Single-Cycle MIPS Processor - Complete Project Documentation

Authors: Artin Zareie | Mohsen Mirzaei
Date: August 2025
Course: Computer Architecture Project 2

Table of Contents

1. Project Overview
2. Project Structure
3. Instruction Set Architecture (ISA)
4. CPU Architecture
5. Component Documentation
6. Memory System
7. Development Tools
8. Testing and Verification
9. Example Programs
10. Build and Simulation
11. Design Decisions and Features
12. Future Enhancements

Project Overview

This project implements a **single-cycle MIPS processor** in Verilog HDL. The processor executes each instruction in exactly one clock cycle, providing a simplified but educational implementation of the classic MIPS architecture.

Key Features

- **Single-cycle execution**: All instructions complete in one clock cycle
- **32-bit MIPS architecture**: Full 32-bit data and address paths
- **Comprehensive instruction set**: R-type, I-type, and J-type instructions
- **Custom crypto instructions**: Hardware-accelerated encryption/decryption
- **Complete memory hierarchy**: Instruction and data memory subsystems
- **Full simulation environment**: Comprehensive testbenches and verification tools
- **Assembly toolchain**: Custom assembler for converting MIPS assembly to machine code

Educational Goals

The processor serves as an educational platform for understanding: - Computer architecture fundamentals - Instruction set design and implementation - Datapath and control unit design - Memory hierarchy organization - Hardware description language (Verilog) development - Digital design verification methodologies

---

Project Structure

```
single-cycle-mips/
|-- docs/                      # Documentation files
|   |-- ALU.md               # ALU component documentation
|   |-- ControlUnit.md       # Control unit documentation
|   |-- Crypt.md             # Crypto unit documentation
|   |-- DataMem.md           # Data memory documentation
|   |-- DRAM.md              # DRAM module documentation
|   |-- GPRF.md              # General Purpose Register File documentation
|   |-- ImmExt.md            # Immediate extension unit documentation
|   |-- InstMem.md           # Instruction memory documentation
|   |-- ISA.md               # Instruction Set Architecture specification
|   |-- MIPS_CPU.md          # Top-level CPU documentation
|   |-- Mux.md               # Multiplexer documentation
|   `-- PROJECT_DOCUMENTATION.md  # This comprehensive documentation
|-- src/                       # Source code directory
|   |-- mips_cpu.v          # Top-level CPU module
|   |-- core/               # Core processor components
|   |   |-- alu_control.v  # ALU control unit
|   |   |-- alu.v          # Arithmetic Logic Unit
|   |   |-- control_unit.v # Main control unit
|   |   |-- crypt.v        # Cryptographic unit
|   |   |-- immext.v       # Immediate extension unit
|   |   |-- program_counter.v # Program counter
|   |   `-- reg_file.v     # Register file
|   |-- helpers/            # Helper modules
|   |   |-- mux21.v        # 2:1 multiplexer
|   |   `-- mux41.v        # 4:1 multiplexer
|   `-- mem/                # Memory subsystem
|       |-- dmem.v         # Data memory
|       |-- dram.v         # DRAM controller
|       `-- imem.v         # Instruction memory
|-- sim/                    # Simulation files
|   |-- tb/                 # Testbench files
|   |   |-- alu_tb.v       # ALU testbench
|   |   |-- crypt_tb.v     # Crypto unit testbench
|   |   |-- dmem_tb.v      # Data memory testbench
|   |   |-- dram_tb.v      # DRAM testbench
|   |   |-- gprf_tb.v      # Register file testbench
|   |   |-- imem_tb.v      # Instruction memory testbench
|   |   |-- immext_tb.v    # Immediate extension testbench
|   |   |-- mips_cpu_tb.v  # CPU testbench
|   |   |-- mux21_tb.v     # 2:1 mux testbench
|   |   |-- mux41_tb.v     # 4:1 mux testbench
|   |   `-- tb_InnerProduct.v # Dot product test
|   `-- stimuli/           # Test stimulus files
|       |-- alu_ports.csv
|       |-- assembled.hex
|       |-- crypt_ports.csv
|       |-- dram_ports.csv
|       |-- gprf_ports.csv
|       `-- immext_ports.csv
|-- asm-codes/             # Assembly code examples
|   |-- asm-code.asm       # Basic assembly examples
|   |-- dot-product.asm    # Dot product implementation
|   |-- dot-product.hex    # Assembled dot product
|   |-- final-test.asm     # Final test program
|   `-- final-test.hex     # Assembled final test
|-- build/                 # Build output directory
|-- refs/                  # Reference documents
|-- assembler.py           # MIPS assembler tool
|-- playground.py          # Development playground
`-- makefile              # Build automation
```

Instruction Set Architecture (ISA)

The processor implements a subset of the MIPS32 instruction set with custom extensions for cryptographic operations.

Instruction Formats

R-Type Instructions

```
| opcode (6) | rs (5) | rt (5) | rd (5) | shamt (5) | funct (6) |
|    0x00    |        |        |        |           |           |
```

I-Type Instructions

```
| opcode (6) | rs (5) | rt (5) | immediate (16) |
|            |        |        |                |
```

J-Type Instructions

```
| opcode (6) | address (26) |
|            |              |
```

Supported Instructions

Arithmetic Operations

- **add rd, rs, rt**: Add two registers
- **addi rt, rs, imm**: Add immediate to register
- **sub rd, rs, rt**: Subtract two registers
- **subi rt, rs, imm**: Subtract immediate from register (alias for addi with negative immediate)
- **mul rd, rs, rt**: Multiply two registers

Logical Operations

- **and rd, rs, rt**: Bitwise AND
- **or rd, rs, rt**: Bitwise OR
- **xor rd, rs, rt**: Bitwise XOR

Shift and Rotate Operations

- **sll rd, rt, shamt**: Shift left logical (immediate)
- **srl rd, rt, shamt**: Shift right logical (immediate)
- **sra rd, rt, shamt**: Shift right arithmetic (immediate)
- **sllv rd, rt, rs**: Shift left logical (variable)
- **srlv rd, rt, rs**: Shift right logical (variable)
- **srav rd, rt, rs**: Shift right arithmetic (variable)
- **rol rd, rt, shamt**: Rotate left (immediate)
- **ror rd, rt, shamt**: Rotate right (immediate)
- **rolv rd, rt, rs**: Rotate left (variable)
- **rorv rd, rt, rs**: Rotate right (variable)

Memory Operations

- **lw rt, offset(rs)**: Load word from memory
- **sw rt, offset(rs)**: Store word to memory

Control Transfer Operations

- **beq rs, rt, label**: Branch if equal
- **jmp address**: Unconditional jump
- **j address**: Jump (alias for jmp)

Custom Cryptographic Operations

- **enc rd, rs, rt**: Encrypt data (custom implementation)
- **dec rd, rs, rt**: Decrypt data (custom implementation)

Register Conventions

The processor implements 32 general-purpose registers following MIPS conventions:

Register	Number	Usage
$zero	0	Always zero
$at	1	Assembler temporary
$v0-$v1	2-3	Function return values
$a0-$a3	4-7	Function arguments
$t0-$t7	8-15	Temporary registers
$s0-$s7	16-23	Saved registers
$t8-$t9	24-25	More temporaries
$k0-$k1	26-27	Kernel registers
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

CPU Architecture

Datapath Overview

The single-cycle MIPS processor implements a classic 5-stage datapath collapsed into a single cycle:

1. **Instruction Fetch (IF)**: Retrieve instruction from memory
2. **Instruction Decode (ID)**: Decode instruction and read registers
3. **Execute (EX)**: Perform ALU operations
4. **Memory Access (MEM)**: Access data memory if needed
5. **Write Back (WB)**: Write results back to register file

Top-Level Interface

```verilog
module MIPS_CPU(
    input wire clk,                     // System clock
    input wire reset,                   // Synchronous reset

    output wire [31:0] pc_debug,           // Current PC value (for debugging)
    output wire [31:0] instruction_debug,  // Current instruction (for debugging)
    output wire [31:0] alu_result_debug,   // ALU result (for debugging)
    output wire [31:0] mem_data_debug      // Memory read data (for debugging)
);
```

Datapath Components

Program Counter (PC) Management

- **PC+4 calculation**: Sequential instruction addressing
- **Branch address**: `PC + 4 + (sign_ext_imm << 2)`
- **Jump address**: `{PC[31:28], instruction[25:0], 2'b00}`
- **PC source selection**: 4:1 mux controlled by branch/jump logic

Control Path The control unit generates all necessary control signals based on the instruction opcode and function code: - Register write enable and destination selection - ALU operation selection - Memory read/write controls - Branch and jump controls - Immediate extension control

Data Path

- 32-bit wide data paths throughout
- Multiple multiplexers for data routing
- Support for both register-register and register-immediate operations

---

Component Documentation

Control Unit

Module: `ControlUnit`
File: `src/core/control_unit.v`

The control unit is the brain of the processor, generating all control signals based on the instruction opcode and function code.

Interface:

```verilog
module ControlUnit(
    input [5:0] opcode,
    input [5:0] funct,
    output reg Branch,
    output reg Jump,
    output reg MemRead,
    output reg MemWrite,
    output reg [1:0] RegWriteSrc,
    output reg RegWrite,
    output reg [1:0] RegDst,
    output reg ALUSrc,
    output reg SignExtend,
    output reg ShiftOp,
    output reg VarShift
);
```

Functionality: - Decodes instruction opcodes and function codes - Generates control signals for all datapath components - Handles special cases for shift/rotate operations - Supports both register and immediate operand modes

Arithmetic Logic Unit (ALU)

Module: `ALU`
File: `src/core/alu.v`

The ALU performs all arithmetic, logical, shift, and rotate operations.

Interface:

```
module ALU(
    input [31:0] a, b,
    input [3:0] alu_op,
    output reg [31:0] result,
    output zero
);
```

Supported Operations: - Arithmetic: ADD, SUB, MUL - Logical: AND, OR, XOR - Shifts: SLL, SRL, SRA - Rotates: ROL, ROR - Comparison: equality check for branches

Register File (GPRF)

Module: `RegFile`
File: `src/core/reg_file.v`

32-register general-purpose register file with dual read ports and single write port.

Interface:

```
module RegFile(
    input clk,
    input reset,
    input [4:0] read_addr1, read_addr2, write_addr,
    input [31:0] write_data,
    input write_enable,
    output [31:0] read_data1, read_data2
);
```

Features: - Dual-port read capability - Single-port write capability - Register $0 hardwired to zero - Synchronous write, asynchronous read

Memory Subsystem

Instruction Memory (InstMem) **Module**: `InstMem`
File: `src/mem/imem.v`

Stores program instructions with word-addressed access.

Data Memory (DataMem) **Module**: `DataMem`
File: `src/mem/dmem.v`

Stores program data with load/store operations.

DRAM Controller **Module**: `DRAM`
File: `src/mem/dram.v`

Low-level memory interface providing byte-addressable access.

Cryptographic Unit

Module: `Crypt`
File: `src/core/crypt.v`

Custom hardware acceleration for encryption/decryption operations.

Interface:

```
module Crypt(
    input [31:0] data,
    input [31:0] key,
    input encrypt, // 1 for encrypt, 0 for decrypt
    output [31:0] result
);
```

Immediate Extension Unit

Module: `ImmExt`
File: `src/core/immext.v`

Extends 16-bit immediates to 32-bit values with sign or zero extension.

Helper Modules

2:1 Multiplexer **Module**: `Mux21`
File: `src/helpers/mux21.v`

4:1 Multiplexer **Module**: `Mux41`
File: `src/helpers/mux41.v`

Memory System

Memory Organization

The processor uses a Harvard architecture with separate instruction and data memories:

Instruction Memory

- **Size**: Configurable (default 1024 words)
- **Width**: 32 bits
- **Addressing**: Word-aligned (addresses in increments of 4)
- **Access**: Read-only during execution
- **Initialization**: Loaded from HEX files

Data Memory

- **Size**: Configurable (default 1024 words)
- **Width**: 32 bits
- **Addressing**: Byte-addressable with word alignment
- **Access**: Read/write operations
- **Initialization**: Typically initialized to zero

Memory Interface

Both memories use a unified DRAM interface for consistency:

```
module DRAM(
    input clk,
    input [31:0] addr,
    input [31:0] wdata,
    input we,
    output [31:0] rdata
);
```

Memory Map

Address Range	Usage
0x00000000 - 0x000003FF	Instruction Memory (1024 words)
0x10000000 - 0x100003FF	Data Memory (1024 words)

Development Tools

MIPS Assembler

File: `assembler.py`

A comprehensive Python-based assembler that converts MIPS assembly language to machine code.

Features

- Support for all ISA instructions
- Label resolution for branches and jumps
- Register name aliasing (both $name and $number formats)
- Comprehensive error checking and reporting
- Multiple output formats (binary, hexadecimal)

Usage

```
python assembler.py input.asm -o output.hex
```

Supported Assembly Syntax

```
# Comments start with #
label:
    addi $t0, $zero, 10    # Add immediate
    lw $t1, 0($sp)         # Load word
    beq $t0, $t1, label    # Branch if equal
    jmp end                # Jump
end:
    # End of program
```

Build System

File: `makefile`

Comprehensive makefile supporting: - Individual component testing - Full system simulation - Waveform generation - Automated test execution

Key Targets

```
make test-gprf          # Test register file
make test-alu           # Test ALU
make test-mips-cpu      # Test complete CPU
make test-dot-product   # Test dot product program
make clean              # Clean build artifacts
```

Simulation Environment

The project uses **Icarus Verilog** for simulation and **GTKWave** for waveform viewing.

Simulation Flow

1. **Compilation**: Verilog files compiled with `iverilog`
2. **Simulation**: Test execution with `vvp`
3. **Waveform Generation**: VCD files for debugging
4. **Analysis**: Waveform viewing with `gtkwave`

Testing and Verification

Testing Strategy

The project employs a comprehensive testing strategy:

1. **Unit Testing**: Individual component verification
2. **Integration Testing**: Inter-component communication
3. **System Testing**: Complete processor functionality
4. **Program Testing**: Real assembly programs execution

Testbench Architecture

Each component has a dedicated testbench following a consistent pattern:

```
module component_tb;
    // Test signals
    reg clk, reset;
    reg [31:0] test_input;
    wire [31:0] test_output;

    // Component instantiation
    Component uut (
        .clk(clk),
        .reset(reset),
        .input(test_input),
        .output(test_output)
    );

    // Clock generation
    always #5 clk = ~clk;

    // Test sequence
    initial begin
        // Test cases
        $dumpfile("build/component_tb.vcd");
        $dumpvars(0, component_tb);

        // Simulation
        #1000 $finish;
    end
endmodule
```

Test Coverage

Component Tests

- **ALU**: All arithmetic, logical, shift, and rotate operations
- **Control Unit**: All instruction types and special cases
- **Register File**: Read/write operations, hazard conditions
- **Memory**: Load/store operations, addressing modes
- **Crypto Unit**: Encryption/decryption functionality

System Tests

- **Instruction Execution**: Complete instruction set verification
- **Program Execution**: Real programs like dot product calculation
- **Edge Cases**: Boundary conditions and error scenarios

Verification Methodology

1. **Functional Verification**: Correct operation verification
2. **Timing Verification**: Single-cycle timing constraints
3. **Coverage Analysis**: Instruction and branch coverage
4. **Regression Testing**: Automated test suite execution

Example Programs

Dot Product Calculator

File: `asm-codes/dot-product.asm`

Demonstrates vector operations and memory access patterns:

```
# Dot Product Calculator
# Computes the dot product of two 4-element vectors
# Vector A stored at word addresses 0, 1, 2, 3
# Vector B stored at word addresses 4, 5, 6, 7
# Result stored in register $s6

# Initialize vector pointers
addi $s0, $0, 0      # $s0 = base address of Vector A
addi $s1, $0, 16     # $s1 = base address of Vector B

# Initialize loop control variables
addi $s2, $0, 4      # $s2 = loop limit (4 elements)
addi $s3, $0, 0      # $s3 = loop counter
addi $s6, $0, 0      # $s6 = accumulator for result

loop:
    # Load current elements from both vectors
    lw $s4, 0($s0)       # $s4 = A[i]
    lw $s5, 0($s1)       # $s5 = B[i]

    # Compute element-wise product
    mul $t0, $s4, $s5    # $t0 = A[i] * B[i]

    # Add to running sum
    add $s6, $s6, $t0    # $s6 += A[i] * B[i]

    # Move to next elements
    addi $s0, $s0, 4     # Increment Vector A pointer
    addi $s1, $s1, 4     # Increment Vector B pointer
    addi $s3, $s3, 1     # Increment loop counter

    # Check loop condition
    beq $s3, $s2, done   # Exit if processed all elements
    jmp loop             # Continue loop

done:
    # Result is in $s6
    # End of program
```

This program demonstrates: - Memory addressing and pointer arithmetic - Loop control structures - Arithmetic operations (multiplication and addition) - Conditional branching - Register usage conventions

Final Test Program

File: `asm-codes/final-test.asm`

Comprehensive test exercising all instruction types and processor features.

Build and Simulation

Prerequisites

- **Icarus Verilog**: Open-source Verilog simulator
- **GTKWave**: Waveform viewer
- **Python 3.x**: For assembler tool
- **Make**: Build automation

Installation (Ubuntu/Debian)

```
sudo apt-get update
sudo apt-get install iverilog gtkwave python3 make
```

Build Commands

Component Testing

```
# Test individual components
make test-alu           # Test ALU functionality
make test-gprf          # Test register file
make test-control       # Test control unit
make test-memory        # Test memory subsystem
```

System Testing

```
# Test complete CPU
make test-mips-cpu      # Basic CPU functionality
make test-dot-product   # Dot product program execution
```

Assembly and Simulation

```
# Assemble program
python assembler.py asm-codes/program.asm -o build/program.hex

# Run simulation
make PROGRAM=program test-mips-cpu
```

Waveform Analysis

```
# Generate and view waveforms
make test-mips-cpu
gtkwave build/mips_cpu_tb.vcd
```

Debugging Workflow

1. **Write/modify assembly program**
2. **Assemble to machine code**
3. **Run simulation**
4. **Analyze waveforms**
5. **Debug and iterate**

Performance Analysis

The single-cycle design allows for: - **Clock Period**: Determined by longest combinational path - **Throughput**: One instruction per clock cycle - **Latency**: One clock cycle per instruction

---

Design Decisions and Features

Single-Cycle Implementation

Advantages: - Simplified control logic - Predictable timing - Educational clarity - No pipeline hazards

Trade-offs: - Lower maximum clock frequency - Some hardware duplication - Longer critical path

Custom Extensions

Cryptographic Instructions The processor includes custom encryption/decryption instructions demonstrating: - ISA extensibility - Custom functional units - Application-specific acceleration

Enhanced Shift/Rotate Operations Complete set of shift and rotate operations including: - Immediate and variable shift amounts - Logical and arithmetic shifts - Left and right rotations

Memory Architecture

Harvard architecture chosen for: - Simplified single-cycle implementation - Clear separation of instruction and data - Educational demonstration of memory hierarchy

Register File Design

Dual-port read architecture enables: - Simultaneous access to two source operands - Efficient R-type instruction execution - Single-cycle operation support

---

Future Enhancements

Performance Improvements

1. **Pipelined Implementation**: Convert to 5-stage pipeline for higher throughput
2. **Cache Memory**: Add instruction and data caches
3. **Branch Prediction**: Implement simple branch prediction mechanisms
4. **Forwarding Logic**: Add data forwarding for pipeline hazards

ISA Extensions

1. **Floating-Point Unit**: Add IEEE 754 floating-point support
2. **Vector Instructions**: SIMD operations for parallel computation

3. **Privileged Instructions**: Supervisor mode and system calls
4. **Advanced Crypto**: AES, SHA hardware acceleration

Development Tools

1. **Debugger Interface**: GDB-compatible debugging support
2. **Profiling Tools**: Performance analysis and optimization
3. **Advanced Assembler**: Macro support, linking capabilities
4. **C Compiler Backend**: Generate MIPS code from C programs

System Integration

1. **Peripheral Interfaces**: UART, SPI, I2C controllers
2. **Interrupt Controller**: Hierarchical interrupt management
3. **Memory Management**: Virtual memory and protection
4. **Multi-core Support**: Symmetric multiprocessing capabilities

---

Conclusion

This single-cycle MIPS processor project successfully demonstrates the fundamental concepts of computer architecture and digital design. The implementation provides a solid foundation for understanding:

- Instruction set architecture design
- Processor datapath and control logic
- Memory system organization
- Hardware description language development
- Digital system verification methodologies

The project serves as an excellent educational platform and a stepping stone for more advanced processor implementations. The clean, modular design facilitates easy extension and modification for future enhancements.

The comprehensive toolchain, including assembler and simulation environment, provides a complete development ecosystem for experimenting with computer architecture concepts and validating design decisions.

References

1. **Patterson, D. A., & Hennessy, J. L.** (2020). *Computer organization and design: the hardware/software interface.* Morgan Kaufmann.

2. **Harris, D. M., & Harris, S. L.** (2021). *Digital design and computer architecture.* Morgan Kaufmann.

3. **MIPS Technologies Inc.** (2001). *MIPS32™ Architecture For Programmers Volume I: Introduction to the MIPS32™ Architecture.*

4. **Vahid, F.** (2010). *Digital design with RTL design, VHDL, and Verilog.* John Wiley & Sons.

End of Documentation