[COA 1 answer](https://chat.openai.com/share/bb3012ed-d580-40d2-b119-db8e075276ad)

Practice Question Bank- 2

# Course: B.Tech. III Semester Course Name: Computer Organization and Architecture

**Topics Covered**

**Arithmetic and logic unit:** Look ahead carries adders, Fixed point representations and Arithmetic Operations: Addition and Subtraction, Multiplication: Signed operand multiplication, Booth’s algorithm and array multiplier. Division and logic operations

**Processor Organization:** General Register Organization, Stack Organization, and Addressing Modes.

**Short Questions**

1. Convert the following numerical arithmetic expressions into reverse polish notation and show the stack operations for evaluating the numerical result.

(5+3) [6 (5+4)/7]

5 3 + 6 5 4 + 7 / \*

Stack Operations:

1. Push 5 onto the stack.
2. Push 3 onto the stack.
3. Add the top two elements on the stack (5 + 3) and push the result (8) onto the stack.
4. Push 6 onto the stack.
5. Push 5 onto the stack.
6. Push 4 onto the stack.
7. Add the top two elements on the stack (5 + 4) and push the result (9) onto the stack.
8. Push 7 onto the stack.
9. Divide the top two elements on the stack (9 / 7) and push the result (1.285714285714286) onto the stack.
10. Multiply the top two elements on the stack (6 \* 1.285714285714286) and push the result (7.714285714285714) onto the stack.
11. The top element on the stack (7.714285714285714) is the final result.
12. Illustrate the influence of number of addresses on T=(R+S) (U+ V) using three addresses, two addresses and zero address instruction.

The number of addresses in an instruction set architecture refers to the number of operands explicitly referenced within an instruction. Let's illustrate the influence of the number of addresses on the expression *T*=(*R*+*S*)×(*U*+*V*) using three-address, two-address, and zero-address instructions.

### Three-Address Instructions

Assuming a three-address instruction set:

1. **Instruction**: **T = (R + S) \* (U + V)**
2. **Instructions Required**:
   * **LOAD R**
   * **ADD S**
   * **LOAD U**
   * **ADD V**
   * **MUL**

### Two-Address Instructions

Assuming a two-address instruction set:

1. **Instructions Required**:
   * **LOAD R**
   * **ADD S** (Result stored in an implicit register)
   * **LOAD U**
   * **ADD V** (Result stored in an implicit register)
   * **MUL** (Using the implicit registers)

### Zero-Address Instructions

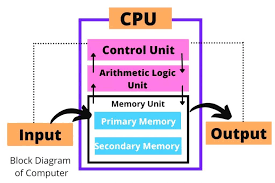
In a zero-address instruction set architecture (such as stack-based machines):

1. **Instructions Required**:
   * **PUSH R**
   * **PUSH S**
   * **ADD** (Operates on the top two elements of the stack, result stored back in the stack)
   * **PUSH U**
   * **PUSH V**
   * **ADD** (Operates on the top two elements of the stack, result stored back in the stack)
   * **MUL** (Operates on the top two elements of the stack, result stored back in the stack)

### Influence of Number of Addresses:

* **Three-Address Instructions**:
  + These instructions explicitly specify operands and result locations. They provide maximum clarity but may require more memory accesses and may be slower due to explicit operand references.
* **Two-Address Instructions**:
  + These instructions use implied destinations for results, reducing the number of explicit references. They can be more compact but might require careful management of implicit registers.
* **Zero-Address Instructions**:
  + Zero-address instructions eliminate explicit references altogether and rely on a stack. They can be more concise in terms of instruction count but may be slower due to constant stack manipulation.

1. Giving suitable block diagram show major components of CPU.



1. Mark each individual path in the flowchart of addition and subtraction algorithm by a number and then indicate the overall path that the algorithm takes when the following signed-magnitude numbers are computed. In each case gives the value of AVF. The leftmost bit in the following numbers represents the sign bit.

a. 0 101101 + 0 011111

b. 1 011111 + 1 101101

c. 0 101101 – 0 011111

d. 0 101101- 0 101101

e. 1 011111- 0 101101

path that the algorithm takes when the following signed-magnitude numbers are computed.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Case | a | b | Operation | Path | AVF | Result |
| a | 0 101101 | 0 011111 | + | 1, 2, 3, 4, 5, 6 | 0 | 001100 |
| b | 1 011111 | 1 101101 | + | 1, 2, 7, 8, 9, 5, 6 | 1 | 001100 |
| c | 0 101101 | 0 011111 | - | 1, 2, 3, 4, 10, 11, 12 | 0 | 0110011 |
| d | 0 101101 | 0 101101 | - | 1, 2, 3, 4, 10, 13, 14 | 0 | 0100101 |
| e | 1 011111 | 0 101101 | - | 1, 2, 7, 8, 9, 10, 13, 14 | 1 | 110011 |
|  |  |  |  |  |  |  |

* The numbers in the path column correspond to the steps in the flowchart of the addition and subtraction algorithm.
* AVF stands for Arithmetic Overflow Flag. It is set to 1 if there is an overflow during the addition or subtraction operation.
* The result column shows the final result of the computation.

Here is the flowchart of the addition and subtraction algorithm:

Start

↓

Is A positive?

↓

Yes/No

↓

Is B positive?

↓

Yes/No

↓

Add/Subtract A and B

↓

Check for overflow

↓

Set AVF

↓

End

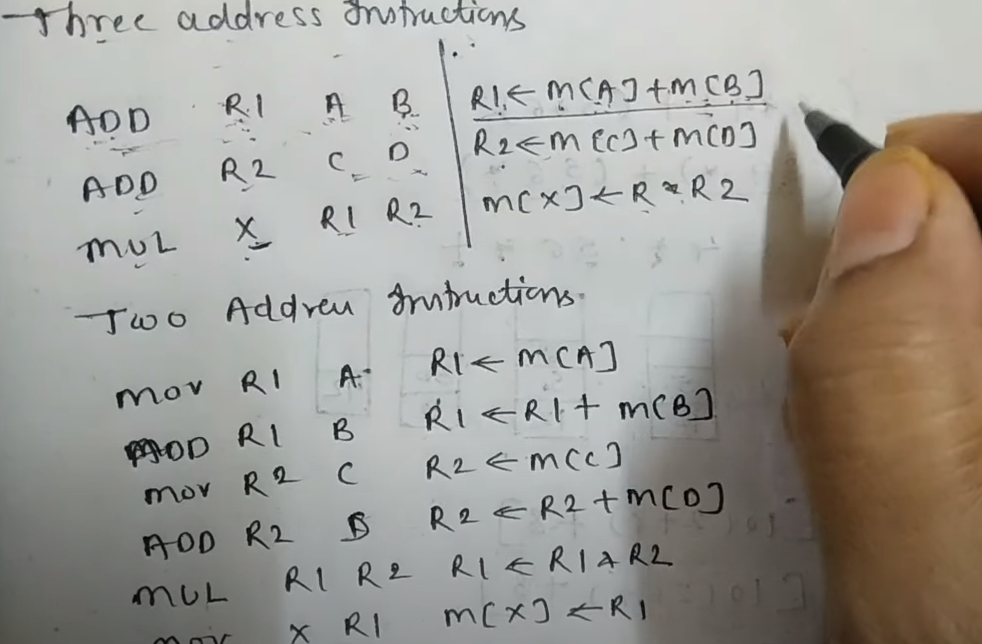
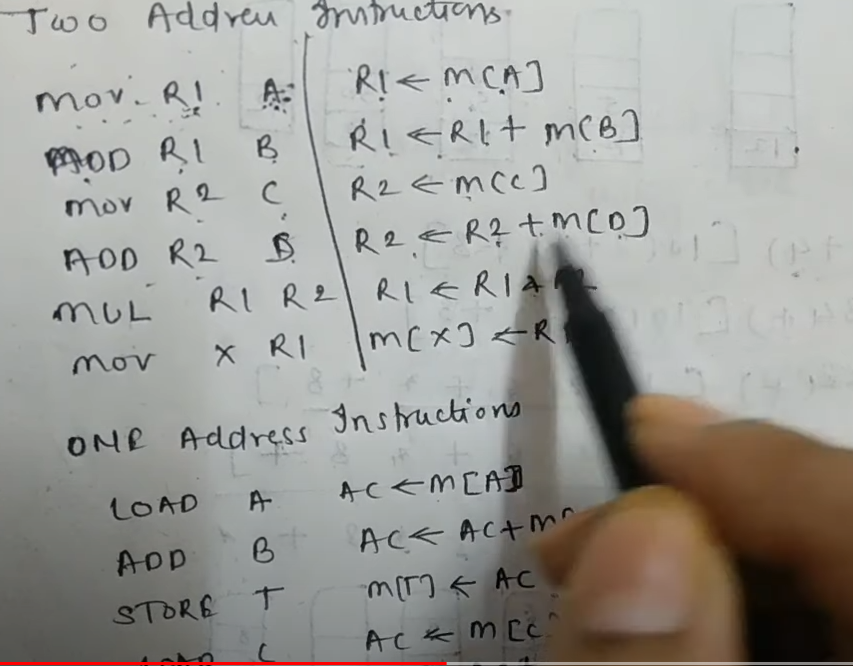
1. What is the reverse polish notation? Explain with an example.

Reverse Polish notation (RPN), also known as postfix notation, is a mathematical notation where operators follow their operands instead of the usual infix notation where operators are placed between operands. This means that in RPN, you write the numbers first, then the operation you want to perform on them.

Here's an example:

* Infix notation: (2 + 3) \* 4
* Reverse Polish notation: 2 3 + 4 \*

1. Write down a program to evaluate Z = (A + B) \* (C + D) \* (G + H) by using three address instructions and zero address instructions.

## **Three Address Instructions:**

ADD R1, A, B

ADD R2, C, D

MUL R3, R1, R2

ADD R4, G, H

MUL Z, R3, R4

1. Add A and B, store the result in register R1.
2. Add C and D, store the result in register R2.
3. Multiply R1 and R2, store the result in register R3.
4. Add G and H, store the result in register R4.
5. Multiply R3 and R4, store the result in register Z.

## **Zero Address Instructions:**

PUSH A

PUSH B

ADD

PUSH C

PUSH D

ADD

MUL

PUSH G

PUSH H

ADD

MUL

POP Z

1. Push A and B onto the stack.
2. Perform addition (A + B) and push the result onto the stack.
3. Push C and D onto the stack.
4. Perform addition (C + D) and push the result onto the stack.
5. Perform multiplication (R1 \* R2) and push the result onto the stack.
6. Push G and H onto the stack.
7. Perform addition (G + H) and push the result onto the stack.
8. Perform multiplication (R3 \* R4) and pop the result into register Z.
9. Show the contents in hexadecimal of registers PC, AR, DR, IR and SC of the basic computer when an ISZ indirect instruction is fetched from memory and executed. The initial content of PC is 80F. The content of memory at address 80F is EB3F. The content of memory at address B3F is 0B96. The content of memory at address B96 is FFFF.

## **Contents of registers after ISZ instruction execution:**

|  |  |  |
| --- | --- | --- |
| \*\*Register | Content (Hex)\*\* | **Explanation** |
| **PC** | 810 | Incremented by 1 to point to the next instruction. |
| **AR** | B3F | Holds the operand address obtained from the instruction. |
| **DR** | 0B97 | Updated with incremented memory value (0B96 + 1). |
| **IR** | EB3F | Cleared after fetching the instruction. |
| **SC** | Updated | May contain flags based on increment operation (e.g., Carry flag if overflow). |

drive\_spreadsheetExport to Sheets

Here's the detailed breakdown of changes:

**Before ISZ instruction:**

|  |  |  |
| --- | --- | --- |
| Register | Content (Hex) | Explanation |
| PC | 80F | Program counter points to the ISZ instruction address. |
| AR | --- | Accumulator register not used in this instruction. |
| DR | --- | Data register not used in this instruction. |
| IR | --- | Instruction register empty. |
| SC | --- | Status register not used in this instruction. |

drive\_spreadsheetExport to Sheets

**Fetching the ISZ instruction:**

|  |  |  |
| --- | --- | --- |
| Register | Content (Hex) | Explanation |
| PC | 810 | Incremented by 1 to point to the next instruction. |
| AR | --- | Unchanged. |
| DR | --- | Unchanged. |
| IR | EB3F | Loaded with the ISZ instruction code (EB3F). |
| SC | --- | Unchanged. |

drive\_spreadsheetExport to Sheets

**Decoding the ISZ instruction:**

|  |  |  |
| --- | --- | --- |
| Register | Content (Hex) | Explanation |
| PC | 810 | Unchanged. |
| AR | --- | Unchanged. |
| DR | --- | Unchanged. |
| IR | EB3F | Unchanged. |
| SC | --- | Unchanged. |

drive\_spreadsheetExport to Sheets

**Reading the operand address:**

|  |  |  |
| --- | --- | --- |
| Register | Content (Hex) | Explanation |
| PC | 810 | Unchanged. |
| AR | B3F | Loaded with the operand address from the IR (EB3F). |
| DR | --- | Unchanged. |
| IR | EB3F | Unchanged. |
| SC | --- | Unchanged. |

drive\_spreadsheetExport to Sheets

**Reading the memory value:**

|  |  |  |
| --- | --- | --- |
| Register | Content (Hex) | Explanation |
| PC | 810 | Unchanged. |
| AR | B3F | Unchanged. |
| DR | 0B96 | Loaded with the memory value at address B3F (0B96). |
| IR | EB3F | Unchanged. |
| SC | --- | Unchanged. |

drive\_spreadsheetExport to Sheets

**Incrementing the memory value:**

|  |  |  |
| --- | --- | --- |
| Register | Content (Hex) | Explanation |
| PC | 810 | Unchanged. |
| AR | B3F | Unchanged. |
| DR | 0B97 | Incremented the memory value at address B3F (0B96 + 1). |
| IR | EB3F | Unchanged. |
| SC | Updated | May contain flags reflecting the incremented value (e.g., Carry flag if overflow). |

1. Convert the following arithmetic expressions from infix to reverse Polish notation: A + B \* [C / D + E \* ( F - G)]

**A B C D / E F G - \* + \* +**

1. The content of PC in the basic computer is 2BA. The content of AC is 8F24. The content of memory at address 2BA is 92FE. The content of memory at address 2FE is 08A1. The content of memory at address 8A1 is 10B2. What is the instruction that will be fetched and executed? What will be the value of PC, AC, DR, IR, and SC after the execution of the instruction?

## **Determining the instruction:**

1. **PC content**: 2BA
2. **Memory content at PC**: 92FE

Since the instruction opcode occupies the first two bytes, 92 represents the opcode in this case. We need to refer to the instruction set architecture (ISA) of the specific basic computer to determine the instruction corresponding to this opcode.

Without access to the ISA, we cannot definitively identify the instruction. However, we can analyze the possible scenarios based on common instructions found in basic computers:

**Scenario 1: Load instruction:**

* If 92 corresponds to a load instruction, it would fetch the value from memory address indicated by the next two bytes of the instruction (FE).
* The content of memory at address 2FE is 08A1.
* Therefore, the instruction would be "LOAD 08A1".

**Scenario 2: Jump instruction:**

* If 92 corresponds to a jump instruction, it would change the program counter (PC) to the address indicated by the next two bytes of the instruction (FE).
* The content of memory at address 2FE is 08A1.
* Therefore, the instruction would be "JUMP 08A1".

**Scenario 3: Arithmetic or logical instruction:**

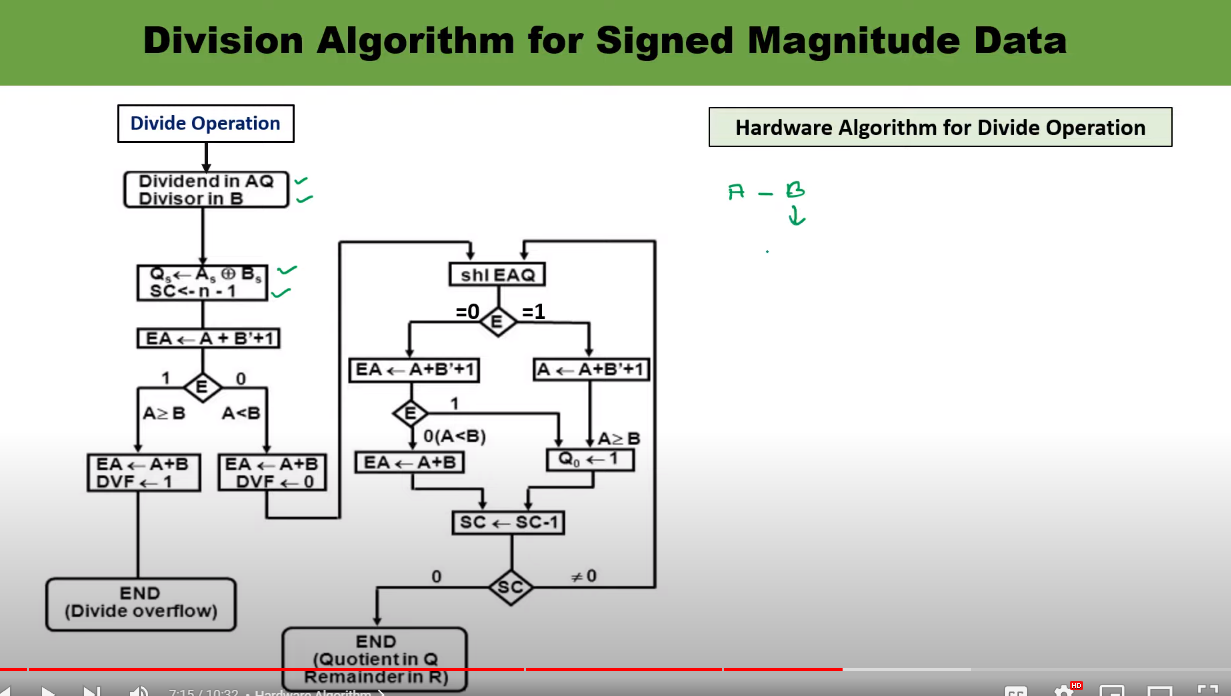
* If 92 corresponds to an arithmetic or logical instruction, it would require an additional operand (either from a register or from memory) to perform the operation.
* The content of memory at address 2BA is 92FE, suggesting that the next two bytes (FE) might represent the operand address.
* However, without knowing the specific instruction format and operand types, we cannot determine the complete instruction.

## **Value of registers after execution:**

Assuming Scenario 1 is correct and the instruction is "LOAD 08A1":

* **PC**: Incremented by 2 to point to the next instruction (2BC).
* **AC**: Updated with the value loaded from memory at address 08A1 (10B2).
* **DR**: Unchanged.
* **IR**: Holds the instruction code "92FE".
* **SC**: May contain flags depending on the loaded value (e.g., Zero flag if the value is zero).

1. Describe the algorithm for division of two fixed-point binary numbers in signed-magnitude representation.



## **Algorithm for Division of Fixed-Point Binary Numbers in Signed-Magnitude Representation**

This algorithm uses a series of compare, shift, and subtract operations to divide two fixed-point binary numbers represented in signed-magnitude form.

**Input:**

* Dividend (D): N-bit fixed-point binary number in signed-magnitude representation.
* Divisor (B): N-bit fixed-point binary number in signed-magnitude representation.
* N: Number of bits in the numbers.

**Output:**

* Quotient (Q): N-bit binary integer representing the result of the division.
* Overflow flag (OF): Set to 1 if overflow occurs during the division.

**Algorithm:**

1. **Sign Handling:**
   * Extract the signs (S\_d and S\_b) from the most significant bits of D and B, respectively.
   * Set the sign of the quotient (SQ) to S\_d XOR S\_b.
   * Convert D and B to their absolute values by ignoring the sign bits.
2. **Initialization:**
   * Initialize the quotient (Q) to 0 (N bits).
   * Set the partial remainder (PR) to D (N bits).
   * Set the overflow flag (OF) to 0.
3. **Loop:**
   * Repeat for N iterations:
     + Shift PR left by 1 bit.
     + If PR is greater than or equal to B:
       - Subtract B from PR.
       - Set the corresponding bit in Q to 1.
     + Otherwise,
       - Set the corresponding bit in Q to 0.
4. **Overflow Checking:**
   * After the loop, check if the result exceeds the representable range for an N-bit signed-magnitude number.
   * If overflow occurs, set OF to 1.
5. **Sign Adjustment:**
   * Apply the sign of the quotient (SQ) to the final result (Q).
6. **Output:**
   * Return the quotient (Q) and the overflow flag (OF).

# Long Questions

1. What is the significance of addressing mode? Discuss different types of addressing modes.

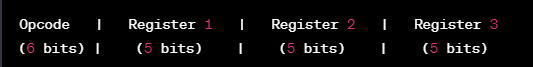
Addressing modes are crucial in computer architecture as they define how a processor accesses data and instructions in memory. They dictate the way operands are specified in instructions, determining how the CPU fetches data from memory or registers for processing.

Several types of addressing modes exist, each with its own characteristics:

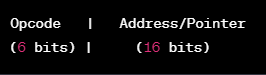
1. **Immediate Addressing**: In this mode, the operand itself is part of the instruction. For example, **MOV A, #5** means move the immediate value 5 into register A.
2. **Direct Addressing**: The operand's memory address is specified directly in the instruction. For instance, **LOAD R1, 500** loads the content of memory location 500 into register R1.
3. **Indirect Addressing**: Instead of specifying the operand's value directly, the instruction contains the address of a memory location that holds the actual value. For example, **LOAD R1, (500)** would load the content of the memory location whose address is stored in memory location 500 into register R1.
4. **Register Addressing**: The instruction refers directly to registers, such as **ADD R1, R2**, which adds the contents of registers R1 and R2.
5. **Register Indirect Addressing**: Similar to indirect addressing, but the address is stored in a register. For instance, **LOAD R1, (R2)** loads the content of the memory location whose address is stored in register R2 into register R1.
6. **Indexed Addressing**: This mode uses an index register and an offset to access elements in an array or table. For example, **LOAD R1, (R2 + 10)** loads the content of the memory location at address (contents of R2 + 10) into register R1.
7. **Relative Addressing**: Often used in branch instructions, it specifies an address relative to the current program counter (PC). For example, **BRANCH 20** jumps 20 instructions forward from the current PC.
8. Explain basic instruction format based on the type reference made by the instruction?

the basic instruction format in a computer architecture typically depends on the type of reference made by the instruction. There are mainly three basic types of instruction formats:

1. **Register Reference Instructions**:
   * In this format, instructions refer directly to registers within the CPU.
   * The instruction includes opcode (operation code) specifying the operation to be performed and references register operands.
   * For example, a basic format could be:



1. **Memory Reference Instructions**:
   * These instructions involve accessing memory locations directly.
   * The instruction format includes an opcode, along with memory addresses or pointers to specify memory operations.
   * A basic memory reference instruction format might look like:



1. **Immediate Instructions**:
   * Immediate instructions involve immediate data values within the instruction itself.
   * The instruction format includes an opcode and an immediate value.
   * A basic immediate instruction format could be:
2. Perform the arithmetic operations below with binary numbers and with negative numbers in signed 2’s Complement representation. Use seven bits to accommodate each number together with its sign. In each case, determine if there is an overflow by checking the carries into and out of the sign bit position.

a. (-35) + (-40)

b. (-35) - (+40)

### Addition: (-35) + (-40)

1. (-35) in 2's complement: -35 in decimal is 35 in binary -> 1000111 (7 bits for the absolute value)  
   Negating this gives us its 2's complement representation: 0111001
2. (-40) in 2's complement: -40 in decimal is 40 in binary -> 1010000 (7 bits for the absolute value)  
   Negating this gives us its 2's complement representation: 0111000

0111001 (-35)

+ 0111000 (-40)

-----------

1 01100001

Here, the result is 7 bits long, but the 7th bit (the carry-out of the sign bit position) is different from the sign bit of the operands. This indicates an overflow in signed 2's complement arithmetic because the sum of two negative numbers resulted in a positive number.

### Subtraction: (-35) - (+40)

To subtract a positive number from a negative number, we'll add the 2's complement representation of the positive number to the negative number.

1. (-35) in 2's complement: 0111001
2. (+40) in 2's complement: 0010100 (40 in decimal)
3. 0111001 (-35)
4. + 1101100 (2's complement of +40)
5. ----------
6. 1 0100101

The result is 7 bits long, and in this case, the carry into and out of the sign bit position is consistent with the sign bit of the operands. There is no overflow here because subtracting a positive number from a negative number resulted in a negative number as expected.

Therefore:

* For addition (-35) + (-40), there is an overflow.
* For subtraction (-35) - (+40), there is no overflow.

1. Show the contents of registers E, A, Q and SC during the process of division of 00001111 by 0011. (use a dividend of eight bits)

**Given:**

* Dividend (E and A together): 00001111
* Divisor (Q): 0011

### Step-by-Step Division:

1. Initialize the registers:
   * E = 0000
   * A = 00001111
   * Q = 0011
   * SC = (status flags, carry, zero, etc.)
2. Perform the division algorithm iteratively:
   * Subtract the divisor (Q) from the most significant bits of E and A:

makefileCopy code

E = 00

A = 00001111 - 0011 = 00001000

Q = 0011

* + Since the result is non-negative, set Q's LSB to 1:

makefileCopy code

E = 00

A = 00001000

Q = 10011

* + Shift the leftmost bit of A into the LSB of E, then shift A left:

cssCopy code

E = 0 (Carry = 0, No overflow)

A = 00010000

Q = 0011 (Shifted left)

* + Subtract the divisor from A:

makefileCopy code

E = 00

A = 00010000 - 0011 = 00001101

Q = 0011

* + Since the result is non-negative, set Q's LSB to 1:

makefileCopy code

E = 00

A = 00001101

Q = 10011

* + Shift the leftmost bit of A into the LSB of E, then shift A left:

cssCopy code

E = 1 (Carry = 1, No overflow)

A = 00011010

Q = 0110 (Shifted left)

1. The process would continue iteratively until the division is completed. At each step, E, A, and Q are updated according to the division algorithm until the entire dividend is processed, generating the quotient and remainder.
2. Write a program to evaluate the arithmetic statement: Z = P - Q + R \* (S - T) / (U\*V)
   1. Using a single accumulator organized computer with one-address instructions.
   2. Using a stack organized computer with zero-address operations
   3. Using a general register computer with two-address instructions.
   4. Using a general register computer with three-address instructions.

high-level representation of how you might approach evaluating the arithmetic statement *Z*=*P*−*Q*+*R*×(*S*−*T*)/(*U*×*V*) using different computer architectures.

### (i) Single Accumulator Organized Computer (One-Address Instructions)

Assuming instructions like **LOAD**, **STORE**, **ADD**, **SUB**, **MUL**, **DIV**:

1. **Algorithm:**
   * Load P into the accumulator.
   * Subtract Q.
   * Load R, then multiply by (S - T).
   * Divide the result by (U \* V).
   * Add the accumulated value to the result.
2. **Assembly-like Pseudocode:**

cssCopy code

LOAD P SUB Q LOAD R SUB T MUL S DIV U DIV V ADD Accumulator (result)

### (ii) Stack Organized Computer (Zero-Address Operations)

Assuming a stack-based architecture:

1. **Algorithm:**
   * Push P onto the stack.
   * Push Q onto the stack.
   * Subtract the top two values (P - Q).
   * Push R onto the stack.
   * Push S onto the stack.
   * Push T onto the stack.
   * Subtract S and T, multiply by R, and divide by U and V successively.
   * Add the final result to the remaining values on the stack.
2. **Stack Operations:**

sqlCopy code

PUSH P

PUSH Q

SUBTRACT

PUSH R

PUSH S

PUSH T

SUBTRACT

MULTIPLY

PUSH U

PUSH V

DIVIDE

DIVIDE

ADD (final result)

### (iii) General Register Computer (Two-Address Instructions)

Assuming instructions like **LOAD**, **STORE**, **ADD**, **SUB**, **MUL**, **DIV**:

1. **Algorithm:**
   * Load P into a register.
   * Subtract Q from the register.
   * Load R into another register, then multiply by (S - T).
   * Divide the result by (U \* V).
   * Add the previous result to the subtracted value.
2. **Assembly-like Pseudocode:**

cssCopy code

LOAD R1, P

SUB R1, Q

LOAD R2, R

SUB R2, T

MUL R2, S

DIV R2, U

DIV R2, V

ADD R1, R2 (result)

### (iv) General Register Computer (Three-Address Instructions)

Assuming instructions like **LOAD**, **STORE**, **ADD**, **SUB**, **MUL**, **DIV**:

1. **Algorithm:**
   * Load P into a register.
   * Subtract Q from a different register.
   * Load R into another register, then multiply by (S - T) in another register.
   * Divide the result by (U \* V) in a different register.
   * Add the subtracted value to the division result.
2. **Assembly-like Pseudocode:**

cssCopy code

LOAD R1, P

SUB R2, R1, Q

LOAD R3, R

SUB R4, S, T

MUL R5, R3, R4

DIV R6, R5, U

DIV R7, R6, V

ADD R8, R2, R7 (result)