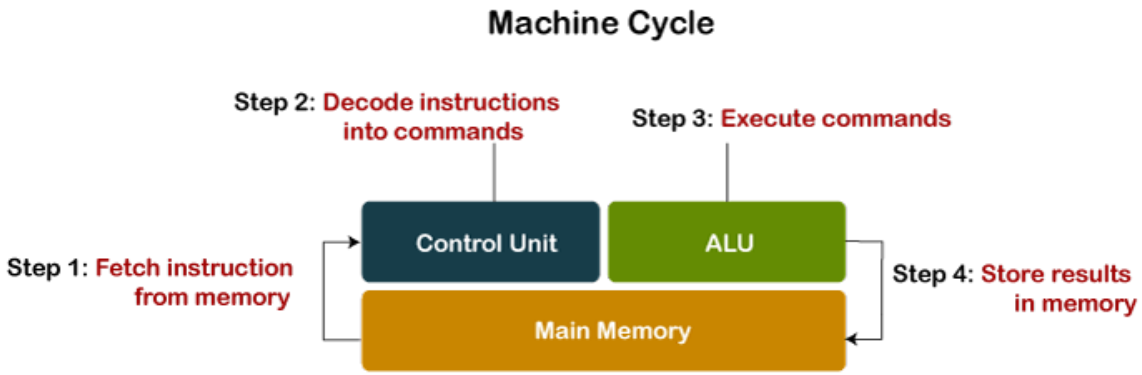
Answer to the question no 2(b)

 # Design simple units of ALU and characteristics of ALU?

Answer:

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.



Except performing calculations related to addition and subtraction, ALUs handle the multiplication of two integers as they are designed to execute integer calculations; hence, its result is also an integer. However, division operations commonly may not be performed by ALU as division operations may produce a result in a floating-point number. Instead, the floating-point unit (FPU) usually handles the division operations; other non-integer calculations can also be performed by FPU.

Additionally, engineers can design the ALU to perform any type of operation. However, ALU becomes more costly as the operations become more complex because ALU destroys more heat and takes up more space in the CPU. This is the reason to make powerful ALU by engineers, which provides the surety that the CPU is fast and powerful as well.

The calculations needed by the CPU are handled by the arithmetic logic unit (ALU); most of the operations among them are logical in nature. If the CPU is made more powerful, which is made on the basis of the ALU is designed. Then it creates more heat and takes more power or energy. Therefore, it must be moderation between how complex and powerful ALU is and not be more costly. This is the main reason the faster CPUs are more costly; hence, they take much power and destroy more heat. Arithmetic and logic operations are the main operations that are performed by the ALU; it also performs bit-shifting operations.

Although the ALU is a major component in the processor, the ALU's design and function may be different in the different processors. For case, some ALUs are designed to perform only integer calculations, and some are for floating-point operations. Some processors include a single arithmetic logic unit to perform operations, and others may contain numerous ALUs to complete calculations. The operations performed by ALU are:

- **Logical Operations:** The logical operations consist of NOR, NOT, AND, NAND, OR, XOR, and more.
- **Bit-Shifting Operations:** It is responsible for displacement in the locations of the bits to the by right or left by a certain number of places that are known as a multiplication operation.
- **Arithmetic Operations:** Although it performs multiplication and division, this refers to bit addition and subtraction. But multiplication and division operations are more costly to make. In the place of multiplication, addition can be used as a substitute and subtraction for division.

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.

Through the introduction of something called 'object-oriented programming,' these basic variables, which are the operands in many computer programs, have been invested with more detailed properties and characteristics, through ideas like programmed programming classes and arrays.

 # Characteristics of ALU

Arithmetic and logical units, or ALU, carry out operations including addition, subtraction, multiplication, and division. The task of controlling computer functions is carried out by the control unit, or CU. It oversees and provides all computer components with the necessary instructions.

Explanation:

The CU, or control unit, has the following characteristics:

- This component of the CPU is in charge of all operations that are carried out.
- It is in charge of directing the system to carry out commands.
- It facilitates communication between the arithmetic logical unit and the memory.
- It also helps with the necessary loading of information and instructions from the secondary memory to the main memory.

The following are the ALU's characteristics:

- The ALU is in charge of carrying out all logical and mathematical processes.
- The following are some examples of arithmetic operations: addition, subtraction, multiplication, and division.
- The following list of logical operations includes comparisons of numbers, letters, and/or special characters.
- The Equal-to conditions, Less-than conditions, and Greater-than conditions are likewise handled by the ALU.

Answer to the question no 2(c)

 # Convert 5G.AB216 into an equivalent binary number?

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)

 # describe current usage of assembly language?

Answer:

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,

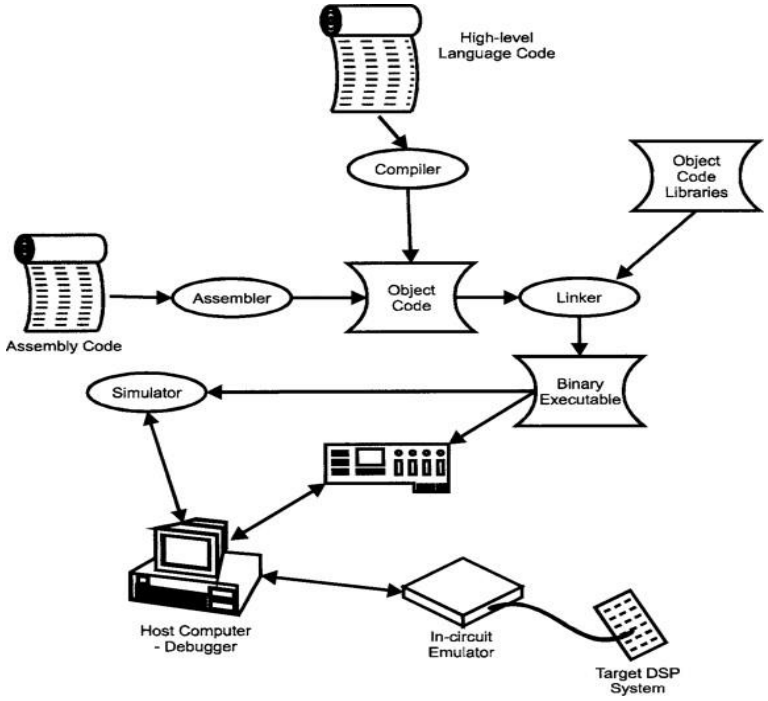
as well as assembly programmers. Moreover, and to the dismay of efficiency lovers, increasing processor performance has meant that most CPUs sit idle most of the time, with delays caused by predictable bottlenecks such as I/O operations and paging. This has made raw code execution speed a non-issue for 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 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)

 # discuss input output in assembly language program?
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 taken

MIX devices

Information on the devices for the hypothetical MIX processor's input/output instructions.

unit number	peripheral	block size	control
t	Tape unit no. i ($0 \le i \le 7$)	100 words	M=0, tape rewound; M < 0, skip back M records; M > 0, skip forward M records

d	Disk or drum unit no. d (8 ≤ d ≤ 15)	100 words	position device according to X-register (extension)
16	Card reader	16 words	
17	Card punch	16 words	
18	Printer	24 words	IOC 0(18) skips printer to top of following page
19	Typewriter and paper tape	14 words	paper tape reader: rewind tape

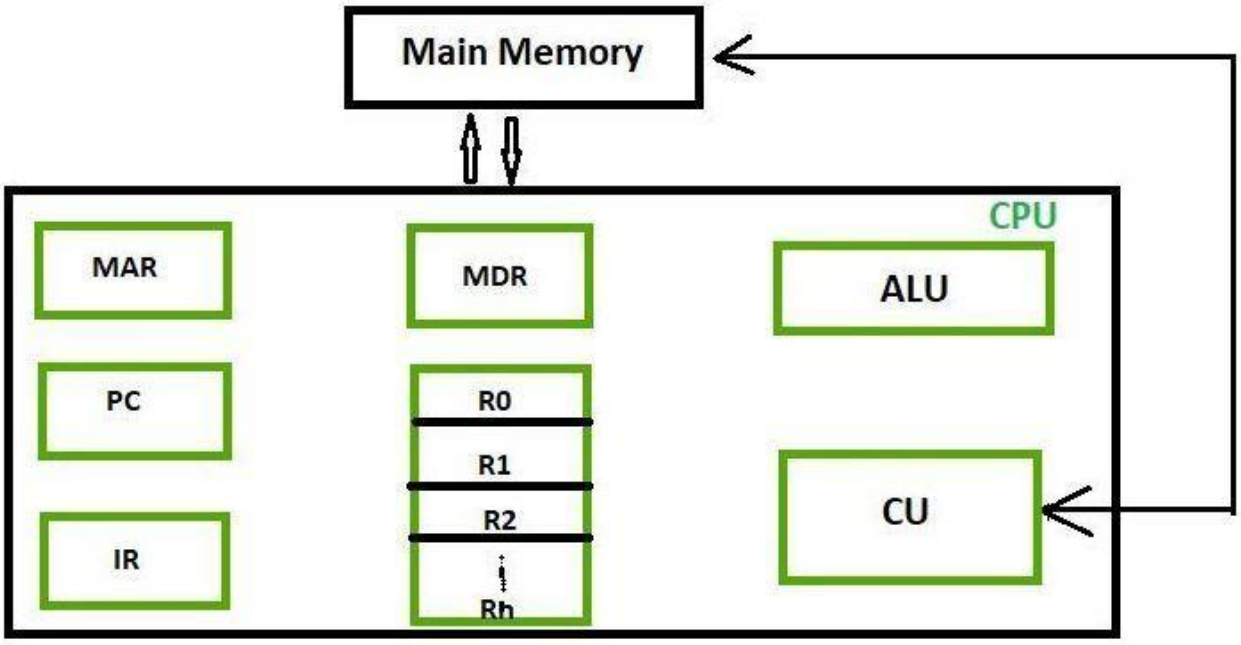
Answer to the question no 3(c)

 # describe different type of registers?

Answer:

In Computer Architecture, the Registers are very fast computer memory which are used to execute programs and operations efficiently. This does by giving access to commonly used values, i.e., the values which are in the point of operation/execution at that time. So, for this purpose, there are several different classes of CPU registers which works in coordination with the computer memory to run operations efficiently.

The sole purpose of having register is fast retrieval of data for processing by CPU. Though accessing instructions from RAM is comparatively faster with hard drive, it still isn't enough for CPU. For even better processing, there are memories in CPU which can get data from RAM which are about to be executed beforehand. After registers we have cache memory, which are faster but less faster than registers.



These are classified as given below.

- **Accumulator:**
  This is the most frequently used register used to store data taken from memory. It is in different numbers in different microprocessors.

- **Memory Address Registers (MAR):**
  It holds the address of the location to be accessed from memory. MAR and MDR (Memory Data Register) together facilitate the communication of the CPU and the main memory.

- **Memory Data Registers (MDR):**
  It contains data to be written into or to be read out from the addressed location.

- **General Purpose Registers:**
  These are numbered as R0, R1, R2…. Rn-1, and used to store temporary

data during any ongoing operation. Its content can be accessed by assembly programming. Modern CPU architectures tends to use more GPR so that register-to-register addressing can be used more, which is comparatively faster than other addressing modes.

- **Program Counter (PC):**
  Program Counter (PC) is used to keep the track of execution of the program. It contains the memory address of the next instruction to be fetched. PC points to the address of the next instruction to be fetched from the main memory when the previous instruction has been successfully completed. Program Counter (PC) also functions to count the number of instructions. The incrementation of PC depends on the type of architecture being used. If we are using 32-bit architecture, the PC gets incremented by 4 every time to fetch the next instruction.

- **Instruction Register (IR):**
  The IR holds the instruction which is just about to be executed. The instruction from PC is fetched and stored in IR. As soon as the instruction in placed in IR, the CPU starts executing the instruction and the PC points to the next instruction to be executed.

- **Condition code register (CCR) :**
  Condition code registers contain different flags that indicate the status of any operation. For instance let's suppose an operation caused creation of a negative result or zero, then these flags are set high accordingly. And the flags are

1. Carry C: Set to 1 if an add operation produces a carry or a subtract operation produces a borrow; otherwise cleared to 0.
2. Overflow V: Useful only during operations on signed integers.
3. Zero Z: Set to 1 if the result is 0, otherwise cleared to 0.

4. Negate N: Meaningful only in signed number operations. Set to 1 if a negative result is produced.
5. Extend X: Functions as a carry for multiple precision arithmetic operations.

These are generally decided by ALU.

So, these are the different registers which are operating for a specific purpose.

>>>>End<<<<