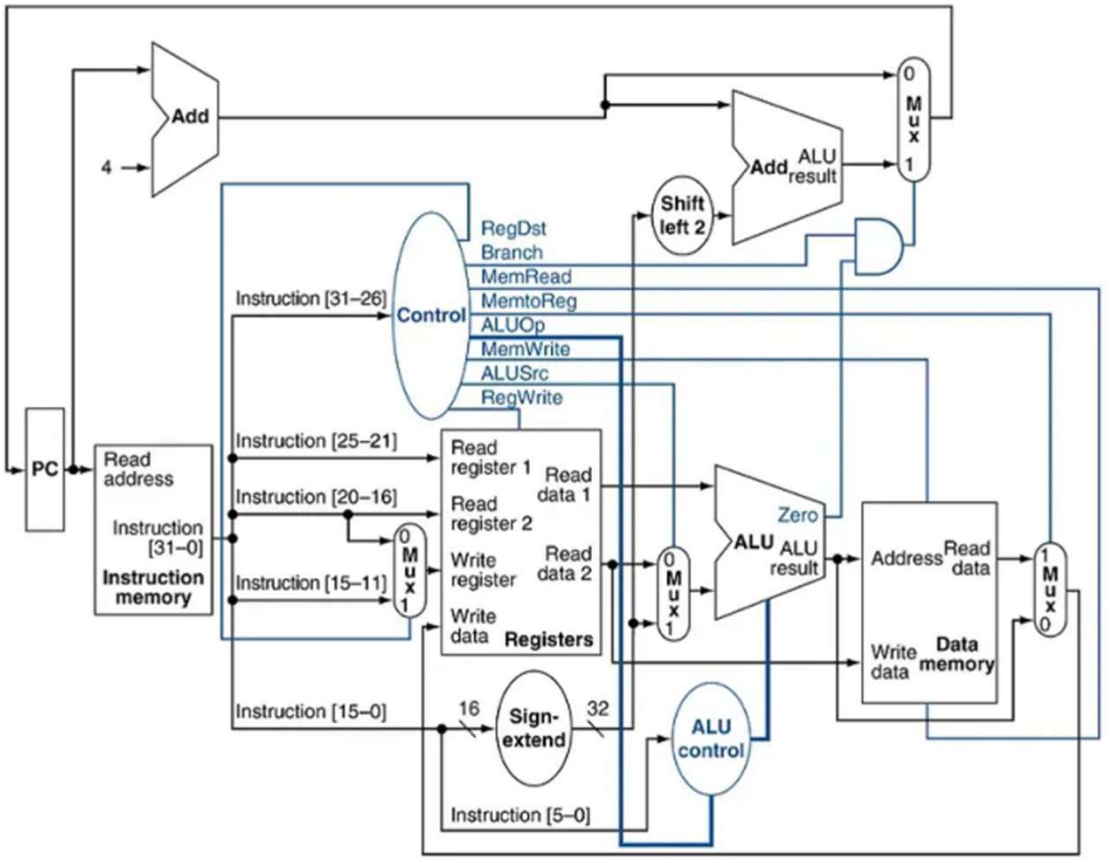
RISC-V Micro Processor

Block Diagram

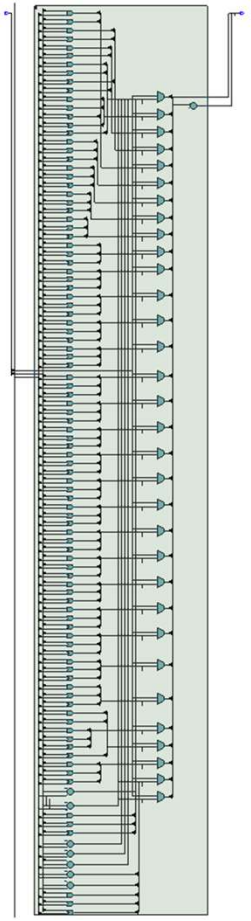


Implemented instructions

	R Type	I Type	S Type
Use:	Register to Register	Immediate and loading ops	Storing operations
	Anything I can do in an ALU		
Code (ASM)	add x3, x1, x2	addi x5, x1, 10	sw x2, 4(x1)

Coming soon

B Type	U Type	J Type	M Extension
Conditional branching	Build large constants	Function calls / jumps	Hardware, multiply, divide, etc.
Uses two registers and an *immediate offset* to decide if the PC jumps	Loads a 20-bit immediate into upper bits of a register	Adds an immediate offset to PC and stores return address	Mostly can be done through the ALU
		Next to be implemented	



ALU

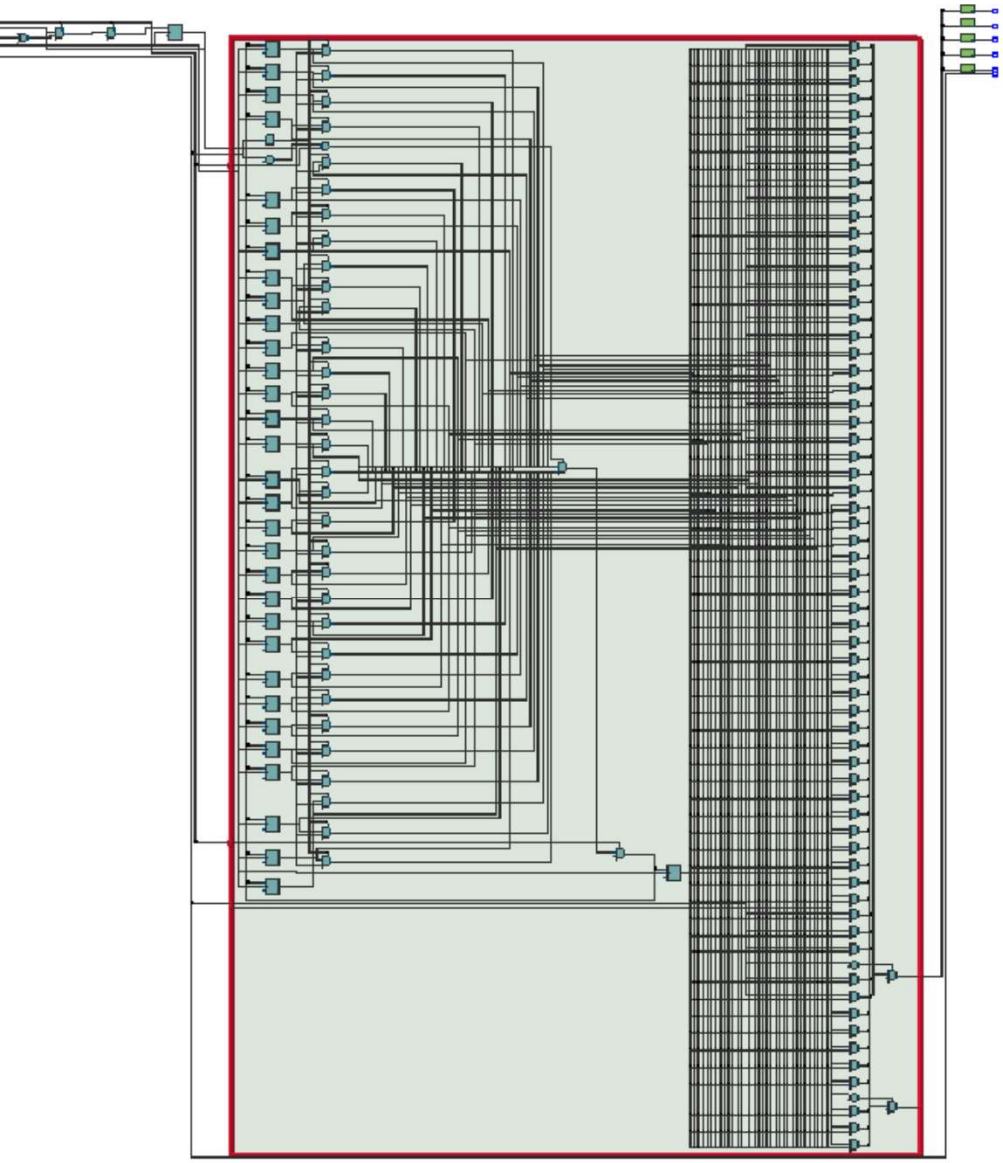
- It’s a little absurd, the RTL

```
module Arithmetic_Unit (
    input [31:0] A,
    input [31:0] B,
    input [3:0] ALU_Control,
    output reg [31:0] ALU_Result,
    output Zero
);
    localparam ADD  = 4'b0000;
    localparam SUB  = 4'b0001;
    localparam AND  = 4'b0010;
    localparam OR   = 4'b0011;
    localparam XOR  = 4'b0100;
    localparam SLT  = 4'b0101; // Set less than
    localparam SLL  = 4'b0110; // Shift left logical
    localparam SRL  = 4'b0111; // Shift right logical
    localparam SRA  = 4'b1000; // Shift right arithmetic
    localparam STLU = 4'b1001; // Set Less Than Unsigned
    always @(*) begin
        case(ALU_Control)
            ADD:  ALU_Result = A + B;
            SUB:  ALU_Result = A - B;
            AND:  ALU_Result = A & B;
            OR:   ALU_Result = A | B;
            XOR:  ALU_Result = A ^ B;
            SLT:  ALU_Result = ($signed(A) < $signed(B)) ? 32'd1 : 32'd0;
            SLL:  ALU_Result = A << B[4:0];
            SRL:  ALU_Result = A >> B[4:0];
            SRA:  ALU_Result = $signed(A) >>> B[4:0];
            STLU:  ALU_Result = (A < B) ? 1 : 0;
            default: ALU_Result = 32'd0;
        endcase
    end

    assign Zero = (ALU_Result == 32'd0);
endmodule
```

Code

- The ALU code seems to me to be easier to understand than the full RTL Viewer
- This is where all R type instructions are done.
- A few I type instructions like ADDI,
- Most of the M extensions need to be added in

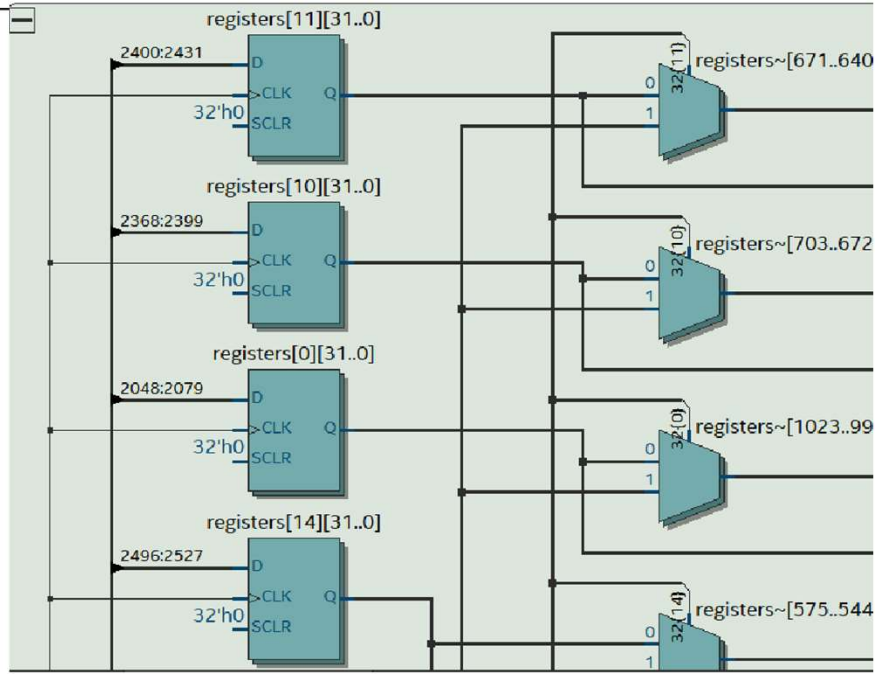


Registers

- All RegesterFile.v Does is create 32, 32 bit registers

RegisterFile.v

```
1  module RegisterFile (
2      input clk,
3      input reset,
4      input [4:0] rs1_addr,       // Read port 1 (5 bits = 32 regs)
5      input [4:0] rs2_addr,       // Read port 2
6      input [4:0] rd_addr,        // Write port
7      input [31:0] rd_data,       // Data to write
8      input reg_write,            // Write enable
9      output [31:0] rs1_data,     // Read data 1
10     output [31:0] rs2_data      // Read data 2
11 );
12
13     // Storage for 32 registers
14     reg [31:0] registers [0:31];
15
16     // Read ports: x0 always returns zero, others return stored value
17     assign rs1_data = (rs1_addr == 5'd0) ? 32'd0 : registers[rs1_addr];
18     assign rs2_data = (rs2_addr == 5'd0) ? 32'd0 : registers[rs2_addr];
19
20     // Write port: block writes to x0
21     always @(posedge clk) begin
22         if (reset) begin
23             integer i;
24             for (i = 0; i < 32; i = i + 1)
25                 registers[i] <= 32'd0;
26         end
27         else if (reg_write && rd_addr != 5'd0)
28             registers[rd_addr] <= rd_data;
29     end
30
31 endmodule
```



I and S types

- Primarily access the previously shown RegisterFile.v to read, write and perform quick actions

Program Counter and Branch Unit

```
`default_nettype none


module BranchUnit (
    input [31:0] PC,              // Current program counter
    input [31:0] rs1_data,        // Register rs1 value (for JALR)
    input [31:0] immediate,       // Immediate value from instruction

    output [31:0] branch_target,  // Branch target: PC + imm
    output [31:0] jump_target,    // JAL target: PC + imm
    output [31:0] jalr_target     // JALR target: (rs1 + imm) & ~1
);

    // Branch and JAL: PC + immediate
    // (Both use PC-relative addressing)
    assign branch_target = PC + immediate;
    assign jump_target   = PC + immediate;

    // JALR: (rs1 + immediate) with LSB cleared
    // LSB must be cleared to ensure alignment
    assign jalr_target = (rs1_data + immediate) & 32'hFFFFFFFE;

endmodule

`default_nettype wire
```

```
module ProgramCounter (
    input MainClock,
    input reset,                // Clear counter (renamed from ClearCounter)
    input PCWrite,              // Enable PC update
    input [1:0] PCSrc,          // Source select: 00=PC+4, 01=branch, 10=jump,
    input [31:0] branch_target,   // Branch target address
    input [31:0] jump_target,     // Jump target address (JAL)
    input [31:0] jalr_target,     // JALR target from ALU
    output reg [31:0] PC          // Program Counter (32-bit for RISC-V)
);

    // Internal signal for next PC value
    reg [31:0] next_pc;

    // Combinational logic for next PC calculation
    always @(*) begin
        case (PCSrc)
            2'b00:   next_pc = PC + 4;           // Normal: PC = PC + 4
            2'b01:   next_pc = branch_target;    // Branch taken
            2'b10:   next_pc = jump_target;      // JAL
            2'b11:   next_pc = jalr_target;      // JALR
            default: next_pc = PC + 4;
        endcase
    end
```

- This is the frame work for the J commands and allow for the storing of

Additional Notes

- The FSM design from lab 8 is being kept around. Same 4 phases of Fetch (stage 1), Decode (stage 2), Execute A (stage 3), Execute B (stage 4).
- I have a very poorly made main I didn’t feel like showing.