Build your own RISC-V Processor in a Day

From ISA to RTL Implementation

Maktab e Digital Systems (MEDS)

UET Lahore

July 16, 2025

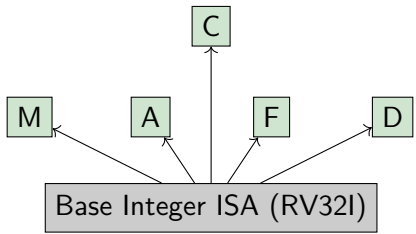
Workshop Agenda

1. **Session 1: Introduction to RISC-V**
2. **Session 2: Assembly Programming**
3. **Session 3: Processor Architecture**
4. **Session 4: RTL Implementation**
5. **Session 5: Verification & Testing**

What is RISC-V?

- **RISC-V** = Reduced Instruction Set Computer
- Open-source instruction set architecture (ISA)
- Developed at UC Berkeley
- Pronounced "risk-five"
- Key advantages:
  - Open and free
  - Modular design
  - Suitable for all computing domains
  - Growing ecosystem

RISC-V ISA Modularity



Today's Focus: RV32I Base Integer ISA

RISC-V Register File

Register	ABI Name	Description
x0	zero	Always zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5-x7	t0-t2	Temporary
x8	s0/fp	Saved/Frame pointer
x9	s1	Saved register
x10-x11	a0-a1	Arguments/return
x12-x17	a2-a7	Arguments
x18-x27	s2-s11	Saved registers
x28-x31	t3-t6	Temporary

- 32 registers in RV32I
- Each register is 32 bits wide
- x0 is hardwired to zero
- ABI names for software compatibility
- Today we'll use x0-x31 notation

RISC-V Instruction Formats

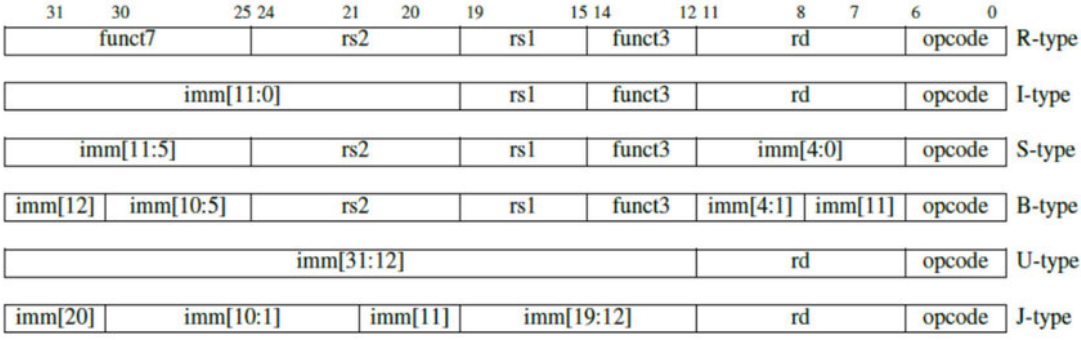


Figure: RISC-V Instruction Format

Our Subset of RV32I Instructions

Arithmetic & Logic (R-type):

- `add rd, rs1, rs2`
- `sub rd, rs1, rs2`
- `and rd, rs1, rs2`
- `or rd, rs1, rs2`
- `xor rd, rs1, rs2`
- `slt rd, rs1, rs2`

Immediate (I-type):

- `addi rd, rs1, imm`
- `andi rd, rs1, imm`
- `ori rd, rs1, imm`
- `xori rd, rs1, imm`
- `slti rd, rs1, imm`

Memory (I/S-type):

- `lw rd, offset(rs1)`
- `sw rs2, offset(rs1)`

Branch (B-type):

- `beq rs1, rs2, offset`
- `bne rs1, rs2, offset`
- `blt rs1, rs2, offset`
- `bge rs1, rs2, offset`

Jump (J-type):

- `jal rd, offset`

Instruction Encoding Examples

Example 1: `add x1, x2, x3`

funct7	rs2	rs1	funct3	rd	opcode
0000000	00011	00010	000	00001	0110011

Machine code: `0x003100B3`

Example 2: `addi x1, x2, 100`

imm[11:0]	rs1	funct3	rd	opcode
000001100100	00010	000	00001	0010011

Machine code: `0x06410093`

C to Assembly: Basic Operations

C Code:

```c
int a = 5;
int b = 10;
int c = a + b;
```

RISC-V Assembly:

```
addi x1, x0, 5     # a = 5
addi x2, x0, 10    # b = 10
add  x3, x1, x2    # c = a + b
```

Key Points:

- Variables map to registers
- Constants use immediate instructions
- `x0` is always zero (useful for loading immediates)

C to Assembly: Conditional Statements

C Code:

```c
if (a == b) {
    c = a + b;
} else {
    c = a - b;
}
```

RISC-V Assembly:

```
beq x1, x2, equal
sub x3, x1, x2     # else case
jal x0, end        # skip equal
equal:
add x3, x1, x2     # if case
end:
```

Key Points:

- Conditional branches test conditions
- Unconditional jumps for control flow
- Labels mark target addresses

C to Assembly: Loops

C Code:

```c
int sum = 0;
for (int i = 0; i < 10; i++) {
    sum = sum + i;
}
```

RISC-V Assembly:

```
addi x1, x0, 0     # sum = 0
addi x2, x0, 0     # i = 0
addi x3, x0, 10    # limit = 10
loop:
bge  x2, x3, end   # if i >= 10, exit
add  x1, x1, x2    # sum += i
addi x2, x2, 1     # i++
jal  x0, loop      # repeat
end:
```

Memory Operations

C Code:

```c
int arr[5] = {1, 2, 3, 4, 5};
int x = arr[2];
arr[3] = x + 1;
```

RISC-V Assembly:

```
# Assume arr base in x10
lw  x1, 8(x10)     # x = arr[2]
addi x2, x1, 1     # x + 1
sw  x2, 12(x10)    # arr[3] = x
   + 1
```

Key Points:

- Array indexing uses byte addressing
- Each integer = 4 bytes
- `arr[i]` → `offset = i * 4`

Assembly to Machine Code

Step 1: Resolve Labels

```
0x0000: addi x1, x0, 0     # sum = 0
0x0004: addi x2, x0, 0     # i = 0
0x0008: addi x3, x0, 10    # limit = 10
0x000C: bge  x2, x3, 0x001C  # loop: if i >= 10, exit
0x0010: add  x1, x1, x2    # sum += i
0x0014: addi x2, x2, 1     # i++
0x0018: jal  x0, 0x000C    # jump to loop
0x001C: # end:
```

Step 2: Encode Instructions

- `addi x1, x0, 0` → `0x00000093`
- `bge x2, x3, 0x14` → `0x0031D863`
- `jal x0, 0x000C` → `0xFF5FF06F`

Creating Instruction Memory

Memory Initialization File (mem.hex):

```
00000093  // addi x1, x0, 0
00000113  // addi x2, x0, 0
00A00193  // addi x3, x0, 10
0031D863  // bge x2, x3, 20
002080B3  // add x1, x1, x2
00110113  // addi x2, x2, 1
FF5FF06F  // jal x0, -12
```

Creating Instruction Memory

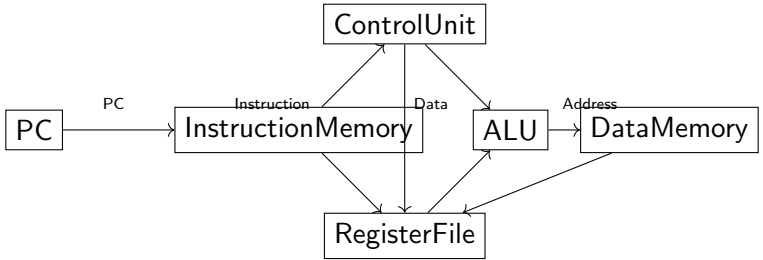
In SystemVerilog:

```
module instruction_memory (
    input  logic [31:0] addr,
    output logic [31:0] instruction
);
    logic [31:0] mem [0:1023];

    initial begin
        $readmemh("mem.hex", mem);
    end

    assign instruction = mem[addr[31:2]];
endmodule
```

Single-Cycle Processor Overview



Key Components:

- Program Counter (PC)
- Instruction Memory
- Control Unit
- Register File
- ALU
- Data Memory

Instruction Fetch, Decode, Execute

1. **Fetch:** Read instruction from memory at PC address
2. **Decode:**
   - Extract fields (opcode, rs1, rs2, rd, immediate)
   - Generate control signals
   - Read register file
3. **Execute:**
   - Perform ALU operation
   - Access data memory (if needed)
   - Write back to register file
   - Update PC

Single-Cycle: All steps complete in one clock cycle

Detailed Datapath

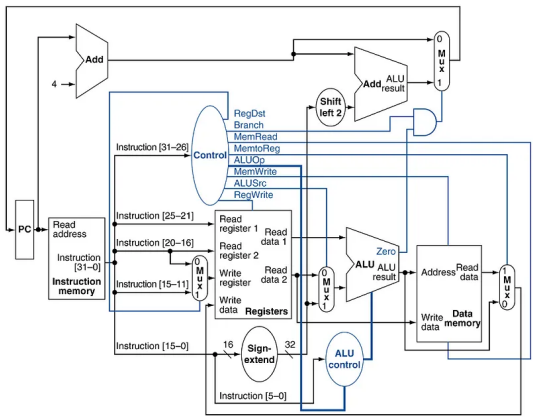


Figure: RISC-V Datapath

Control Unit Design

Instruction	RegWrite	ALUSrc	MemRead	MemWrite	Branch	MemtoReg	ALUOp
R-type	1	0	0	0	0	0	10
I-type (ALU)	1	1	0	0	0	0	10
Load	1	1	1	0	0	1	00
Store	0	1	0	1	0	X	00
Branch	0	0	0	0	1	X	01

Control Signal Functions:

- **RegWrite:** Enable register file write
- **ALUSrc:** Select ALU input (register vs immediate)
- **MemRead/MemWrite:** Enable data memory operations
- **Branch:** Enable branch condition check
- **MemtoReg:** Select register write data source
- **ALUOp:** ALU operation type

ALU Control

ALUOp	Funct3	Funct7[5]	ALU Control
00	XXX	X	0010 (add)
01	XXX	X	0110 (sub)
10	000	0	0010 (add)
10	000	1	0110 (sub)
10	001	X	0000 (and)
10	010	X	0001 (or)
10	100	X	0100 (xor)
10	101	X	0111 (slt)

ALU Control

ALU Operations:

- 0000: AND
- 0001: OR
- 0010: ADD
- 0100: XOR
- 0110: SUB
- 0111: SLT (Set Less Than)

Top-Level Processor Module

```
module processor (
    input  logic clk,
    input  logic reset,
    output logic [31:0] pc_out,
    output logic [31:0] instruction_out
);
```

Top-Level Processor Module

```
    // Datapath Signals
    logic [31:0] pc, pc_next, pc_plus_4;
    logic [31:0] instruction;
    logic [31:0] read_data1, read_data2, write_data;
    logic [31:0] alu_result, mem_read_data;
    logic [31:0] immediate;
    logic [4:0] rs1, rs2, rd;
    logic [6:0] opcode;
    logic [2:0] funct3;
    logic [6:0] funct7;
```

Top-Level Processor Module

```
1
2
3       // Control signals
4       logic reg_write, alu_src, mem_read, mem_write;
5       logic branch, mem_to_reg, jump;
6       logic [3:0] alu_control;
7       logic zero_flag;
8
9       // Instantiate modules here...
10
11  endmodule
```

Register File Implementation

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

Register File Implementation

```
1
2     logic [31:0] registers [0:31];
3
4     // x0 is always zero
5     assign registers[0] = 32'h0;
6
7     // Read operations (combinational)
8     assign read_data1 = registers[read_addr1];
9     assign read_data2 = registers[read_addr2];
```

Register File Implementation

```
1
2
3       // Write operation (sequential)
4       always_ff @(posedge clk) begin
5           if (reset) begin
6               for (int i = 1; i < 32; i++) begin
7                   registers[i] <= 32'h0;
8               end
9           end else if (write_enable && write_addr != 0) begin
10              registers[write_addr] <= write_data;
11          end
12      end
13
14  endmodule
```

ALU Implementation

```
module alu (
    input  logic [31:0] operand_a,
    input  logic [31:0] operand_b,
    input  logic [3:0]  alu_control,
    output logic [31:0] result,
    output logic        zero
);
```

ALU Implementation

```
1
2       always_comb begin
3           case (alu_control)
4               4'b0000: result = operand_a & operand_b;       // AND
5               4'b0001: result = operand_a | operand_b;       // OR
6               4'b0010: result = operand_a + operand_b;       // ADD
7               4'b0100: result = operand_a ^ operand_b;       // XOR
8               4'b0110: result = operand_a - operand_b;       // SUB
9               4'b0111: result = (operand_a < operand_b) ? 1 : 0; //
                    SLT
10              default: result = 32'h0;
11          endcase
12      end
13
14      assign zero = (result == 32'h0);
15
16  endmodule
```

Control Unit Implementation

```
module control_unit (
    input  logic [6:0] opcode,
    input  logic [2:0] funct3,
    input  logic [6:0] funct7,
    output logic       reg_write,
    output logic       alu_src,
    output logic       mem_read,
    output logic       mem_write,
    output logic       branch,
    output logic       mem_to_reg,
    output logic       jump,
    output logic [3:0] alu_control
);
```

Control Unit Implementation

```
1
2     logic [1:0] alu_op;
3
4     // Main control signals
5     always_comb begin
6         case (opcode)
7             7'b0110011: begin // R-type
8                 reg_write = 1; alu_src = 0; mem_read = 0;
9                 mem_write = 0; branch = 0; mem_to_reg = 0;
10                jump = 0; alu_op = 2'b10;
11            end
12            // Add more cases...
13            default: begin
14                reg_write = 0; alu_src = 0; mem_read = 0;
15                mem_write = 0; branch = 0; mem_to_reg = 0;
16                jump = 0; alu_op = 2'b00;
17            end
```

Immediate Generation

```
module immediate_generator (
    input  logic [31:0] instruction,
    input  logic [6:0]  opcode,
    output logic [31:0] immediate
);
```

Immediate Generation

```
1
2       always_comb begin
3           case (opcode)
4               7'b0010011: begin // I-type
5                   immediate = {{20{instruction[31]}}, instruction
                        [31:20]};
6               end
7               7'b0100011: begin // S-type
8                   immediate = {{20{instruction[31]}},
9                               instruction[31:25], instruction
                                    [11:7]};
10              end
11              default: immediate = 32'h0;
12          endcase
13      end
14
15  endmodule
```

Testbench Structure

```
module tb_processor;
    logic clk, reset;
    logic [31:0] pc_out, instruction_out;

    // Instantiate processor
    processor dut (
        .clk(clk),
        .reset(reset),
        .pc_out(pc_out),
        .instruction_out(instruction_out)
    );
```

Testbench Structure

```
    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    // Test sequence
    initial begin
        reset = 1;
        #10 reset = 0;
```

Testbench Structure

```
        // Run for several cycles
        repeat (20) @(posedge clk);

        $finish;
    end

    // Monitor outputs
    initial begin
        $monitor("Time=%0t PC=%h Instruction=%h",
                 $time, pc_out, instruction_out);
    end

endmodule
```

ALU Testbench

```
module tb_alu;
    logic [31:0] operand_a, operand_b, result;
    logic [3:0] alu_control;
    logic zero;

    // Instantiate ALU
    alu dut (.*);
    // Test vectors
    initial begin
        $display("Testing ALU...");

        // Test ADD
        operand_a = 32'h10; operand_b = 32'h20; alu_control = 4'
           b0010;
        #10 assert(result == 32'h30) else $error("ADD failed");

        // Test SUB ...
```

Workshop Summary

What we've accomplished:

- Understood RISC-V ISA fundamentals
- Converted C code to assembly
- Generated machine code
- Designed single-cycle datapath
- Implemented control logic
- Coded complete processor in SystemVerilog
- Verified with comprehensive testbenches

Next steps:

- Extend instruction set support
- Add pipeline stages
- Optimize for performance

Performance Metrics

Our Single-Cycle Processor:

- **Clock Period:** Limited by longest instruction path
- **CPI:** 1 (Cycles Per Instruction)
- **Instruction Set:** 20+ RV32I instructions
- **Memory:** Harvard architecture (separate I/D)
- **Area:** Approximately 5000 gates

Typical Performance:

- **Frequency:** 50-100 MHz on FPGA
- **Throughput:** 50-100 MIPS
- **Power:** Low (no complex features)
- **Use cases:** Embedded systems, IoT, education

Common Debug Techniques

Simulation Debug:

- Use `$display` and `$monitor` statements
- Add assertions for critical conditions
- Use waveform viewers (GTKWave, ModelSim)
- Check control signal timing

Hardware Debug:

- Add debug ports to processor
- Use FPGA debug cores (ILA, VIO)
- Implement simple serial output
- Use LED indicators for status

Common Issues:

- Incorrect immediate sign extension
- Wrong ALU control signals

Questions & Discussion

Questions?

- What challenges did you face?
- Which part was most interesting?
- Ideas for processor improvements?
- Real-world applications?

Thank you for attending!

Keep building, keep learning!

References & Resources

- **RISC-V ISA Specification:**
  `https://riscv.org/technical/specifications/`
- **"Computer Organization and Design RISC-V Edition":**
  Patterson & Hennessy
- **SystemVerilog IEEE Standard:**
  IEEE 1800-2017
- **Online RISC-V Simulator:**
  `https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/`
- **Workshop GitHub Repository:**
  `https://github.com/meds-uet/rv-workshop`