RISC-V Assembler Module Documentation

Introduction

The RISC-V Assembler module is a two-pass assembler that converts RISC-V assembly code (16-bit variant) into executable machine code. It handles the complete assembly process including label resolution, instruction encoding, and output generation in both binary and human-readable formats.

Key features:

- Supports a subset of RISC-V instructions (16-bit variant)
- Handles labels and symbolic addresses
- Performs two-pass assembly (label resolution then code generation)
- Generates both binary (.bin) and hexadecimal (.hex) output files
- Includes helpful debugging and visualization tools

Class Structure

`RiscVAssembler`

Main assembler class that handles the assembly process from source to machine code.

`BinaryLoader`

Utility class for loading and inspecting binary files created by the assembler.

Detailed Method Documentation

`RiscVAssembler` Methods

`__init__(self)`

Purpose: Initializes the assembler with empty state

Initializes:

- `labels`: Dictionary for label → address mapping
- `instructions`: List of parsed instructions
- `machine_code`: List of generated machine code
- `current_line`: Current line counter
- `opcodes`: Mapping of mnemonics to 4-bit opcodes
- `register_map`: Mapping of register names to numbers

Example:

```
assembler = RiscVAssembler()
```

`assemble_file(filename: str) -> List[int]`

Purpose: Main entry point for assembling a file

Process:

1. Reads the input .asm file
2. Calls `_assemble_lines` to process the content
3. Returns generated machine code

Parameters:

- `filename`: Path to assembly source file

Returns: List of 16-bit machine code instructions

Example:

```python
machine_code = assembler.assemble_file("program.asm")
```

`_assemble_lines(lines: List[str]) -> List[int]`

Purpose: Core two-pass assembly process

Process:

1. First pass (`_find_labels`): Builds label address map
2. Second pass (`_convert_to_machine_code`): Generates machine code

Parameters:

- `lines`: List of assembly source lines

Returns: List of 16-bit machine code instructions

`_find_labels(lines: List[str])`

Purpose: First pass - identifies all labels and their addresses

Process:

- Scans each line for labels (text ending with ':')
- Records label addresses in `self.labels`
- Skips comments and empty lines

Example:

```
loop:           # Label at address 0

    ADD x1, x2, x3
    BEQ x1, x0, loop   # Will resolve to address 0
```

`_convert_to_machine_code(lines: List[str])`

Purpose: Second pass - converts assembly to machine code

Process:

- Processes each instruction line

- Calls `_parse_instruction` for each valid instruction
- Handles labels, comments, and directives
- Builds `self.machine_code` list

`_parse_instruction(line: str, address: int) -> Optional[int]`

Purpose: Parses a single assembly instruction

Process:

1. Splits instruction into components
2. Validates opcode
3. Routes to appropriate encoder based on instruction type
4. Returns 16-bit machine code or None if invalid

Parameters:

- `line`: Assembly instruction line
- `address`: Current instruction address (for PC-relative addressing)

Returns: 16-bit machine code or None

Handles these instruction types:

- R-type: ADD, SUB, AND, OR, XOR
- I-type: ADDI, ANDI, ORI, LW
- S-type: SW
- B-type: BEQ, BNE
- J-type: JAL
- Special: NOP, HALT

Register and Operand Parsing Methods

`_parse_register(reg_str: str) -> int`

Purpose: Converts register name to number

Example:

```
reg_num = self._parse_register("x1")  # Returns 1
reg_num = self._parse_register("sp")  # Returns 2
```

`_parse_immediate(imm_str: str) -> int`

Purpose: Parses immediate values (4-bit)

Supports: Decimal (10) and Hex (0xA) formats

`_parse_memory_operand(operand: str) -> Tuple[int, int]`

Purpose: Parses memory operands like "4(x2)"

Returns: (offset, register) tuple

`_parse_branch_target(target: str, current_addr: int) -> int`

Purpose: Resolves branch targets (labels or offsets)

Handles: PC-relative addressing for labels

Instruction Encoding Methods

`_encode_r_type(opcode, rd, rs1, rs2) -> int`

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

`_encode_i_type(opcode, rd, rs1, imm) -> int`

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

`_encode_s_type(opcode, rs2, rs1, imm) -> int`

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

`_encode_b_type(opcode, rs1, rs2, offset) -> int`

Format: `[4-bit opcode][4-bit rs1][4-bit rs2][4-bit offset]` `_encode_j_type(opcode, rd, offset) -> int`

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

Output Methods

`save_binary_file(filename: str) -> bool`

Purpose: Saves machine code as raw binary

Format: Little-endian 16-bit words

`save_hex_file(filename: str) -> bool`

Purpose: Saves human-readable hex dump

Format: One instruction per line with address, hex, and binary

`display_summary()`

Purpose: Shows assembly summary including labels and generated code

`BinaryLoader` Methods

`load_binary_file(filename: str) -> List[int]`

Purpose: Loads binary file into list of instructions

Returns: List of 16-bit instructions

`display_binary_content(instructions: List[int])`

Purpose: Displays binary content in human-readable format

Main Function

The module includes a `main()` function that provides command-line interface:

```
Usage: python RiscV_Assembler.py <input.asm> [output_prefix]
```

Example Workflow

1. Write assembly code (program.asm)
2. Assemble it:

   ```
   assembler = RiscVAssembler()machine_code =
   assembler.assemble_file("program.asm")
   ```

3. Save outputs:

   ```
   assembler.save_binary_file("program.bin")

   assembler.save_hex_file("program.hex")
   ```

4. Load and inspect:

   ```
   loader = BinaryLoader()

   code = loader.load_binary_file("program.bin")
   ```

5. `loader.display_binary_content(code)`

This assembler provides a complete toolchain for developing programs for a 16-bit RISC-V processor, from assembly source to executable binary.