RISC-V System Architecture Guide

Introduction

This document provides a comprehensive overview of the RISC-V 16-bit processor simulator system architecture, detailing how all components interact to create a functional processor simulation.

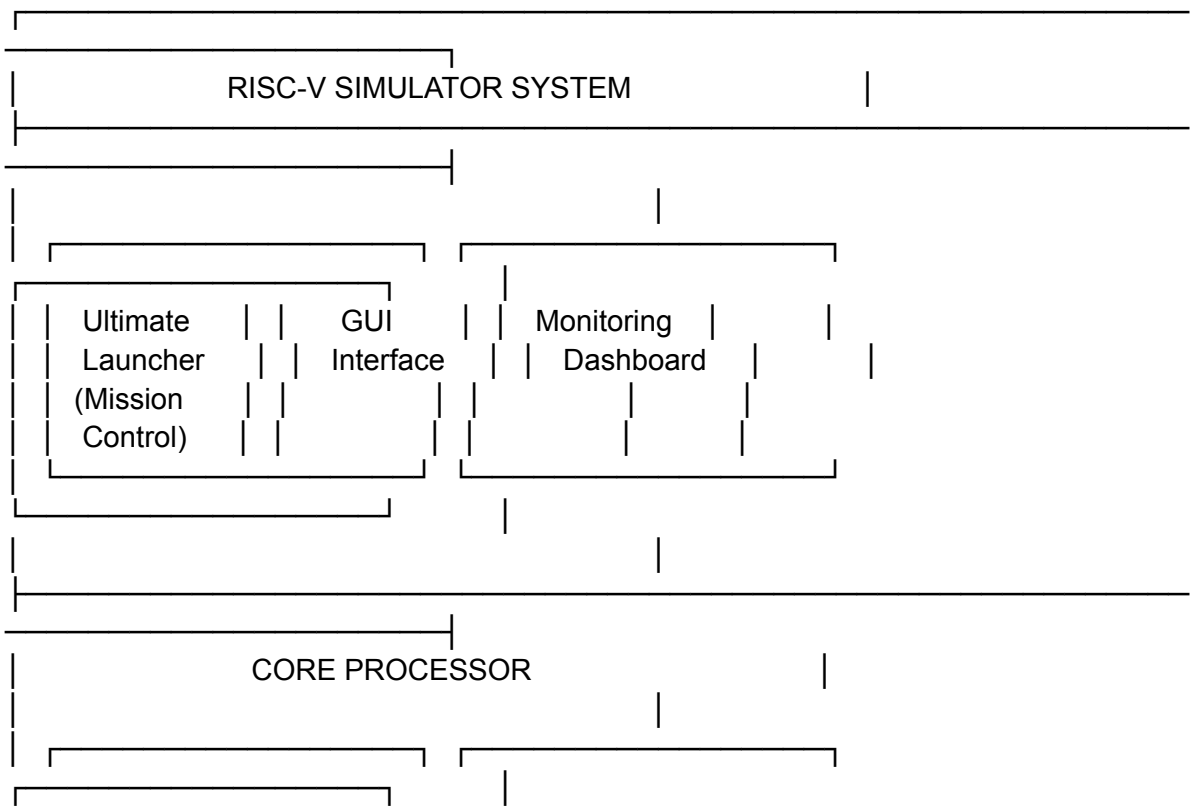
System Overview

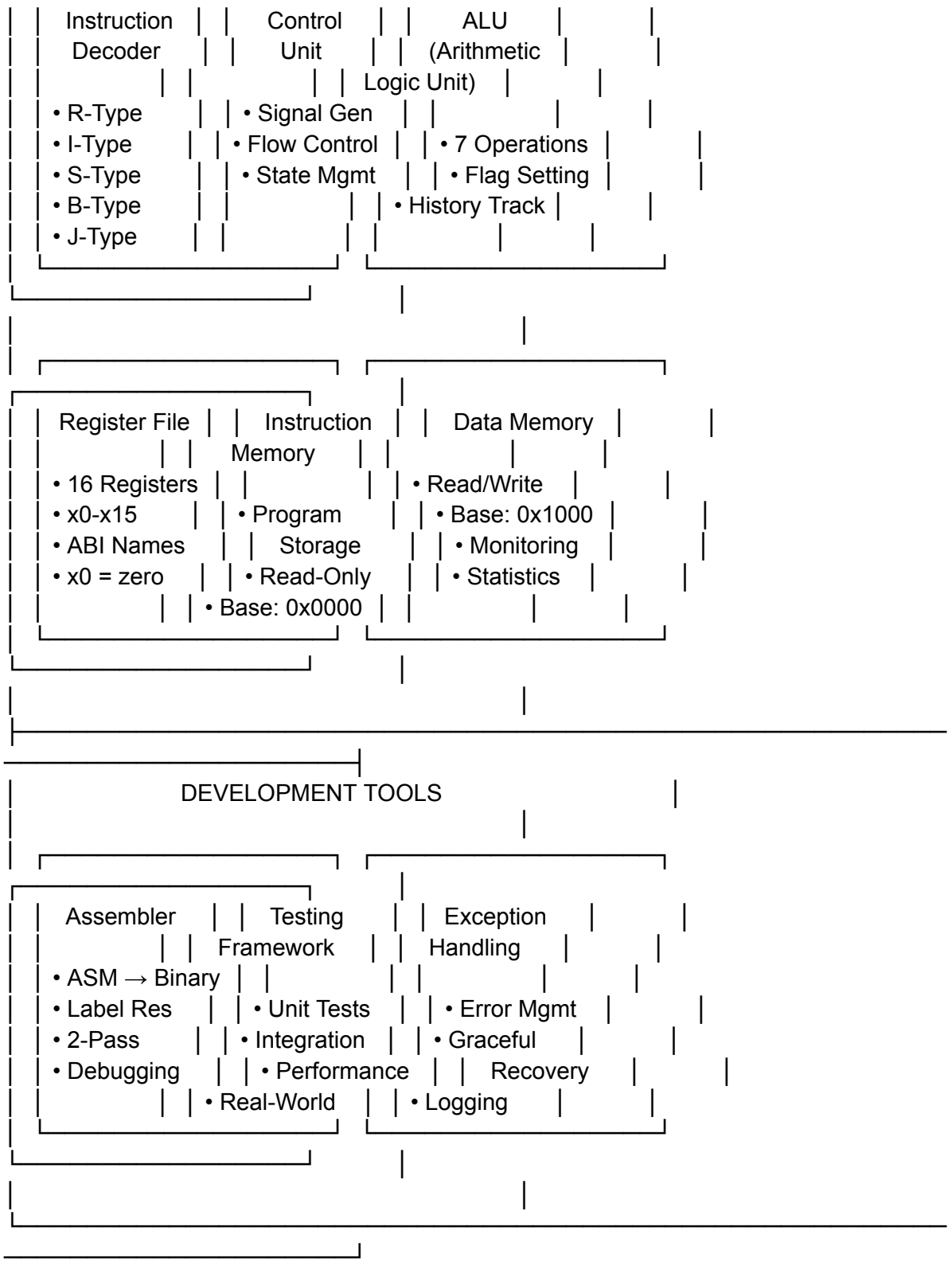
The RISC-V simulator implements a **Harvard architecture** with separate instruction and data memory spaces, supporting a complete RISC-V instruction set in a 16-bit implementation.

Key Architectural Principles

- **Modularity**: Each component is independent and testable
- **Separation of Concerns**: Clear boundaries between functional units
- **Educational Focus**: Designed for learning and understanding
- **Extensibility**: Easy to add new features and instructions

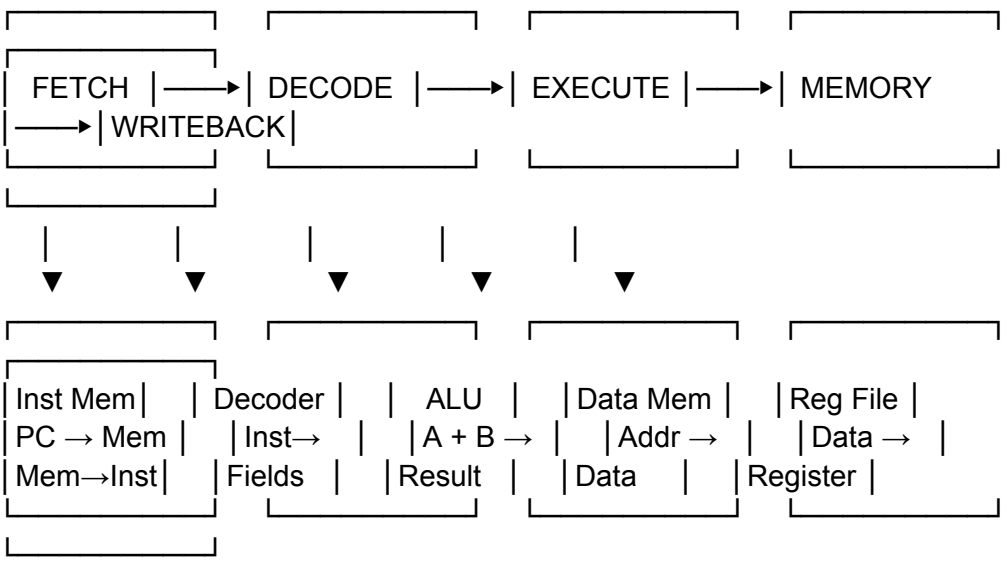
High-Level Architecture Diagram



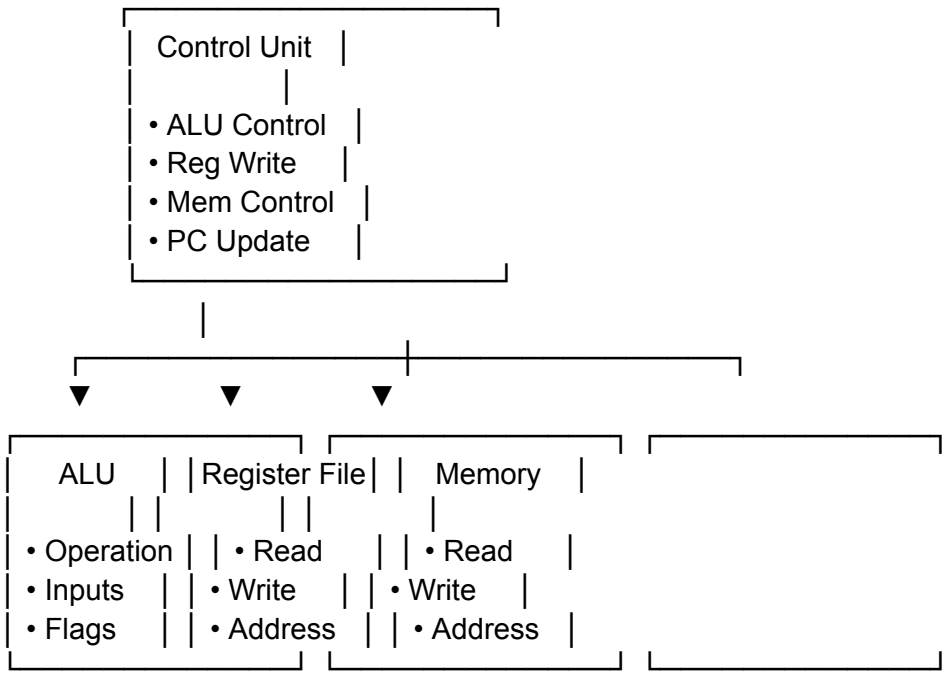


Data Flow Architecture

1. Instruction Execution Pipeline



2. Control Signal Flow



Component Interactions

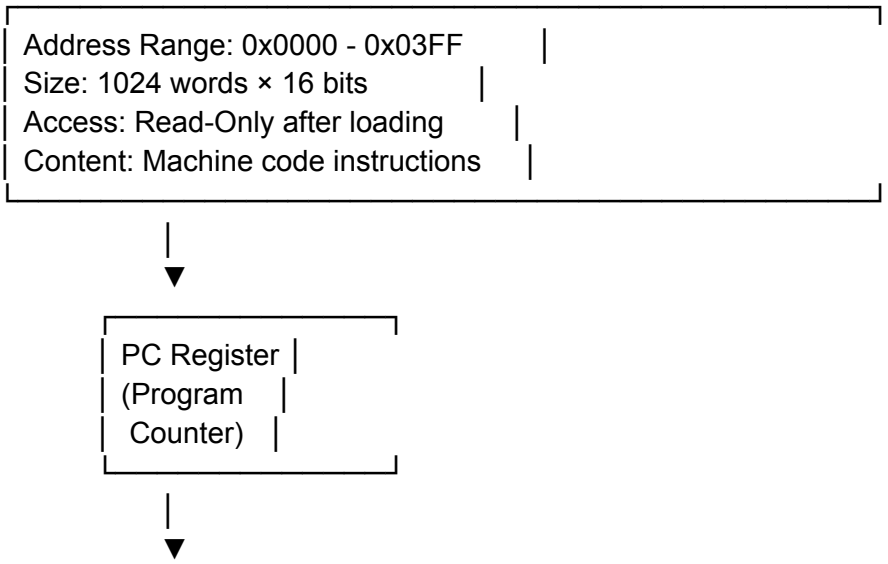
Core Processing Loop

```
while not halted:
    # 1. FETCH
    instruction = instruction_memory.read(pc)

    # 2. DECODE
```

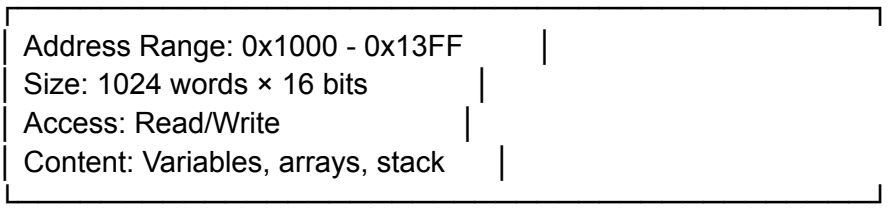
```
    decoded = instruction_decoder.decode(instruction)

    # 3. CONTROL
    signals = control_unit.generate_signals(decoded)

    # 4. EXECUTE
    if signals.alu_operation:
        result = alu.execute(input_a, input_b, signals.alu_op)

    # 5. MEMORY
    if signals.mem_read:
        data = data_memory.read(address)
    elif signals.mem_write:
        data_memory.write(address, data)

    # 6. WRITEBACK
    if signals.reg_write:
        register_file.write(rd, result)

    # 7. PC UPDATE
    pc = update_pc(signals, decoded)
```

Memory Architecture Detail

INSTRUCTION MEMORY (Harvard Architecture)



DATA MEMORY (Harvard Architecture)



System Initialization Sequence

Startup Flow

```
1. System Check
   ├── Python Version Validation
   ├── Required Files Check
   └── Dependency Verification

2. Component Initialization
   ├── Register File → 16 registers to 0
   ├── ALU → Reset flags and history
   ├── Memory → Clear instruction & data memory
   ├── Decoder → Load ISA table
   └── Control Unit → Initialize control table

3. Program Loading
   ├── Assembly File → Assembler → Machine Code
   └── Machine Code → Instruction Memory

4. Execution Ready
   ├── PC = 0x0000
   ├── All components initialized
   └── Ready for instruction execution
```

Extension Points

Adding New Instructions

```
# 1. Update ISA table in InstructionDecoder
isa_table[0xD] = {
    'name': 'NEW_INST',
    'type': 'R',
    'description': 'New instruction description'
}

# 2. Add control signals in ControlUnit
control_table['NEW_INST'] = {
    'alu_operation': ALU.NEW_OPERATION,
    'reg_write_enable': True,
    # ... other signals
}

# 3. Implement in ALU (if needed)
```

```
def _new_operation(self, a, b):
    # Implementation
    return result

# 4. Add to Assembler parsing
opcodes['NEW_INST'] = 0xD
```

Adding New Components

```
# 1. Create new component class
class NewComponent:
    def __init__(self):
        # Initialize component

    def main_operation(self, inputs):
        # Component functionality

# 2. Integrate in MainCPU
class RiscVProcessor:
    def __init__(self):
        self.new_component = NewComponent()

    def step(self):
        # Use new component in execution
```

Performance Characteristics

Typical Performance Metrics

Component	Operation	Performance
ALU	Basic Operations	1 cycle
Memory	Read/Write	1 cycle
Decoder	Instruction Decode	1 cycle
Overall	Instructions/sec	225,000+
Assembly	Lines/sec	20,000+

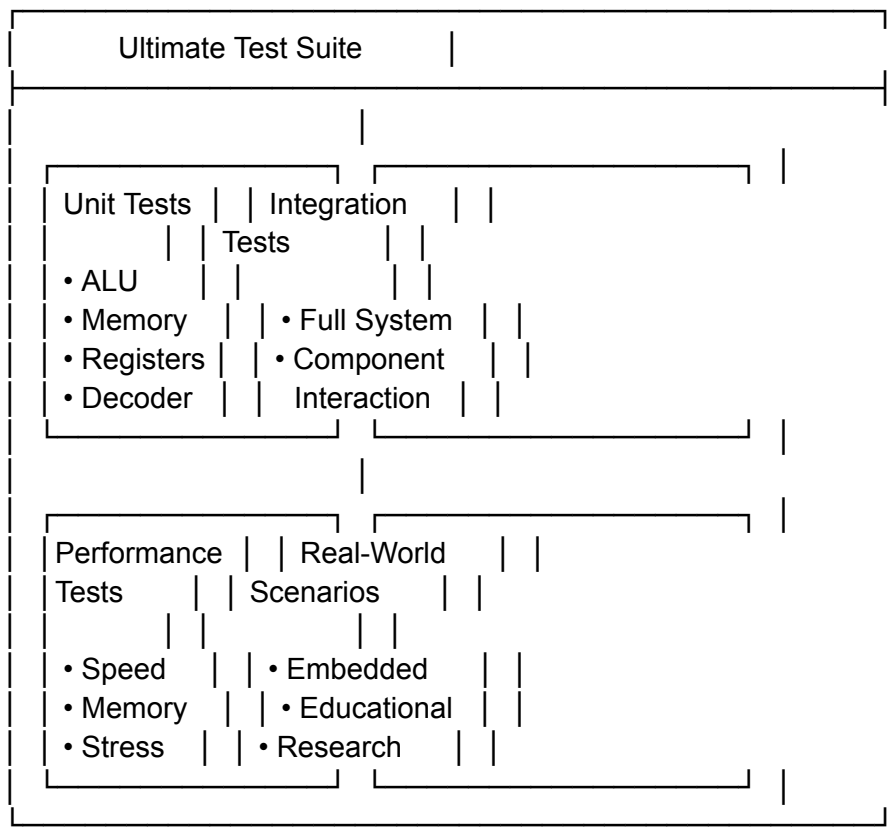
Bottleneck Analysis

Performance Bottlenecks (in order):
1. Memory Operations (highest impact)
2. ALU Complex Operations

3. Branch Prediction
4. Register File Access
5. Instruction Decode (lowest impact)

Testing Architecture

Test Hierarchy



Error Handling Architecture

Exception Hierarchy

```
ProcessorException (Base)
├── MemoryException
│   ├── InvalidAddressException
│   └── MemoryBoundsException
├── InstructionException
│   ├── InvalidOpcodeException
│   └── MalformedInstructionException
├── RegisterException
│   ├── InvalidRegisterException
│   └── ReadOnlyViolationException
```

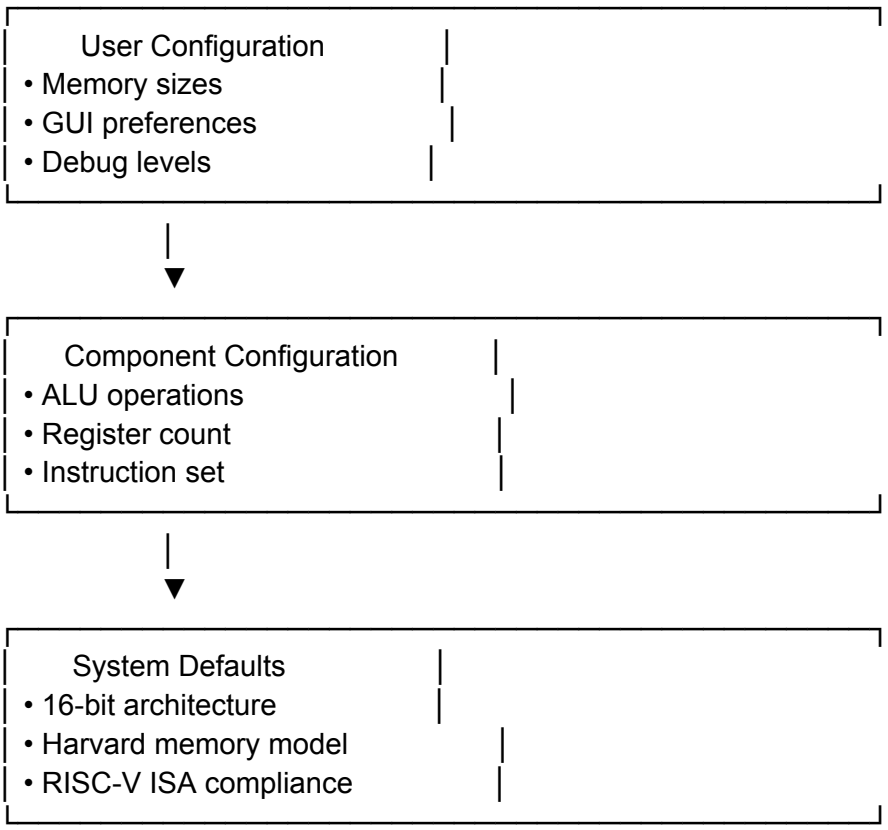
```
└── ExecutionException
    ├── HaltException
    └── TimeoutException
```

Error Recovery Modes

```
# Strict Mode (Development)
try:
    result = memory.read(invalid_address)
except MemoryException:
    raise  # Fail fast for debugging

# Graceful Mode (Production)
result = memory.read(invalid_address)  # Returns 0, logs error
```

Configuration Architecture

System Configuration Layers



Future Architecture Considerations

Planned Extensions

1. **32-bit Support**
    - Expand data path
    - Extended instruction set
    - Larger memory addressing
2. **Pipeline Visualization**
    - Real-time pipeline display
    - Hazard detection
    - Performance analysis
3. **Cache System**
    - L1 instruction cache
    - L1 data cache
    - Cache hit/miss statistics
4. **Interrupt System**
    - Interrupt controller
    - Exception vectors
    - Context switching

Conclusion

The RISC-V simulator architecture provides a solid foundation for:

- **Educational Use**: Clear component separation for learning
- **Research**: Extensible design for experimentation
- **Development**: Modular structure for feature additions
- **Testing**: Comprehensive validation framework

The Harvard architecture with separate memories, combined with the modular component design, creates a system that is both educationally valuable and technically sound.