Control Unit Module Documentation

Introduction

The Control Unit is the "brain" of the processor that coordinates all other components. It takes decoded instructions and generates the appropriate control signals to execute each instruction properly. This implementation models a RISC-V 16-bit processor's control unit with support for multiple instruction types (R, I, S, B, J) and special operations.

Key responsibilities:

- Generates control signals for ALU operations
- Manages register file read/write operations
- Controls memory access (load/store)
- Handles program flow (branches/jumps)
- Maintains execution state and statistics

Class Structure

`ControlUnit`

Main class that implements the processor's control logic through a comprehensive control signal lookup table and signal generation system.

Detailed Method Documentation

`ControlUnit` Methods

`__init__(self)`

Purpose: Initializes the control unit with default state

Initializes:

- `signal_count`: Counter for generated signals
- `signal_history`: List of recent control signals
- `control_table`: Comprehensive lookup table mapping instructions to control signals

The Control Table defines these signals for each instruction:

- `alu_operation`: ALU operation code (from ALU module)
- `alu_src_a`: ALU input A source ("register" or None)
- `alu_src_b`: ALU input B source ("register", "immediate" or None)
- `reg_write_enable`: Whether to write to register file
- `reg_write_src`: Register write source ("alu", "memory", "pc_plus_1")
- `mem_read`: Memory read enable
- `mem_write`: Memory write enable
- `branch`: Branch instruction flag
- `jump`: Jump instruction flag
- `pc_update`: PC update mode ("increment", "conditional", "jump", "halt")

`generate_control_signals(decoded_instruction: Dict[str, Any]) -> Dict[str, Any]`

Purpose: Generates control signals for a decoded instruction

Process:

1. Checks instruction validity
2. Looks up base signals in control table
3. Adds instruction-specific information
4. Logs the signal generation

5. Returns complete control signals dictionary

Parameters:

- `decoded_instruction`: Dictionary containing decoded instruction fields

Returns: Dictionary of control signals including:

- All signals from control table
- Original instruction information
- Validity flags

Example:

```
decoded = {"instruction_name": "ADD", "rd": 1, "rs1": 2, "rs2": 3, "valid":
True}
signals = control_unit.generate_control_signals(decoded)
```

`_generate_error_signals(decoded_instruction: Dict[str, Any]) -> Dict[str, Any]`

Purpose: Generates safe control signals for invalid instructions

Behavior:

- Disables all potentially dangerous operations
- Allows PC to increment
- Marks signals as invalid/error

`_get_memory_operation(signals: Dict[str, Any]) -> str`

Purpose: Helper to get memory operation description

Returns: "READ", "WRITE", or "NONE" based on control signals

Instruction Type Check Methods

`is_branch_instruction(control_signals: Dict[str, Any]) -> bool`

Purpose: Checks if instruction is a branch

Uses: `branch` flag from control signals

`is_jump_instruction(control_signals: Dict[str, Any]) -> bool`

Purpose: Checks if instruction is a jump

Uses: `jump` flag from control signals

`is_memory_instruction(control_signals: Dict[str, Any]) -> bool`

Purpose: Checks if instruction accesses memory

Uses: `mem_read` or `mem_write` flags

`should_halt(control_signals: Dict[str, Any]) -> bool`

Purpose: Checks for HALT instruction

Uses: `pc_update` == "halt"

Statistics and Display Methods

`get_statistics() -> Dict[str, Any]`

Purpose: Returns control unit statistics

Returns: Dictionary with:

- `total_signals_generated`: Count of all signals generated
- `history_size`: Current history entries

`display_control_table()`

Purpose: Prints formatted control table grouped by instruction type

Output: Shows instruction types with their control signals

`display_signal_history(last_n=5)`

Purpose: Shows recent control signal history

Parameters:

- `last_n`: Number of recent entries to display

Demo Function

`demo_control_unit()`

Purpose: Demonstrates control unit functionality

Shows:

1. Complete control table
2. Test cases with various instruction types
3. Signal generation results
4. Signal history
5. Statistics

Key Concepts

Instruction Types Supported:

1. R-Type (Register operations):
   - ADD, SUB, AND, OR, XOR
   - Uses two source registers and writes to destination
   - Example: `ADD x1, x2, x3`
2. I-Type (Immediate operations):
   - ADDI, ANDI, ORI
   - Uses register and immediate value
   - Example: `ADDI x1, x2, 5`
3. Load/Store:
   - LW (load word from memory)

    - SW (store word to memory)
    - Example: `LW x1, 4(x2)`
4. B-Type (Branches):
    - BEQ, BNE (branch if equal/not equal)
    - Example: `BEQ x1, x2, label`
5. J-Type (Jumps):
    - JAL (jump and link)
    - Example: `JAL x1, label`
6. Special:
    - NOP (no operation)
    - HALT (stop execution)

Control Signal Flow:

1. Instruction gets decoded
2. Control unit looks up signals for that instruction
3. Signals get distributed to:
    - ALU (operation and inputs)
    - Register File (read/write)
    - Memory (read/write)
    - PC (next address logic)
4. Components execute based on signals
5. Results are collected and stored

Example Usage

```
# Create control unit

control = ControlUnit()
```

```python
# Example decoded instruction (from InstructionDecoder)

decoded_add = {

    "instruction_name": "ADD",

    "rd": 1,

    "rs1": 2,

    "rs2": 3,

    "valid": True

}

# Generate control signals

signals = control.generate_control_signals(decoded_add)

# Check if this is a memory operation

if control.is_memory_instruction(signals):

    print("This instruction accesses memory")

# Display control table

control.display_control_table()

# Show recent history
control.display_signal_history(3)
```

This control unit implementation provides complete control signal generation for a 16-bit RISC-V processor, with extensive debugging and visualization capabilities to help understand and verify processor operation.