RISC-V Processor Module Documentation

Introduction

The RISC-V Processor module is the complete implementation of a 16-bit RISC-V CPU that integrates all components (ALU, register file, memory, control unit, etc.) into a functional processor. It implements a Harvard architecture with separate instruction and data memory spaces, and supports the full RISC-V instruction set including arithmetic, memory access, control flow, and special operations.

Key features:

- Complete 5-stage pipeline (fetch, decode, execute, memory, writeback)
- Support for R/I/S/B/J-type instructions
- Memory-mapped I/O with protected address spaces
- Comprehensive execution statistics and debugging
- Interactive step-by-step execution
- Program loading from assembly or binary files

Class Structure

`RiscVProcessor`

Main class implementing the complete processor with all components integrated.

Detailed Method Documentation

`RiscVProcessor` Methods

`__init__(instruction_memory_size=1024, data_memory_size=1024)`

Purpose: Initializes the complete processor

Components initialized:

- Register file (16 registers)
- ALU with 7 operations
- Instruction and data memory
- Instruction decoder
- Control unit
- Program counter and execution state

Parameters:

- `instruction_memory_size`: Size of instruction memory in words
- `data_memory_size`: Size of data memory in words

Program Loading Methods

`load_program_from_file(filename)`

Purpose: Loads program from .asm or .bin file

Handles:

- Assembly files (.asm): Assembles then loads
- Binary files (.bin): Directly loads

`_load_from_assembly(filename)`

Internal: Assembles and loads .asm file using RiscVAssembler

`_load_from_binary(filename)`

Internal: Loads raw binary file

`load_program_direct(instructions)`

Purpose: Loads program from list of instructions

Execution Control Methods

`step()`

Purpose: Executes one instruction cycle

Process:

1. Instruction fetch
2. Instruction decode
3. Control signal generation
4. Instruction execution
5. PC update

`run(max_cycles=1000)`

Purpose: Runs program until HALT or max cycles

Parameters:

- `max_cycles`: Safety limit to prevent infinite loops

`reset()`

Purpose: Resets processor to initial state

Resets:

- Program counter
- All registers
- Memory contents
- Statistics

Instruction Execution Methods

`_execute_r_type(decoded, control_signals)`

Executes: ADD, SUB, AND, OR, XOR

Process:

1. Reads source registers
2. Performs ALU operation
3. Writes result to destination register

`_execute_i_type(decoded, control_signals)`

Executes: ADDI, ANDI, ORI

Handles: Signed/unsigned immediate values

`_execute_load(decoded, control_signals)`

Executes: LW

Process:

1. Calculates memory address
2. Reads from data memory
3. Writes to destination register

`_execute_store(decoded, control_signals)`

Executes: SW

Process:

1. Calculates memory address
2. Writes register value to memory

`_execute_branch(decoded, control_signals)`

Executes: BEQ, BNE

Process:

1. Compares registers

2. Updates PC if branch taken

`_execute_jump(decoded, control_signals)`

Executes: JAL

Process:

1. Saves return address
2. Jumps to target

`_execute_special(decoded, control_signals)`

Executes: NOP, HALT

Utility Methods

`_update_pc(control_signals, decoded)`

Purpose: Updates program counter based on instruction type

`_update_statistics(decoded, control_signals)`

Purpose: Updates execution statistics counters

`_log_execution(decoded, control_signals)`

Purpose: Maintains execution history for debugging

Display Methods

`display_status()`

Purpose: Shows current processor state

Displays:

- Register values
- ALU status

- Non-zero memory contents

`display_statistics()`

Purpose: Shows execution statistics

Includes:

- Instruction mix
- Branch statistics
- Memory operations
- Cycles per instruction (CPI)

`display_execution_trace(last_n=10)`

Purpose: Shows recent execution history

Demo Function

`demo_processor()`

Purpose: Demonstrates complete processor functionality

Shows:

1. Processor initialization
2. Program loading
3. Step-by-step execution
4. Full program run
5. Final state and statistics

Key Concepts

Memory Architecture:

- Harvard architecture: Separate instruction and data memory
- Instruction memory: Read-only, contains program

- Data memory: Read/write, starts at 0x1000

Instruction Execution Pipeline:

1. Fetch: Read instruction from memory
2. Decode: Determine operation and operands
3. Execute: Perform ALU operation
4. Memory: Access data memory if needed
5. Writeback: Update registers

Supported Instructions:

```
# R-Type (Register)
ADD  x1, x2, x3   # x1 = x2 + x3
SUB  x1, x2, x3   # x1 = x2 - x3

# I-Type (Immediate)
ADDI x1, x2, 5    # x1 = x2 + 5
LW   x1, 4(x2)    # x1 = memory[x2 + 4]

# S-Type (Store)
SW   x1, 4(x2)    # memory[x2 + 4] = x1

# B-Type (Branch)
BEQ  x1, x2, label  # if x1 == x2, jump to label

# J-Type (Jump)
JAL  x1, label     # jump to label, store return in x1
```

```
# Special
NOP                 # no operation
HALT                # stop execution
```

Example Usage

```python
# Create processor
processor = RiscVProcessor()

# Load program
processor.load_program_from_file("program.bin")

# Run program
processor.run()

# Inspect results
processor.display_status()
processor.display_statistics()

# Step through execution
processor.reset()
for _ in range(5):
    processor.step()
    processor.display_status()
```

Testing

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

1. Processor initialization
2. Direct program loading
3. Step-by-step execution
4. Full program execution
5. Result inspection

Run tests with:

```python
if __name__ == "__main__":
    demo_processor()
```

This processor implementation provides a complete RISC-V 16-bit CPU simulation with comprehensive debugging and visualization capabilities, suitable for both educational purposes and hardware modeling.