Phase One - Additional ALU Instructions and Justification

As part of the ALU instruction set, we have selected the following five additional instructions to complement the existing nine. These instructions provide expanded functionality for logic operations, bit manipulation, and efficient processing.

1. XOR (Exclusive OR)

- Mnemonic: XOR
- Operation: A ⊕ B
- Justification: The XOR operation is widely used in cryptographic applications, error detection, and parity checking. It is also fundamental in constructing adders and other arithmetic circuits.

2. NAND (Not AND)

- Mnemonic: NAND
- Operation: ~(A & B)
- Justification: The NAND operation is crucial in hardware logic design because it is functionally complete, meaning all other logic operations can be built from NAND gates. Additionally, NAND is a fundamental building block in memory circuits and digital logic simplification.

3. NOR (Not OR)

- Mnemonic: NOR
- Operation: ~(A | B)
- Justification: Like NAND, NOR is functionally complete and plays a significant role in digital logic circuits. It is commonly used in processor design, memory storage, and control logic operations.

4. Clear Register

- Mnemonic: CLR
- Operation: A = 0
- Justification: The ability to clear a register is essential for resetting values, optimizing performance, and preventing unintended carry-over errors. This instruction is particularly useful in initialization and error handling scenarios.

5. Rotate Left (Circular Shift Left)

- Mnemonic: ROL
- Operation: Circular left shift (bit wrap-around)

- Justification: Unlike logical shifts, which introduce zeros, rotation allows bits to cycle through, maintaining information. This is beneficial for cryptographic algorithms, efficient multiplication, and data scrambling techniques.

These five instructions enhance the ALU's capabilities by improving logical operations, register management, and performance in specialized applications. Together with the nine previously defined instructions, they form a robust instruction set, reserving two additional instructions for branching operations in future phases.

Phase 2

As part of the ALU instruction set, we have defined a total of 16 operations, including the initial nine, five additional operations, and two reserved for branching. The following table outlines the complete opcode set:

Opcode (Binary)	Mnemonic	Operation	Description	
0000	ADD	A + B	Arithmetic addition	
0001	INC	A + 1	Increment	
0010	DEC	A - 1	Decrement	
0011	CMP	Compare A, B	Comparison with three outputs (equal, less than, greater than)	
0100	NOT	~A	Logic bitwise NOT	
0101	AND	A & B	Logic bitwise AND	
0110	OR	A	B	Logic bitwise OR
0111	SHR	A >> 1	Register right logic shift	
1000	SHL	A << 1	Register left logic shift	
1001	XOR	A ⊕ B	Exclusive OR	
1010	NAND	~(A & B)	Not AND	
1011	NOR	~(A	B)	Not OR
1100	CLR	A = 0	Clear register	
1101	ROL	Circular left shift	Rotate bits left	
1110	BRANCH1	Branching Instruction 1	Reserved for future use	
1111	BRANCH2	Branching Instruction 2	Reserved for future use	

These instructions enhance the ALU's capabilities by improving logical operations, register management, and performance in specialized applications. The additional two reserved branching instructions will be defined in future phases.

Phase three is a diagram that I am attaching in a .txt
file called full -8 bit ALU

Phase 4

Assembly Programs

Program 1: Add Operands Until Zero

```
LOAD R1, VALUE   ; Load initial value into R1
LOAD R2, SUM     ; Load accumulator (SUM) into R2
LOOP:
   CMP R1, 0    ; Compare R1 with zero
   BEQ END      ; If R1 == 0, exit loop
   ADD R2, R1   ; Add R1 to SUM
   LOAD R1, NEXT_VALUE ; Load next value into R1 (assume input update happens externally)
   JMP LOOP     ; Repeat the loop
END:
   STORE R2, SUM ; Store final sum result
```

Program 2: Shift Until LSB is Zero

```
LOAD R1, VALUE   ; Load initial value into R1
LOOP:
   AND R2, R1, 1 ; Check if LSB is 1
   BEQ END      ; If LSB is 0, exit loop
   CMP R1, 255  ; Check if R1 is 11111111
   BEQ END      ; If all bits are 1, exit loop
   SHR R1       ; Shift right by one bit
   JMP LOOP     ; Repeat the loop
END:
   STORE R1, RESULT ; Store final shifted value
```