UNIT-2 ARITHMATIC OPERATIONS AND ADDRESSING MODES

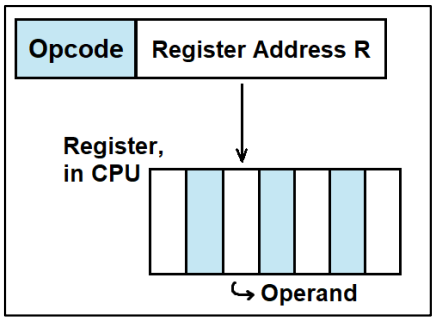
ADDRESSING MODES:

- In computer architecture, addressing modes are mechanisms that determine how the operands of an instruction are accessed. It specifies the way in which the processor interprets the address or location of an operand in memory or a register.
- Instructions in a computer's instruction set architecture (ISA) typically have one or more operands, which can be data values or memory addresses.
- The addressing mode determines how the processor calculates or determines the effective address of an operand. The **effective address** is the actual location where the data is stored or retrieved.

Types of addressing modes:

1. **Register Mode:** The operand is specified by a register's content.

   Example: **ADD R1, R2** (Add the content of register R2 to the content of register R1)

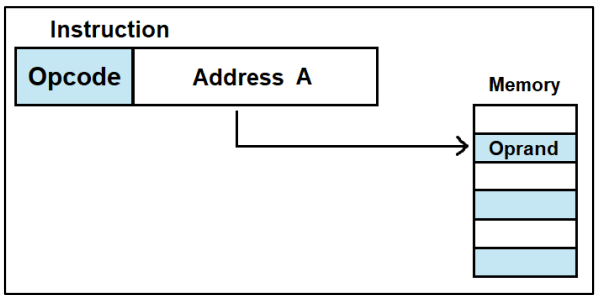


2. **Immediate Mode:** The operand is directly specified within the instruction itself.

   Example: **ADD R1, #5** (Add the immediate value 5 to the content of register R1)

3. **Direct Mode:** The operand is the address of a memory location where the data resides.

   Example: **LOAD R1, [1000]** (Load content of memory location 1000 into register R1)



4. **Indirect Mode:** The operand contains the address of a memory location that holds the actual address of the data.

   Example: **LOAD R1, [R2]** (Load the content of the memory location whose address is stored in register R2 into register R1)

5. **Indexed Mode:** The operand is obtained by adding a constant or a register value to a base address. This mode is often used for accessing elements in arrays or data structures.

   Example: **LOAD R1, [R2+5]** (Load the content of the memory location at the address R2+5 into register R1)

6. **Implied Mode:** It is also called inherent/implicit addressing mode. The operand is implied by the instruction. The operand is hidden/fixed inside the instruction. Example:

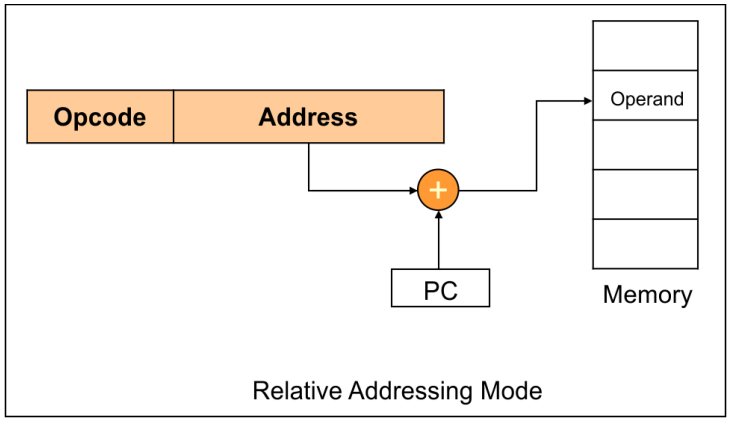
   Complement Accumulator **CMA** (Here accumulator A is implied by the instruction).

   Complement Carry Flag **CMC** (Here Flags register is implied by the instruction)

   Set Carry Flag **STC** (Here Flags register is implied by the instruction)

7. **Relative Mode:** The effective address of the operand is obtained by adding the address of the program counter with the address part of the instruction.

   Effective Address = Value of PC + Address part of the instruction



   Example: **JMP 100** (Perform an unconditional jump to the instruction located at the address PC+100)

8. **Stack Mode:** The operand is accessed from the top of the stack.

   Example: **POP R1** (Pop the top value from the stack and store it in register R1)

9. **Auto-Increment Mode:** The operand is accessed from a memory location, and the address is automatically incremented after the access.

   Example: **LOAD R1, [R2++]** (Load the content of the memory location at the address stored in R2 into register R1 and then increment R2)

10. **Auto-Decrement Mode:** The operand is accessed from a memory location, and the address is automatically decremented after the access.

    Example: **LOAD R1, [--R2]** (Decrement R2 and then load the content of the memory location at the new address into register R1)

Detail explanation of 5 addressing mode which are in syllabus:

REGISTER, IMMEDIATE, DIRECT, INDIRECT, INDEXED.

1. REGISTER ADDRESSING MODES:

- **Register addressing mode** is a type of addressing mode used in computer architectures where the operand is specified as a register.
- In this mode, the instruction operates directly on the contents of specified register, without the need to access memory. It is one of the simplest and most efficient addressing modes.
- In this mode, the instruction specifies the source and/or destination registers where the data is stored or retrieved.
- The data is manipulated within the registers themselves, which are small, high-speed storage locations within the processor.

Example: ADD R1, R2

In this instruction, the ADD operation adds the contents of register R2 to the contents of register R1. The result is stored back in register R1.

Here's how the Register Addressing Mode works:

1. **Instruction Fetch:** The processor fetches the instruction from memory, which contains the opcode (operation code) and register operands.
2. **Decode:** The processor decodes the instruction to determine the operation to be performed and the registers involved.
3. **Operand Access:** The processor accesses the source registers specified in the instruction. It retrieves the contents of these registers from the register file, which is a set of registers within the processor.
4. **Operation:** The processor performs specified operation on the data within the source registers. In the case of our example, it adds the contents of R2 to the contents of R1.
5. **Result Store:** The processor stores the result of the operation back into the destination register, which is R1 in this case.

Advantages of Register Addressing Mode:

1. **Speed:** Register operations are typically faster than memory operations because registers are located within the processor and have very fast access times. This improves the overall execution speed of the instruction.
2. **Efficiency:** Register operations do not require accessing memory, which reduces the memory access time. It saves memory-related instructions and reduces memory traffic.
3. **Code Size:** Using registers directly as operands in instructions helps in reducing the size of the instruction code. Since registers are represented by small identifiers, fewer bits are needed to specify them compared to memory addresses.

Disadvantages of Register Addressing Mode:

1. **Limited Register Space:** Processors have a limited number of registers available for use. Depending on the architecture, the number of registers can range from a few to several dozen. Therefore, register addressing mode may impose limitations on the complexity and size of programs.
2. **Data Sharing:** Since registers are shared resources, multiple instructions may require the same register for their operations. This can lead to register conflicts or dependencies that need to be managed by the processor.

2. IMMEDIATE ADDRESSING MODES:

- **Immediate addressing mode** is a type of addressing mode used in computer architectures where the operand value is directly specified within the instruction itself, rather than referring to a memory location or register. In this mode, the operand is a constant or immediate value that is part of the instruction itself.

Example: ADD R1, #5 (Add the immediate value 5 to the content of register R1)

Here's how the Immediate Addressing Mode works:

1. **Instruction Fetch:** The processor fetches the instruction from memory, which contains the opcode (operation code) and the immediate value.
2. **Decode:** The processor decodes instruction to determine the operation to be performed.
3. **Operand Access:** In this mode, the operand is directly available within the instruction itself. The immediate value is extracted from the instruction and treated as the operand.
4. **Operation:** The processor performs the specified operation using the immediate value as one of the operands.
5. **Result Store:** The result of the operation is stored in the destination register or memory location, depending on the instruction.

Advantages of Immediate Addressing Mode:

1. **Compact Code:** Since the operand value is directly specified within the instruction, it eliminates the need for separate memory accesses or register operations. This reduces the code size and memory requirements.
2. **Efficiency:** Immediate addressing mode can be faster than other addressing modes that involve memory accesses. The immediate value is readily available in the instruction, eliminating the need to fetch data from memory.
3. **Constant Values:** Immediate addressing mode is commonly used for instructions that involve constant values or immediate data, such as adding a fixed offset or performing logical operations on constants.

Disadvantages of Immediate Addressing Mode:

1. **Limited Range:** The immediate value is typically limited by the number of bits allocated to represent it within the instruction. As a result, the range of immediate values that can be used is limited.
2. **Data Immutability:** Since the immediate value is directly specified in the instruction, it cannot be modified or updated during program execution. This limits the flexibility of using dynamic or variable data as operands.
3. **Memory Access:** Immediate addressing mode is not suitable for instructions that require large data values or operands stored in memory. In such cases, other addressing modes like direct or indirect addressing may be more appropriate.

3. DIRECT ADRESSING MODES:

- **Direct addressing mode** is a type of addressing mode used in computer architectures where the operand is specified by a memory address directly. In this mode, the instruction operates on the data stored at the memory location specified by the address.

Example: LOAD R1, [1000] (Load content of memory location 1000 into register R1)

Here's how the Direct Addressing Mode works:

1. **Instruction Fetch:** The processor fetches the instruction from memory, which contains the opcode (operation code) and the memory address of the operand.
2. **Decode:** The processor decodes the instruction to determine operation to be performed.
3. **Operand Access:** In this, the operand is directly specified as a memory address in the instruction. The processor uses this address to access the data stored in memory.
4. **Memory Access:** The processor accesses the memory location specified by the address and retrieves the data stored there. This data becomes the operand for the instruction.
5. **Operation:** The processor performs the specified operation using the operand obtained from memory.
6. **Result Store:** The result of the operation is stored in the destination register or memory location, depending on the instruction.

Advantages of Direct Addressing Mode:

1. **Memory Access:** It allows instructions to operate directly on data stored in memory. This is useful for accessing data elements in arrays or other data structures stored in consecutive memory locations.
2. **Flexibility:** By specifying the memory address directly, it provides flexibility in accessing any desired memory location.

Disadvantages of Direct Addressing Mode:

1. **Memory Latency:** Accessing data from memory typically takes more time compared to accessing registers. This can introduce a latency delay in instruction execution, as the processor needs to wait for the memory access to complete.
2. **Memory Space:** The range of memory addresses that can be directly accessed is limited by the size of the address field in the instruction format. The size of the address field determines the maximum memory address that can be specified.

4. INDIRECT ADDRESSING MODES:

- **Indirect addressing mode** is a type of addressing mode used in computer architectures where the operand specifies a memory address that contains the actual data address. In this mode, the instruction operates on the data stored at the memory address indirectly referenced by the operand.

Example: LOAD R1, [R2]

(Load the content of memory location whose address is stored in register R2 into register R1)

Here's how the Indirect Addressing Mode works:

1. **Instruction Fetch:** The processor fetches instruction from memory, which contains the opcode (operation code) and memory address or register that holds the indirect address.
2. **Decode:** The processor decodes instruction to determine the operation to be performed.
3. **Operand Access:** In this, the operand specifies a memory address or register that holds address of the data. The processor accesses indirect address specified in the instruction.
4. **Memory Access:** The processor accesses the memory location specified by the indirect address and retrieves the actual data address stored there.
5. **Data Access:** Using the obtained data address, the processor accesses memory again to retrieve actual data stored at that address. This data becomes operand for the instruction.
6. **Operation:** The processor performs the specified operation using the operand obtained from memory.
7. **Result Store:** The result of the operation is stored in the destination register or memory location, depending on the instruction.

Advantages of Indirect Addressing Mode:

1. **Flexibility:** It allows for flexible memory access. By referencing a memory address stored elsewhere, it provides ability to access different memory locations dynamically.
2. **Indirection:** Indirect addressing mode enables multiple levels of indirection.
3. **Memory Management:** It can be helpful in managing memory dynamically.

Disadvantages of Indirect Addressing Mode:

1. **Additional Memory Access:** Indirect addressing mode requires an additional memory access to retrieve the actual data address. This introduces some latency and can impact the overall performance of the instruction.
2. **Complexity:** Indirect addressing mode adds complexity to the instruction execution process. The processor needs to follow the indirection chain to obtain the actual data address before performing the operation. This complexity can make the instruction execution slower and more prone to errors if not managed properly.

5. INDEXED ADDRESSING MODES:

- Indexed addressing mode is a type of addressing mode in computer architecture where effective address of an operand is calculated by adding an index/offset to a base address.
- This mode is particularly useful when accessing elements in arrays or data structures where the address calculation involves a fixed displacement from a starting address.

Example: Suppose we have an array of integers starting at memory address 1000, and we want to load the value of the element at index 3 into a register.

LDR R1, [R2, #12]

In this example, R2 is the base register that holds the starting address of the array (1000). The index or offset is 3, and the size of each array element is assumed to be 4 bytes (hence the offset of 12).

The calculation for the effective address is as follows:

Effective Address = Base Address + (Index * Size of Element) = 1000 + (3 * 4) = 1012

The instruction **LDR R1, [R2, #12]** loads the value stored at memory address 1012 into register R1. By using the indexed addressing mode, we can access the desired array element without explicitly specifying the absolute memory address.

Advantages of Indexed Addressing Mode:

1. **Efficient array access:** It is particularly useful for accessing elements in arrays or data structures. By using an index or offset, the effective address can be calculated easily, allowing efficient and direct access to array elements.
2. **Flexibility:** It provides flexibility in addressing different elements within an array. By changing the index or offset, different elements can be accessed without modifying the instruction itself.
3. **Code reusability:** With indexed addressing mode, the same instruction can be used to access different elements at different indices, promoting code reusability and reducing the need for repetitive instructions.

Disadvantages of Indexed Addressing Mode:

1. **Fixed offset/Index size:** Indexed addressing mode typically assumes a fixed offset or index size. This limitation restricts the range of elements that can be accessed using the addressing mode.
2. **Limited addressing range:** The addressing range of indexed addressing mode is limited by the maximum value that can be represented by the offset or index.
3. **Reduced flexibility for non-array data structures:** While indexed addressing mode excels at array access, it may not be as flexible for accessing non-array data structures or irregularly structured data.
4. **Increased complexity for multidimensional arrays:** Indexed addressing mode becomes more complex when dealing with multidimensional arrays.

OPERATIONS IN INSTRUCTION SET:

- Operations in an instruction set refer to the specific tasks or actions that can be performed by a computer's CPU through the instructions provided by the instruction set architecture (ISA). Each operation represents fundamental operation or task that the CPU can execute.
- Instructions in an instruction set typically consist of an opcode (operation code) that specifies the operation to be performed and may include operands that provide additional information or data for the operation.
- The CPU fetches instructions from memory, decodes them to determine the operation to be performed, and then executes the corresponding operation.

Here are some common operations that can be found in an instruction set:

1. **Arithmetic Operations:** These operations perform mathematical calculations on data, such as add, subtract, multiply, and divide. For example, ADD, SUB, MUL, DIV.
2. **Logical Operations:** These operations manipulate binary data using logical operations, such as AND, OR, XOR, and NOT. They are used for tasks like bitwise manipulation and boolean logic.
3. **Data Transfer Operations:** These operations move data between registers, memory, and other storage locations. Examples include LOAD (move data from memory to a register), STORE (move data from a register to memory), and MOVE (copy data between registers).
4. **Control Transfer Operations:** These operations control the flow of program execution, enabling branching and looping. They include instructions like JUMP (unconditional jump), BRANCH (conditional jump), and CALL/RETURN (subroutine calls and returns).

5. **Comparison Operations:** These operations compare two values and set flags or registers based on the result. They are often used in conditional branching and decision-making. Examples include CMP (compare), TEST (perform bitwise AND and set flags), and SET (set a register or flag based on a condition).
6. **Shift and Rotate Operations:** These operations shift or rotate the bits of a value in a register, allowing for logical or arithmetic shifts. Examples include SHIFT LEFT, SHIFT RIGHT, ROTATE LEFT, and ROTATE RIGHT.
7. **Input/ Output Operations:** These operations facilitate communication between the CPU and input/output devices, such as reading from or writing to a disk, keyboard, or network. They typically involve instructions like READ and WRITE.
8. **Floating-Point Operations:** These operations handle floating-point arithmetic for tasks that require high precision, such as scientific calculations. They include operations like FADD, FSUB, FMUL, and FDIV for floating-point numbers.

1. ARITHMATIC OPERATIONS:

- Arithmetic operations in computer architecture refer to the set of instructions that perform mathematical calculations on data within the central processing unit (CPU). These instructions operate on numerical data, such as integers or floating-point numbers, and allow CPU to perform tasks like addition, subtraction, multiplication, and division.

Here are some common arithmetic operations found in an instruction set:

1. **Addition (ADD):** The ADD instruction adds two values together and stores the result in a destination location. For example,

   ADD R1, R2 adds the contents of register R1 and R2, and stores the sum in register R1.

2. **Subtraction (SUB):** The SUB instruction subtracts one value from another and stores the result in a destination location. For example,

   SUB R1, R2 subtracts the content of register R2 from R1, and stores the difference in register R1.

3. **Multiplication (MUL):** The MUL instruction multiplies two values together and stores the product in a destination location. For example,

   MUL R1, R2 multiplies the content of register R1 and R2, and stores the result in register R1.

4. **Division (DIV):** The DIV instruction divides one value by another and stores the quotient in a destination location. For example,

   DIV R1, R2 divides the content of register R1 by R2, and stores quotient in register R1.

5. **Increment (INC) and Decrement (DEC):** These instructions increment or decrement the value of a register or memory location by one. For example, **INC R** increases content of register R1 by one. **DEC R** decreases content of register R1 by one.
6. **Negation (NEG):** This instruction changes sign of a value, by multiplying it by -1.
7. **Overflow and Carry Operations:** These operations handle overflow and carry conditions that may occur during arithmetic operations. They set flags or registers to indicate whether an operation resulted in overflow or carry.

2. LOGICAL OPERATIONS:

- Logical operations refer to the set of instructions that manipulate binary data at bit level. These instructions operate on binary values, typically stored in registers, and allow the CPU to perform tasks like bitwise, logical operations and data manipulation.

Here are some common logical operations found in an instruction set:

1. **Bitwise AND (AND):** The AND instruction performs a logical AND operation between two binary values and stores the result in a destination location. It sets each bit of the result to 1 only if the corresponding bits of both operands are 1. For Example,
    - **AND R1, R2** performs a bitwise AND operation between the contents of registers R1 and R2, and stores the result in register R1.
    - **ANA B** performs a logical AND operation between the contents of the accumulator and the contents of the B register.
2. **Bitwise OR (OR):** The OR instruction performs a logical OR operation between two binary values and stores the result in a destination location. It sets each bit of the result to 1 if at least one of the corresponding bits in the operands is 1. For example,
    - **OR R1, R2** performs a bitwise OR operation between the contents of registers R1 and R2, and stores the result in register R1.
    - **ORA C** performs a logical OR operation between the contents of the accumulator and the contents of the C register.
3. **Bitwise XOR (XOR):** The XOR instruction performs a logical exclusive OR operation between two binary values and stores the result in a destination location. It sets each bit of the result to 1 if the corresponding bits in the operands are different. For example,
    - **XOR R1, R2** performs a bitwise XOR operation between the contents of registers R1 and R2, and stores the result in register R1.
    - **XRA M** performs a logical XOR operation between contents of the accumulator and the contents of the memory location pointed to by the HL register.

4. **Bitwise NOT (NOT):** The NOT instruction performs a logical negation operation on a binary value and stores the result in a destination location. It flips each bit of the operand, turning 1s into 0s and 0s into 1s. For example,
    - **NOT R1** performs a bitwise NOT operation on the content of register R1 and stores the result in register R1.

In addition to these basic logical operations, some instruction sets may provide additional instructions for more specialized logical operations, such as **shifting and rotating bits**, **testing specific bits**, or **setting** or **clearing individual bits** within a register.

3. DATA TRANSFER OPERATIONS:

- Data transfer operations in computer architecture refer to set of instructions that facilitate movement of data between registers, memory, & other storage locations within the CPU.
- They allows the CPU to read data from memory, write data to memory, move data between registers, and perform other operations related to data transfer and manipulation.

Here are some common data transfer operations found in an instruction set:

1. **Load (LD):** The LD instruction transfers data from a memory location into a register. It specifies the memory location from which the data is loaded and the register that receives the data. For example,

   LD R1, [1000] loads the content of memory location 1000 into register R1.

2. **Store (ST):** The ST instruction transfers data from a register to a memory location. It specifies the register that contains the data to be stored and the memory location where the data should be written. For example,

   ST R1, [2000] stores the content of register R1 into memory location 2000.

3. **Move (MOV):** The MOV instruction copies data from one register to another register or from a register to a memory location, or vice versa. It specifies the source and destination operands for the data transfer. For example,

   MOV R1, R2 copies the content of register R2 into register R1.

4. **Push (PUSH) and Pop (POP):** These instructions are used for stack-based data transfer. **PUSH** places a value onto the top of the stack, and **POP** retrieves the value from the top of the stack into a register or memory location.

5. **Input/Output (I/O) Operations:** These operations facilitate communication between the CPU and input/output devices, such as reading from or writing to a disk, keyboard, network, or other peripherals. They typically involve instructions like **READ** and **WRITE**, which transfer data between the CPU and the I/O device.

6. **Exchange (XCHG):** This instruction exchanges the content of two registers or a register and a memory location. It is used for swapping data between two storage locations.
7. **Block Transfer:** Some instruction sets include specialized instructions for transferring blocks of data between memory and registers. These instructions allow efficient movement of large amounts of data by specifying a source address, a destination address, and a count of bytes or elements to be transferred.

The data transfer instructions move data b/w registers or b/w memory and registers.

Opcode	Operand	Data transfer instructions	Explanation	Example
MOV	Rd, Rs	Move	Rd=Rs	MOV A, B
MOV	Rd, M	Move	Rd=Mc	MOV A, 2050
MOV	M, Rs	Move	M=Rs	MOV 2050, A
MVI	Rd, 8-bit data	Move Immediate	Rd=8-bit data	MVI A, 50
MVI	M, 8-bit data	Move Immediate	M=8-bit data	MVI 2050, 50
LDA	16-bit address	Load Accumulator Directly from Memory	A=contents at address	LDA 2050
STA	16-bit address	Store Accumulator Directly in Memory	Contents at address=A	STA 2050

4. CONTROL FLOW OPERATIONS: (Also known as Control Transfer Operations)

- Control flow operations are instructions that control the flow of program execution.
- They allow the central processing unit (CPU) to alter the order of instruction execution, perform conditional branching, and facilitate subroutine calls and returns.
- Control flow operations are essential for implementing decision-making, loops, function calls, and other control structures within a program.

Here are some common control flow operations found in an instruction set:

1. **Jump (JMP):** It unconditionally transfers control to a different memory address or instruction. It allows for direct, unconditional branching to a specified location. For example, **JMP 100** transfers control to the instruction located at memory address 100.
2. **Branch (conditional jump):** Branch instructions perform conditional jumps based on certain conditions. They evaluate specific flags or conditions in the CPU's status register to determine whether to transfer control to a different memory address or instruction. Examples of branch instructions include **BEQ** (Branch if Equal), **BNE** (Branch if Not Equal), **BGT** (Branch if Greater Than), etc.

3. **Subroutine Call (CALL):**
    - The CALL instruction is used to transfer control to a subroutine or function.
    - It saves the return address (usually the address of the next instruction after the CALL) onto the stack and transfers control to the specified subroutine.
    - After the subroutine completes, it returns control to the instruction following the CALL instruction.
4. **Return (RET):** This is used to return control from a subroutine back to the calling routine. It retrieves the return address from stack and transfers control to that address.
5. **Interrupts and Exceptions:**
    - Many instruction sets provide instructions to handle interrupts and exceptions.
    - These instructions allow the CPU to respond to external events or exceptional conditions during program execution.
    - When an interrupt or exception occurs, the CPU suspends the current execution, saves the necessary context, and transfers control to a predefined interrupt handler or exception handler.
6. **Conditional Execution:**
    - Some instruction sets support conditional execution, where the execution of an instruction depends on a specified condition.
    - For example, an instruction may only be executed if a certain flag is set or if a specific condition is met.

INSTRUCTION SET FORMAT: FIXED, VARIABLE AND HYBRID

1. **Fixed Instruction Set Format:**

In a fixed instruction set format, all instructions have the same length. Each instruction is divided into fixed-size fields, and the meaning of each field is predefined. This format simplifies instruction fetching and decoding.

Example:

Consider a simple fixed instruction set format with 16-bit instructions, where each instruction has two fields: a 4-bit opcode and a 12-bit operand field. The opcode field specifies the operation to be performed, and the operand field contains the data or memory address involved in the operation.

Example Instructions:

- **ADD R1, 5:** Opcode (4 bits) = 0001, Operand (12 bits) = 000000000101
- **SUB R2, 10:** Opcode (4 bits) = 0010, Operand (12 bits) = 000000001010

2. **Variable Instruction Set Format:**

   In a variable instruction set format, instructions can have varying lengths. The length of an instruction depends on factors such as the opcode, number and type of operands, and other instruction-specific considerations.

   Example:

   Let's consider a variable instruction set format where instructions have different lengths. The instruction length can vary based on the opcode and the addressing mode.

   Example Instructions:

   - **ADD R1, R2:** Opcode (4 bits) = 0001, Operand1 (4 bits) = R1, Operand2 (4 bits) = R2
   - **LOAD R3, [100]:**
     Opcode (4 bits) = 0010, Operand1 (4 bits) = R3, Operand2 (12 bits) = 100

3. **Hybrid Instruction Set Format:**

   A hybrid instruction set format combines fixed-length and variable-length instructions. It provides a balance between the two approaches, ensuring some fields have fixed positions and lengths, while others can vary.

   Example:

   Let's consider a hybrid instruction set format with 32-bit instructions. The format reserves the first 8 bits for the opcode and uses the remaining bits for various fields, such as registers, immediate values, and memory addresses.

   Example Instructions:

   - **ADD R1, R2:**

     Opcode (8 bits) = 00000001, Register1 (4 bits) = R1, Register2 (4 bits) = R2
   - **LOAD R3, [2000]:**

     Opcode (8 bits) = 00000010, Register (4 bits) = R3,

     Memory Address (20 bits) = 00000000000011111000

* FLOATING - POINT NUMBERS :-

→ It allows a much larger range of values of a no. but require either costly processing HW or lengthy slow implementation.

→ The floating point representation, has three parts :

(i) Mantissa (M)
(ii) Exponent (E)
(iii) Base (B)

(i) Mantissa (M) :
→ represents a signed, fixed pt. no.
→ may be fraction or integer.
→ also has significand or fraction.

(ii) Exponent (E) :
→ It designates the position of the decimal (or binary) point.
→ It is an integer.

(iii) Base (B) :
→ It is usually 2 or 10.

→ floating point of a no. can be represented as;

$$M \times B^E$$

Eg. $1.0 \times 10^{(18)}$ → Exponent
↑ ↑ Base
Mantissa

FLOATING POINT NUMBERS :

eg.1 Decimal Nos. $(+6132.789)_{10}$

If we move (.) in left, exponent is +ve
If we move (.) in right, exponent is -ve.

say, $6132.789 \Rightarrow 6.132789 \times 10^3$

$\therefore M = (6.132789)_{10}$
$B = 10$; Bcoz of decimal no.
$E = 3$

eg.2 Binary Nos. $(+1001.11)_2$

say, $1001.11 \Rightarrow (.100111) \times 2^4 \; 10^4$

$\therefore M = (.100111)_2$
$B = 2$; Bcoz of binary no.
$E = 4$

- For negative exponent, eg. $(+123.567)_{10}$

$123.567 \Rightarrow 12356.7 \times 10^{-2}$

reverse: $12356.7 \times 10^{-2} \Rightarrow 123.567$

FLOATING-POINT ADDITION ALGORITHM:

1. Align the exponents by adjusting the significands and exponents.
2. Add the significands based on the sign of the numbers.
3. Normalize the result by adjusting the exponent and significand if necessary.
4. Apply rounding rules to obtain the final result.

Example: Let's add two floating-point numbers: **1.234 x 10^3 and 5.678 x 10^2**.

Step 1: Align the exponents,

We need to adjust the second number to have the same exponent as the first number.

1.234 x 10^3 + 0.5678 x 10^3

Step 2: Add the significands 1.234 + 0.5678 = 1.8018

Step 3: Normalize the result by adjusting the exponent and significand. 1.8018 x 10^3

Step 4: Apply rounding rules to obtain the final result.

For above example, rounding to two decimal places: 1.80 x 10^3

Therefore, the sum of 1.234 x 10^3 and 5.678 x 10^2 is approximately 1.80 x 10^3.

FLOATING-POINT SUBTRACTION ALGORITHM:

1. Align the exponents by adjusting the significands and exponents.
2. Subtract the significands.
3. Normalize the result by adjusting the exponent and significand if necessary.
4. Apply rounding rules to obtain the final result.

Example: Let's subtract two floating-point numbers: 2.345 x 10^4 and 1.234 x 10^3.

Step 1: Align the exponents,

We need to adjust the second number to have the same exponent as the first number.

2.345 x 10^4 - 0.1234 x 10^4

Step 2: Subtract the significands. 2.3450 - 0.1234 = 2.2216

Step 3: Normalize the result by adjusting the exponent and significand. 2.2216 x 10^4

Step 4: Apply rounding rules to obtain the final result.

For above example, rounding to three decimal places: 2.222 x 10^4

Therefore,

The difference between 2.345 x 10^4 and 1.234 x 10^3 is approximately 2.222 x 10^4.

FLOATING-POINT MULTIPLICATION ALGORITHM:

1. Multiply the significands.
2. Add the exponents.
3. Normalize the result by adjusting the exponent and significand if necessary.
4. Apply rounding rules to obtain the final result.

Example: Let's multiply two floating-point numbers: 2.345 x 10^4 and 1.234 x 10^3.

Step 1: Multiply the significands 2.345 x 1.234 = 2.88863

Step 2: Add the exponents. 4 + 3 = 7

Step 3: Normalize the result by adjusting the exponent and significand. 2.88863 x 10^7

Step 4: Apply rounding rules to obtain the final result.

For above example, rounding to four decimal places: 2.8886 x 10^7

Therefore, the product of 2.345 x 10^4 and 1.234 x 10^3 is approximately 2.8886 x 10^7.

FLOATING-POINT DIVISION ALGORITHM:

1. Divide the significands.
2. Subtract the exponents.
3. Normalize the result by adjusting the exponent and significand if necessary.
4. Apply rounding rules to obtain the final result.

Example: Let's divide 5.76 x 10^2 by 0.3 x 10^2.

Step 1: Divide the significands 5.76 and 0.3. 5.76 / 0.3 = 19.2

Step 2: Subtract the exponents

Since both numbers have the same exponent of 2, there is no need to subtract the exponents.

Step 3: Normalize the result by adjusting the exponent and significand if necessary.

Step 4: Apply rounding rules to obtain the final result.

For above example, rounding to two decimal places: 19.20

Therefore, the division of 5.76 x 10^2 by 0.3 x 10^2 is approximately 19.20.