Instruction Decoder Module Documentation

Introduction

The Instruction Decoder module is responsible for interpreting 16-bit binary machine code instructions into structured data that the processor can execute. It supports the full RISC-V 16-bit instruction set architecture (ISA) including R-type, I-type, S-type, B-type, J-type, and special instructions.

Key features:

- Comprehensive instruction set coverage
- Detailed instruction decoding with assembly representation
- Support for signed immediates and offsets
- Instruction validation and error handling
- Decoding statistics and history tracking

Class Structure

`InstructionDecoder`

Main class that implements the instruction decoding logic with complete ISA support.

Detailed Method Documentation

`InstructionDecoder` Methods

`__init__(self)`

Purpose: Initializes the decoder with instruction set definitions

Initializes:

- `isa_table`: Complete instruction set architecture lookup table
- `decode_count`: Total instructions decoded counter
- `decode_history`: List of recently decoded instructions

The ISA Table contains:

- Opcode mapping (0x0-0xF)
- Instruction names (e.g., "ADD", "LW")
- Instruction types (R, I, S, B, J, Special)
- Human-readable descriptions

`decode(instruction: int) -> Dict[str, Any]`

Purpose: Main decoding function for 16-bit instructions

Process:

1. Extracts 4-bit opcode
2. Looks up instruction in ISA table
3. Routes to appropriate type decoder
4. Returns structured instruction data

Parameters:

- `instruction`: 16-bit machine code instruction

Returns: Dictionary containing:

- Instruction type and name
- Decoded fields (registers, immediates)
- Raw instruction value
- Assembly language representation
- Validity flag

Example:

```
decoder = InstructionDecoder()
decoded = decoder.decode(0x510A)   # ADDI x1, x0, 10
```

Type-Specific Decoders

`_decode_r_type(instruction, opcode, inst_info)`

Decodes: Register-to-register operations

Format: `[4-bit opcode][4-bit rd][4-bit rs1][4-bit rs2]`

Handles: ADD, SUB, AND, OR, XOR

Example: `ADD x3, x1, x2` → `0x0312`

`_decode_i_type(instruction, opcode, inst_info)`

Decodes: Immediate operations and loads

Format: `[4-bit opcode][4-bit rd][4-bit rs1][4-bit imm]`

Handles: ADDI, ANDI, ORI, LW

Example: `LW x4, 0(x2)` → `0x8420`

`_decode_s_type(instruction, opcode, inst_info)`

Decodes: Store operations

Format: `[4-bit opcode][4-bit rs2][4-bit rs1][4-bit offset]`

Handles: SW

Example: `SW x3, 0(x2)` → `0x9320`

`_decode_b_type(instruction, opcode, inst_info)`

Decodes: Conditional branches

Format: `[4-bit opcode][4-bit rs1][4-bit rs2][4-bit offset]`

Handles: BEQ, BNE

Example: `BEQ x3, x4, 1` → `0xA341`

`_decode_j_type(instruction, opcode, inst_info)`

Decodes: Jump and link

Format: `[4-bit opcode][4-bit rd][8-bit offset]`

Handles: JAL

Example: `JAL x1, 16` → `0xC110`

`_decode_special_type(instruction, opcode, inst_info)`

Decodes: Special operations

Handles: NOP, HALT

Example: `HALT` → `0xF000`

Utility Methods

`_create_invalid_instruction(instruction, opcode)`

Purpose: Creates error structure for unknown instructions

Behavior: Marks instruction as invalid with descriptive message

`get_statistics()`

Returns: Decoding statistics dictionary

Contains:

- `total_decoded`: Count of all decoded instructions
- `history_size`: Current history entries

Display Methods

`display_instruction_set()`

Purpose: Prints formatted list of supported instructions

Output: Groups instructions by type with opcodes and descriptions

`display_decode_history(last_n=5)`

Purpose: Shows recently decoded instructions

Parameters:

- `last_n`: Number of recent entries to display

  Output: Shows raw, type, name, and assembly for each instruction

Demo Function

`demo_instruction_decoder()`

Purpose: Demonstrates decoder functionality

Shows:

1. Supported instruction set
2. Test cases with various instruction types
3. Decoding results
4. Decode history
5. Statistics

Key Concepts

Instruction Formats:

1. R-Type (Register operations):
   - Fields: opcode, rd, rs1, rs2
   - Example: `ADD x3, x1, x2`
2. I-Type (Immediate operations):
   - Fields: opcode, rd, rs1, immediate
   - Example: `ADDI x1, x2, 5`
3. S-Type (Store operations):
   - Fields: opcode, rs2, rs1, offset

    - Example: `SW x3, 4(x2)`
4. B-Type (Branch operations):
    - Fields: opcode, rs1, rs2, offset
    - Example: `BEQ x1, x2, label`
5. J-Type (Jump operations):
    - Fields: opcode, rd, offset
    - Example: `JAL x1, label`
6. Special:
    - No operands
    - Example: `HALT`

Signed Immediate Handling:

- 4-bit immediates: Values 0-7 positive, 8-15 interpreted as -8 to -1
- 8-bit offsets: Values 0-127 positive, 128-255 interpreted as -128 to -1

Example Usage

```
# Create decoder

decoder = InstructionDecoder()



# Decode instructions

add_inst = decoder.decode(0x0312)   # ADD x3, x1, x2

lw_inst = decoder.decode(0x8420)    # LW x4, 0(x2)



# Display information

print(add_inst["assembly"])   # "add x3, x1, x2"

print(lw_inst["immediate"])   # 0
```

```
# View supported instructions
decoder.display_instruction_set()


# Check decoding history
decoder.display_decode_history(3)
```

Testing

The module includes a test function (`demo_instruction_decoder()`) that demonstrates:

1. Instruction set display
2. Decoding various instruction types
3. Error handling for invalid instructions
4. History tracking

Run tests with:

```
if __name__ == "__main__":
    demo_instruction_decoder()
```

This instruction decoder provides complete decoding of the 16-bit RISC-V instruction set with detailed error handling and debugging capabilities, serving as a crucial component in the processor pipeline.