ALU Module Documentation

Introduction

The ALU (Arithmetic Logic Unit) module is the computational core of a 16-bit processor implementation. It performs all arithmetic and logical operations for the processor, taking two 16-bit inputs and producing a 16-bit result based on a control signal. The ALU maintains status flags (zero, overflow, negative) that are crucial for conditional operations and program flow control in a processor architecture.

This implementation simulates the behavior of a RISC-V 16-bit processor's ALU, supporting basic arithmetic operations, bitwise logical operations, and comparison operations. It includes debugging features like operation history tracking and status display methods.

Class Methods and Functions

`__init__(self)`

Purpose: Initializes the ALU with default state values

How it works:

- Sets `last_result` to 0
- Initializes all flags (zero, overflow, negative) to False
- Prepares operation counters and history tracking lists
- This is automatically called when creating a new ALU instance

Example:

```
alu = ALU()  # Creates a fresh ALU instance
```

`execute(input_a, input_b, alu_control)`

Purpose: Main method to perform ALU operations

How it works:

1. Ensures inputs are 16-bit by masking with 0xFFFF
2. Uses the alu_control code to select which operation to perform
3. Calls the appropriate internal operation method
4. Updates flags based on the result
5. Records the operation in history
6. Returns the 16-bit result

Parameters:

- `input_a`, `input_b`: 16-bit input values
- `alu_control`: 4-bit operation code (use class constants like ALU.ALU_ADD)

Returns: 16-bit operation result

Example:

```
result = alu.execute(10, 20, ALU.ALU_ADD)  # Returns 30 (0x001E)
```

`_add(a, b)`

Purpose: Performs 16-bit addition

How it works:

- Adds the two inputs
- Checks for overflow (result > 16 bits)
- Returns result masked to 16 bits
- Sets overflow_flag if overflow occurs

Internal method - called by execute()

`_sub(a, b)`

Purpose: Performs 16-bit subtraction

How it works:

- Subtracts b from a
- Handles negative results using two's complement
- Returns result masked to 16 bits
- Doesn't explicitly set overflow flag (unlike addition)

Internal method - called by execute()

`_and(a, b)`

Purpose: Bitwise AND operation

How it works:

- Performs bitwise AND between inputs
- Returns result masked to 16 bits

Internal method - called by execute()

`_or(a, b)`

Purpose: Bitwise OR operation

How it works:

- Performs bitwise OR between inputs
- Returns result masked to 16 bits

Internal method - called by execute()

`_xor(a, b)`

Purpose: Bitwise XOR operation

How it works:

- Performs bitwise XOR between inputs
- Returns result masked to 16 bits

Internal method - called by execute()

`_compare_eq(a, b)`

Purpose: Equality comparison

How it works:

- Compares the two inputs
- Returns 1 if equal, 0 if not
- Used for conditional branching

Internal method - called by execute()

`_compare_ne(a, b)`

Purpose: Inequality comparison

How it works:

- Compares the two inputs
- Returns 1 if not equal, 0 if equal
- Used for conditional branching

Internal method - called by execute()

`_update_flags(result)`

Purpose: Updates status flags after an operation

How it works:

- Sets zero_flag if result is zero
- Sets negative_flag if MSB is 1 (result is negative in signed interpretation)
- Note: overflow_flag is set by individual operations when needed

Internal method - called by execute()

`_get_operation_name(alu_control)`

Purpose: Converts control codes to human-readable names

How it works:

- Uses a dictionary mapping of control codes to names
- Returns "UNKNOWN" for invalid codes
- Used for operation history display

Internal method - called by execute()

`get_flags()`

Purpose: Returns current flag states

How it works:

- Returns a dictionary with current values of:
    - zero_flag
    - overflow_flag
    - negative_flag

Example:

- `flags = alu.get_flags()`
- `if flags['zero']:`

```
print("Last result was zero")
```

`display_status()`

Purpose: Shows current ALU state in formatted output

How it works:

- Prints a formatted box with:
    - Last result in hex and decimal
    - Current state of all flags
    - Total operations count
- Uses ASCII art for visual appeal

Example Output:

- ┌──┐
- │ ALU STATUS │
- ├──┤
- │ Last Result: 0x001E (30) │
- │ Zero Flag: ✗ │
- │ Overflow: ✗ │
- │ Negative: ✗ │
- │ Operations: 3 │

```
└────────────────────────────────────────┘
```

`display_history(last_n=5)`

Purpose: Shows recent operations

How it works:

- Prints the last 'n' operations (default 5)
- Shows operation type, inputs, and result in hexadecimal
- Uses ASCII art for visual appeal

Parameters:

- `last_n`: Number of recent operations to display

Example Output:

- ┌──┐
- │ ALU HISTORY │
- ├──┤
- │ 1. ADD 0x000A, 0x0014 → 0x001E │
- │ 2. SUB 0x0064, 0x0032 → 0x0032 │
- │ 3. AND 0x00FF, 0x000F → 0x000F │

```
└────────────────────────────────────────────────┘
```

`reset()`

Purpose: Resets ALU to initial state

How it works:

- Clears last result
- Resets all flags to False
- Clears operation count and history
- Useful for processor reset scenarios

Example:

```
alu.reset()  # Returns ALU to fresh state
```

Main Function

The module includes a `main()` function that demonstrates basic ALU functionality when run directly, showing:

1. ALU creation
2. Simple operations
3. Status display
4. History display

This serves as both example usage and simple test case.