Unit 3

Explain about micro operations with examples

here are some examples of microoperations with examples:

- Load: This microoperation loads the value from a memory location into a register. For example, to load the value from memory location 0x1000 into register R0, the microoperation would be:

Code snippet
```
0 1 0 0 0 0 0 1
```

- Store: This microoperation stores the value from a register into a memory location. For example, to store the value in register R0 into memory location 0x1000, the microoperation would be:

Code snippet
```
0 1 0 1 0 0 0 0
```

- Add: This microoperation adds the values of two registers and stores the result in a third register. For example, to add the values of registers R0 and R1 and store the result in register R2, the microoperation would be:

Code snippet
```
0 0 0 0 0 1 0 0
```

- Subtract: This microoperation subtracts the value of one register from the value of another register and stores the result in a third register. For example, to subtract the value of register R1 from the value of register R0 and store the result in register R2, the microoperation would be:

Code snippet
```
0 0 0 0 0 1 1 0
```

- Multiply: This microoperation multiplies the values of two registers and stores the result in a third register. For example, to multiply the values of registers R0 and R1 and store the result in register R2, the microoperation would be:

Code snippet
```
0 0 0 1 0 0 0 0
```

- Divide: This microoperation divides the value of one register by the value of another register and stores the result in a third register. For example, to divide the value of register R1 by the value of register R0 and store the result in register R2, the microoperation would be:

Code snippet
```
0 0 0 1 0 0 1 0
```

- AND: This microoperation performs a logical AND operation on the values of two registers and stores the result in a third register. For example, to perform a logical AND operation on the values of registers R0 and R1 and store the result in register R2, the microoperation would be:

Code snippet
```
0 0 1 0 0 0 0 0
```

- OR: This microoperation performs a logical OR operation on the values of two registers and stores the result in a third register. For example, to perform a logical OR operation on the values of registers R0 and R1 and store the result in register R2, the microoperation would be:

Code snippet
```
0 0 1 0 0 0 0 1
```

- XOR: This microoperation performs a logical XOR operation on the values of two registers and stores the result in a third register. For example, to perform a logical XOR operation on the values of registers R0 and R1 and store the result in register R2, the microoperation would be:

Code snippet
```
0 0 1 0 0 0 1 0
```

- Shift: This microoperation shifts the bits of a register to the left or right by a specified number of places. For example, to shift the bits of register R0 to the left by 2 places, the microoperation would be:

Code snippet
```
0 0 1 1 0 0 0 0
```

- Rotate: This microoperation rotates the bits of a register to the left or right by a specified number of places. For example, to rotate the bits of register R0 to the right by 2 places, the microoperation would be:

Code snippet
```
0 0 1 1 1 0 0 0
```

These are just a few examples of microoperations. There are many other types of microoperations that can be performed, depending on the design of the computer.

Data Transfer Instructions and Data Manipulations and Program Control

Data Transfer Instruction

Data transfer instructions are used to move data between memory and processor registers, processor registers, and I/O devices, and from one processor register to another. There are eight commonly used data transfer instructions:

- Move (MOV): This instruction copies data from one location to another. The source and destination can be a register, a memory location, or an immediate value.
- Load (LD): This instruction copies data from a memory location to a register. The source is always a memory location, and the destination is always a register.
- Store (ST): This instruction copies data from a register to a memory location. The source is always a register, and the destination is always a memory location.
- Exchange (XCHG): This instruction exchanges the contents of two registers. The two registers can be the same or different.
- Input (IN): This instruction copies data from an I/O device to a register. The source is always an I/O device, and the destination is always a register.
- Output (OUT): This instruction copies data from a register to an I/O device. The source is always a register, and the destination is always an I/O device.
- Push (PUSH): This instruction pushes data onto the stack. The data can be from a register or a memory location.
- Pop (POP): This instruction pops data off the stack and stores it in a register.

Data transfer instructions are an essential part of computer programming. They are used to move data around in the computer system, which is necessary for performing any kind of computation.

Here are some examples of data transfer instructions:

- MOV A, 10 This instruction copies the value 10 to the accumulator register A.
- LDA 100H This instruction loads the value at memory location 100H into the accumulator register A.
- STA 200H This instruction stores the value in the accumulator register A to memory location 200H.
- IN 1 This instruction reads a byte of data from input port 1 and stores it in the accumulator register A.
- OUT 2 This instruction writes a byte of data from the accumulator register A to output port 2.

.

Data transfer instructions are typically very fast, since they do not involve any computation. This makes them ideal for moving large amounts of data around in the computer system.

Data Manipulation

Data manipulation instructions are a type of instruction that performs operations on data. They are used to perform arithmetic, logical, and shift operations on data.

- Arithmetic instructions perform mathematical operations on data, such as addition, subtraction, multiplication, and division.
- Logical instructions perform logical operations on data, such as AND, OR, XOR, and NOT.
- Shift instructions shift the bits in a data value to the left or right by a specified number of places.

Data manipulation instructions are essential for computer programming. They allow programs to perform calculations, make decisions, and manipulate data.

Here are some examples of data manipulation instructions:

- ADD A, B This instruction adds the value in register B to the value in register A and stores the result in register A.
- SUB A, B This instruction subtracts the value in register B from the value in register A and stores the result in register A.
- MUL A, B This instruction multiplies the value in register B by the value in register A and stores the result in register A.
- DIV A, B This instruction divides the value in register A by the value in register B and stores the result in register A.
- AND A, B This instruction performs an AND operation on the value in register A and the value in register B and stores the result in register A.
- OR A, B This instruction performs an OR operation on the value in register A and the value in register B and stores the result in register A.
- XOR A, B This instruction performs an XOR operation on the value in register A and the value in register B and stores the result in register A.
- SHL A, 1 This instruction shifts the bits in register A one place to the left.
- SHR A, 1 This instruction shifts the bits in register A one place to the right.

Data manipulation instructions are an important part of computer programming. They allow programs to perform calculations, make decisions, and manipulate data.

Program Control

Program control in computer organization refers to the management of the flow of instructions within a program. It involves making decisions, branching to different parts of the program, and repeating sections of code based on certain conditions. Program control instructions play a crucial role in directing the execution of a program. Here are some common instructions used for program control:

1. Conditional Branch Instructions:

These instructions allow the program to make decisions based on conditions. They evaluate a condition and branch to a different part of the program depending on the result. Examples include:

- Branch If Equal (BEQ): Branches to a specified address if two values are equal.
- Branch If Not Equal (BNE): Branches to a specified address if two values are not equal.
- Branch If Greater Than (BGT): Branches to a specified address if one value is greater than another.

2. Unconditional Branch Instructions:

These instructions unconditionally transfer the program execution to a different location. They are used for loops, subroutines, and other control structures. Examples include:

- Branch (B): Jumps to a specified address unconditionally.
- Jump and Link (JAL): Jumps to a specified address and stores the return address in a register for subroutine calls.

3. Subroutine Instructions:

These instructions facilitate the use of subroutines or functions. They allow the program to call a section of code, execute it, and then return to the original calling location. Examples include:

- Call (CALL): Transfers control to a subroutine and saves the return address.
- Return (RET): Returns from a subroutine to the caller.

4. Loop Control Instructions:

These instructions are used to repeat a section of code multiple times, often referred to as loops. They typically involve decrementing a counter and checking for the loop condition. Examples include:

- Loop (LOOP): Decrements a counter and jumps to a specified address if the counter is not zero.
- Jump If Zero (JZ): Jumps to a specified address if a register contains zero.

5. Interrupt Instructions:

Interrupts are a mechanism that allows the program execution to be temporarily suspended to handle an event. Interrupt instructions enable the program to respond to various events and handle them accordingly.

These are just a few examples of program control instructions commonly found in computer architectures. Different instruction sets and architectures may have their own variations or additional instructions tailored to specific needs. The arrangement and combination of these instructions dictate the flow of control within a program and enable the execution of complex tasks.

Q:) Explain About Address Modes

Addressing modes in computer organization determine how the CPU accesses data. They define the rules for finding and using the data needed for instructions.

There are different addressing modes:

1. Immediate Mode: The data is directly given in the instruction.
Example: ADD R1, #5 (Add 5 to R1).

2. Register Mode: The data is in a CPU register.
Example: ADD R1, R2 (Add the value in R2 to R1).

3. Direct Mode: The data is at a specific memory location.
Example: LOAD R1, 1000 (Load the value from memory location 1000 into R1).

4. Indirect Mode: The data is at a memory location pointed to by a given address.
Example: LOAD R1, (1000) (Load the value from the memory location whose address is stored at location 1000 into R1).

5. Indexed Mode: The data is at a memory location calculated by adding an index or offset to a base address.
Example: LOAD R1, (R2 + 5) (Load the value from the memory location calculated by adding 5 to the value in R2 into R1).

6. Relative Mode: The data is at a memory location relative to the current program counter.

Example: JUMP 100 (Jump to the instruction at a location calculated by adding 100 to the current program counter).

These addressing modes help computers access and manipulate data efficiently and flexibly.

Explain about instruction formats in simple words ?

Sure! Imagine you have a really big toy box filled with different types of toys. Each toy has its own special way of playing with it. That's kind of how computers work too!
In a computer, the "toy box" is called memory, and inside it, we have special boxes called instructions. These instructions tell the computer what to do, just like when someone tells you how to play with a toy.
Now, these instructions are written in a special language that only computers can understand. But don't worry, we can break it down into simpler parts!
Think of each instruction as a sentence made up of words. Each word in the sentence has a specific meaning. For example, let's take the sentence "Put the red block on the table."
In a computer, an instruction could look like this: "ADD number1, number2, result." Here, "ADD" is like the word "Put" in our sentence. It tells the computer that we want to add two numbers together.
Now, the words "number1," "number2," and "result" are like the names of the toys. They tell the computer which numbers to add and where to put the result. It's just like saying "the red block" and "on the table" in our sentence.

But computers are really smart and can understand different types of instructions. Some instructions might be shorter, like "MOVE number1, result." Here, instead of adding, we're moving a number to a different place. So, just like you have different toys with different ways to play, computers have different instructions with different ways to do things. The instruction format is like a special set of rules that tells the computer how to understand and follow each instruction.
So, in summary, instruction formats are like the special language computers use to understand instructions. They're like sentences made up of words, where each word has a special meaning. It helps the computer know what to do with the numbers and where to put the results, just like you know how to play with different toys!

Explain about Risc and it’s Characteristics

Q) What is Risc and explain it ?

RISC stands for Reduced Instruction Set Computing. It is a type of computer architecture that uses a small, simple instruction set. RISC instructions are typically one or two bytes long, and they perform a single, basic operation. This makes RISC instructions easy to decode and execute, which can lead to faster performance.

Here are some of the key properties of RISC:

- `Small instruction set`: RISC architectures have a small number of instructions. This makes it easier for the processor to decode and execute instructions, which can lead to faster performance.
- `Fixed-length instructions`: RISC instructions are all the same length,

typically one or two bytes. This makes it easier for the processor to fetch and decode instructions, which can also lead to faster performance.

- `Simple instructions`: RISC instructions are typically simple and perform a single, basic operation. This makes them easier to decode and execute, which can also lead to faster performance.
- `Pipelined execution`: RISC processors can often execute multiple instructions in parallel. This is done by dividing the execution of each instruction into a number of stages, and then executing each stage of multiple instructions at the same time.

RISC architectures have become increasingly popular in recent years. They are now used in a wide range of devices, including personal computers, smartphones, and tablets.

Here are some of the advantages of RISC architectures:

- Faster performance: RISC architectures can often execute instructions faster than CISC architectures. This is because RISC instructions are simpler and easier to decode and execute.
- Simpler design: RISC architectures are simpler to design and implement than CISC architectures. This makes them less expensive to manufacture, and it also makes them easier to debug and maintain.
- More efficient use of power: RISC architectures can often use power more efficiently than CISC architectures. This is because RISC instructions are simpler and require less processing time.

Here are some of the disadvantages of RISC architectures:

- Limited functionality: RISC architectures may not have as many instructions as CISC architectures. This can make them less suitable for some applications, such as scientific computing.
- More complex software: RISC software may be more complex than CISC software. This is because RISC instructions are simpler and require more instructions to perform the same task.
- Less compatibility: RISC software may not be compatible with CISC

software. This can make it difficult to run CISC software on a RISC machine.