IEEE
CAIRO UNIVERSITY STUDENT BRANCH
IEEE
CUSB

Digital Workshop

(2023/2024)

(Final Project)

2024/2025
Ahmed Wael

IEEE

Single Cycle RV-32I Processor

Introduction

In this project, It is required to implement a 32-bit single-cycle microarchitecture RISC-V processor based on Harvard Architecture. The single- cycle microarchitecture executes an entire instruction in one cycle. In other words, instruction fetch, instruction decode, execute, write back, and program counter update occurs within a single clock cycle, Also, the program and data memory have their own address and data buses for communication with the processor Unit.

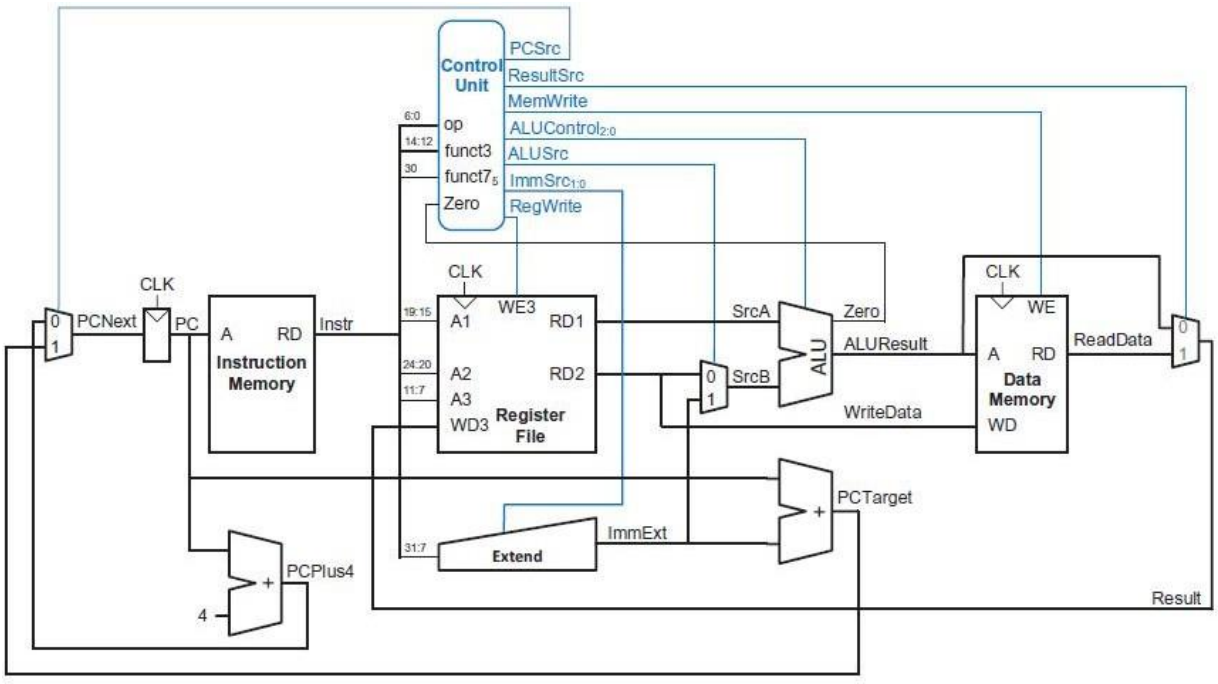
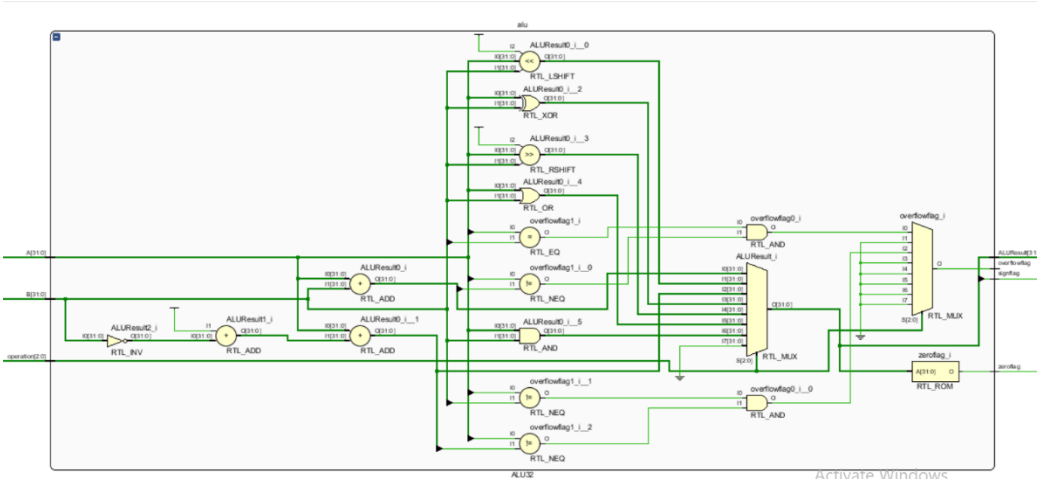


Figure 1: Complete single-cycle RISC-V processor

Main Modules

1. ALU

An Arithmetic/Logical Unit (ALU) combines a variety of mathematical and logical operations into a single unit. For example, a typical ALU might perform addition, subtraction, magnitude comparison, AND, and OR operations. ALU forms the heart of most computer systems. The 3-bit ALU Control signal specifies the operation. The ALU generates a **32-bit ALU Result, a Zero flag** that indicates whether ALU Result == 0, **and a sign flag** that indicates ALU result sign (ALU Result [31]). I also added an **optional overflow flag.**

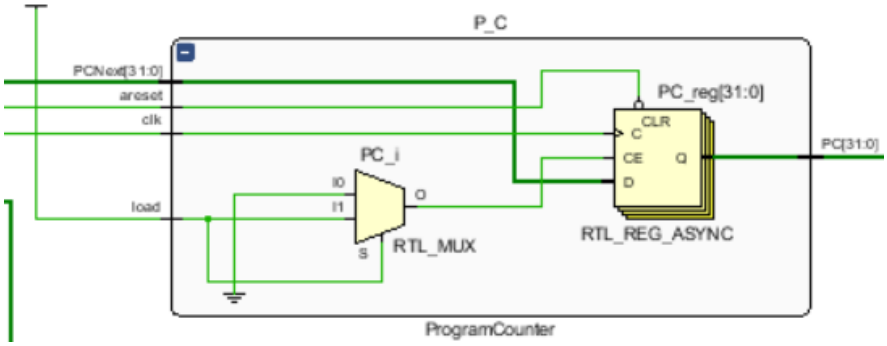


```
`timescale 1ns/1ps
module ALU32(
input [31:0]A,B,
input [2:0] operation,
output reg[31:0] ALUResult,
output reg zeroflag,
output reg signflag,
output reg overflowflag
    );
     reg [32:0]sum_result;
     reg[31:0]negB;
    always@(*)begin
    signflag=1'b0;
    overflowflag=1'b0;
    zeroflag=1'b0;
    case(operation)
    3'b000: begin
    sum_result=A+B;
    ALUResult=sum_result[31:0];
    if ((A[31] == B[31]) && (A[31] != ALUResult[31])) begin
                    overflowflag = 1'b1;
                end else begin
                    overflowflag = 1'b0;
                end
                end
    3'b001:ALUResult=A<<B;
    3'b010:begin
    negB = ~B + 1;
                sum_result = A + negB;
                ALUResult = sum_result[31:0];


                if ((A[31] != B[31]) && (A[31] != ALUResult[31])) begin
                    overflowflag = 1'b1;
                end else begin
                    overflowflag = 1'b0;
                end

            end


    3'b100:ALUResult=A^B;
    3'b101:ALUResult=A>>B;
    3'b110:ALUResult=A|B;
    3'b111:ALUResult=A&B;
    default:ALUResult=32'b0;
    endcase
     signflag=ALUResult[31];
    if(ALUResult==32'b0) zeroflag=1'b1;
    else zeroflag=1'b0;
    end

endmodule
```

2. Program Counter

2.1. Program Counter Register

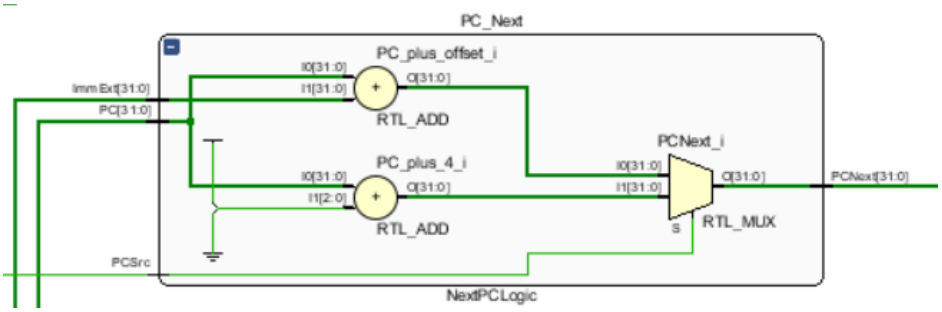
To fetch the instructions from the instruction memory, we need a **pointer** to keep track of the address of the current instruction for this task we use the program counter. The program counter is simply a **32-bit register** that has the address of the current instruction at its output and the address of the next instruction at its input. Firstly, you need to implement this register and then the logic that calculates the address of the next instruction. The program counter has four inputs a **32-bit word** which is the next address, the **clock** signal, **asynchronous reset**, and a **load** signal (always high except for the **HLT** instruction). And have one **32-bit output PC**.




```verilog
`timescale 1ns/1ps
module ProgramCounter (
    input clk,                  // Clock signal
    input areset,               // Asynchronous reset (active low)
    input load,                 // Load control signal
    input [31:0] PCNext,        // Next address input
    output reg [31:0] PC        // Current program counter output
);

always @(posedge clk or negedge areset) begin
    if (!areset) begin

        PC <= 32'b0;
    end
    else begin
        if (load) begin

            PC <= PCNext;
        end
       if(!load) begin
       PC<=PC;
       end
    end

end

endmodule
```

2.2 Next PC calculation logic

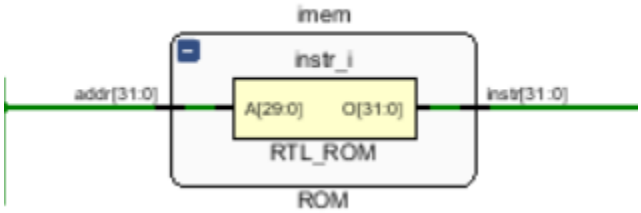
Now we need to implement the logic that calculates the address of the next instruction. Normally the address of the next instruction is **PC + 4**, but in the case of a **taken branch** the address of the next instruction is **PC + target**. The circuit that handles this consists of 3 elements, a **2x1 multiplexer** and **two binary adders** that have the current value of PC at one of its operands as you can see in figure one. This circuit has two inputs a **32-bit** number **ImmExt** that is coming from the **sign extend unit** (target address for branch instructions) and **PCSrc** signal to select the right source for next PC. And has one **32-bit** output which is the **next PC**.




```
`timescale 1ns/1ps
module NextPCLogic (
    input [31:0] PC,         // Current program counter
    input [31:0] ImmExt,     // Sign-extended immediate (branch target offset)
    input PCSrc,             // Control signal for branch selection
    output [31:0] PCNext     // Next program counter value
);

// Internal signals
wire [31:0] PC_plus_4;
wire [31:0] PC_plus_offset;

// Calculate both possible next addresses
assign PC_plus_4 = PC + 32'd4;
assign PC_plus_offset = PC + ImmExt;

// Select appropriate next PC based on PCSrc
assign PCNext = (PCSrc) ? PC_plus_offset : PC_plus_4;

endmodule
```

3. Instruction memory

- The instruction memory has a single read port.
- It takes a 32-bit instruction address input, A, and reads the 32-bit data (i.e., instruction) from that address onto the read data output, RD.
- The PC is simply connected to the address input of the instruction memory.
- The instruction memory reads out, or fetches, the 32-bit instruction, labeled Instr.
- Our instruction memory is a Read Only Memory (ROM) that holds the program that your CPU will execute.
- The ROM Memory has width = 32 bits and depth = 64 entries.
- Instructions are read asynchronously.




```
`timescale 1ns/1ps
module ROM(
    input [31:0] addr,
    output  [31:0] instr
);
    reg [31:0] memory [0:63];
    initial begin
        $readmemh("program.txt", memory);
    end

        assign instr = memory[addr[31:2]];

endmodule
```

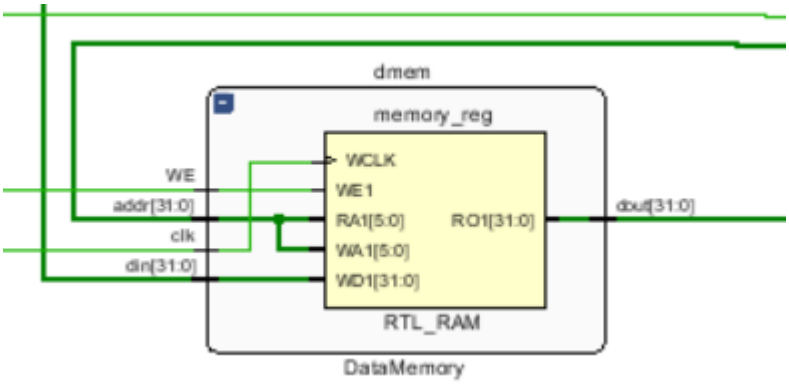
4. Register File

- The Register File contains the 32-bit registers.
- The register file has two read output ports (RD1 and RD2) and a single input write port (WD3), RD1 and RD2 are read with no respect to the clock edge.
- The register file is read asynchronously and written synchronously at the rising edge of the clock.
- The register file supports simultaneous read and writes. The register file has width = 32 bits and depth = 32 entries supports simultaneous read and writes.
- The register file has active low asynchronous reset signal.
- A1 is the register address from which the data are read through the output port RD1. Whereas A2 is corresponding to the register address of output port RD2.

```
`timescale 1ns/1ps
module RegFile(
    input wire clk,
    input wire RegWrite,
    input wire reset,
    input wire [4:0] a1, a2, a3,     // Read and write addresses
    input wire [31:0] wd3,           // Write data
    output wire [31:0] rd1, rd2      // Read data outputs
);

    // 32 general-purpose 32-bit registers
    reg [31:0] Regs[0:31];

    // Register read is combinational
    assign rd1 = Regs[a1];
    assign rd2 = Regs[a2];

    // Register write + synchronous reset (unrolled)
    always @(posedge clk) begin
        if (!reset) begin
            Regs[0]  <= 32'b0;
            Regs[1]  <= 32'b0;
            Regs[2]  <= 32'b0;
            Regs[3]  <= 32'b0;
            Regs[4]  <= 32'b0;
            Regs[5]  <= 32'b0;
            Regs[6]  <= 32'b0;
            Regs[7]  <= 32'b0;
            Regs[8]  <= 32'b0;
            Regs[9]  <= 32'b0;
            Regs[10] <= 32'b0;
            Regs[11] <= 32'b0;
            Regs[12] <= 32'b0;
            Regs[13] <= 32'b0;
            Regs[14] <= 32'b0;
            Regs[15] <= 32'b0;
            Regs[16] <= 32'b0;
            Regs[17] <= 32'b0;
            Regs[18] <= 32'b0;
            Regs[19] <= 32'b0;
            Regs[20] <= 32'b0;
            Regs[21] <= 32'b0;
            Regs[22] <= 32'b0;
            Regs[23] <= 32'b0;
            Regs[24] <= 32'b0;
            Regs[25] <= 32'b0;
            Regs[26] <= 32'b0;
            Regs[27] <= 32'b0;
            Regs[28] <= 32'b0;
            Regs[29] <= 32'b0;
            Regs[30] <= 32'b0;
            Regs[31] <= 32'b0;
        end else if (RegWrite && a3 != 5'd0) begin
            Regs[a3] <= wd3;
        end
    end
endmodule
```

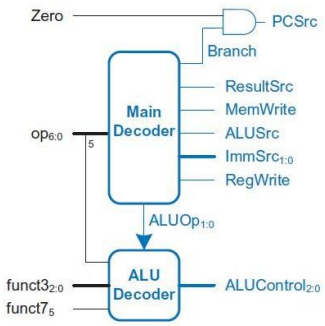
5. Data Memory

- It has a single read/write port.
- If its write enable, WE, is asserted, then it writes data WD into address A on the rising edge of the clock.
- It reads are asynchronous while writes are synchronous to the rising edge of the “clk” signal.
- The Word width of the data memory is 32-bits to match the datapath width. The data memory contains 64 entries.
- RD is read with no respect to the clock edge.
- A is the memory address from which the data are read through the output port RD.

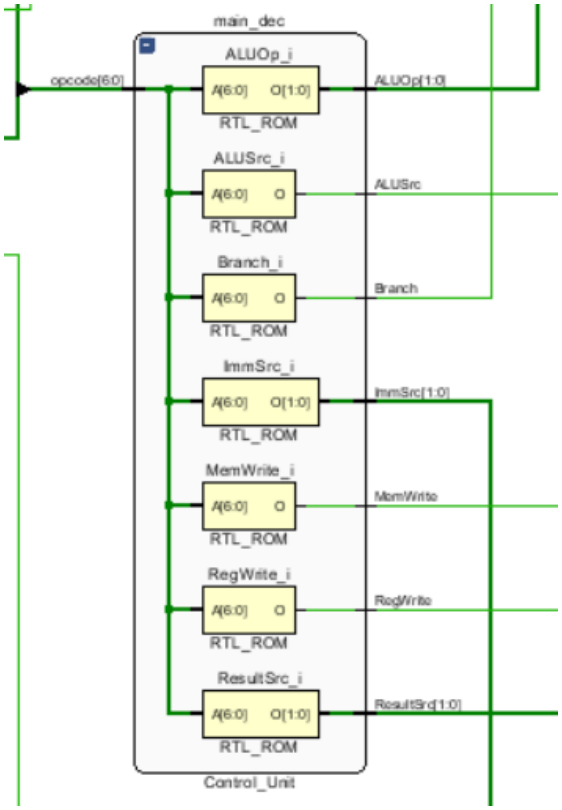



```systemverilog
`timescale 1ns/ 1ps
module DataMemory(
    input wire clk,
    input wire WE,                  // Write Enable
    input wire [31:0] addr,
    input wire [31:0] din,          // Data input (write)
    output wire [31:0] dout          // Data output (read)
);

    // 64 words of 32-bit memory
    reg [31:0] memory [0:63];



    // Write operation (synchronous)
    always @(posedge clk)
    begin

         if (WE == 1) begin
            memory[addr[31:2]] <= din;
        end
    end
    //Read Operation (asynchronous)
    assign dout =  memory[addr[31:2]];

    endmodule
```

6. Control Unit

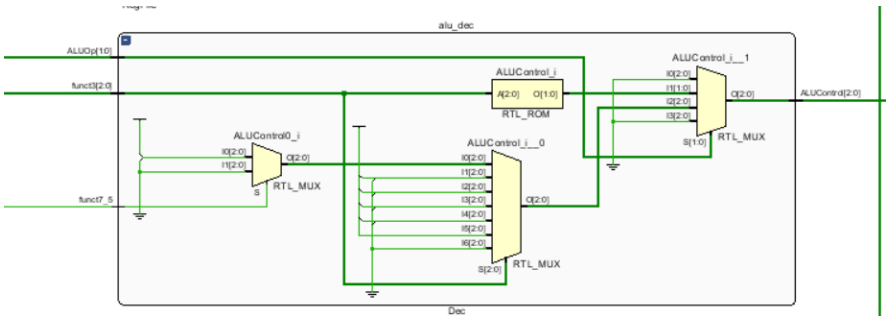


The control unit computes the control signals based on the opcode and funct3, funct7 fields ofthe instruction, Instr[14:12] and Instr[30] respectively. Most of the control information comes from the opcode, but R-type instructions and I-type instructions also use the funct3 and funct7 fields to determine the ALU operation. Thus, I will simplify our design by factoring the control unit into two blocks of combinational logic




```
module Control_Unit
( input  [6:0] opcode,
    output reg  RegWrite,
    output reg [1:0] ImmSrc,
    output reg ALUSrc,
    output reg MemWrite,
    output reg [1:0] ResultSrc,
    output reg  Branch,
    output reg [1:0] ALUOp
);

always @(*) begin
    case (opcode)
        7'b0000011: begin // load
            RegWrite   = 1;
            ImmSrc     = 2'b00;
            ALUSrc     = 1;
            MemWrite   = 0;
            ResultSrc  = 2'b01;
            Branch     = 0;
            ALUOp      = 2'b00;
        end
        7'b0100011: begin // store
            RegWrite   = 0;
            ImmSrc     = 2'b01;
            ALUSrc     = 1;
            MemWrite   = 1;
            ResultSrc  = 2'bxx;
            Branch     = 0;
            ALUOp      = 2'b00;
        end
        7'b0110011: begin // R-type
            RegWrite   = 1;
            ImmSrc     = 2'bxx;
            ALUSrc     = 0;
            MemWrite   = 0;
            ResultSrc  = 2'b00;
            Branch     = 0;
            ALUOp      = 2'b10;
        end
        7'b0010011: begin // I-type (immediate ALU ops)
            RegWrite   = 1;
            ImmSrc     = 2'b00;
            ALUSrc     = 1;
            MemWrite   = 0;
            ResultSrc  = 2'b00;
            Branch     = 0;
            ALUOp      = 2'b10;
        end
        7'b1100011: begin // branch
            RegWrite   = 0;
            ImmSrc     = 2'b10;
            ALUSrc     = 0;
            MemWrite   = 0;
            ResultSrc  = 2'bxx;
            Branch     = 1;
            ALUOp      = 2'b01;
        end
        default: begin
            RegWrite   = 0;
            ImmSrc     = 2'b00;
            ALUSrc     = 0;
            MemWrite   = 0;
            ResultSrc  = 2'b00;
            Branch     = 0;
            ALUOp      = 2'b00;
        end
    endcase
end

endmodule
```

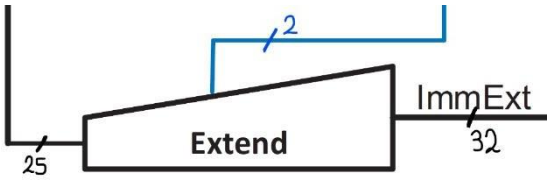
6.1 ALU Decoder




```
`timescale 1ns/1ps
module Dec (
    input  [1:0] ALUOp,
    input  [2:0] funct3,
    input        funct7_5,
    output reg [2:0] ALUControl
);

always @(*) begin
    case (ALUOp)
        2'b00: ALUControl = 3'b000; // load/store ? ADD
        2'b01: begin                // branch
            case (funct3)
                3'b000: ALUControl = 3'b010; // BEQ ? SUB
                3'b001: ALUControl = 3'b010; // BNE ? SUB
                3'b100: ALUControl = 3'b010; // BLT ? SUB
                default: ALUControl = 3'b000;
            endcase
        end
        2'b10: begin                // R-type or I-type ALU
            case (funct3)
                3'b000: ALUControl = funct7_5 ? 3'b010 : 3'b000; // SUB or
ADD
                3'b001: ALUControl = 3'b001; // SHL
                3'b100: ALUControl = 3'b100; // XOR
                3'b101: ALUControl = 3'b101; // SHR
                3'b110: ALUControl = 3'b110; // OR
                3'b111: ALUControl = 3'b111; // AND
                default: ALUControl = 3'b000;
            endcase
        end
        default: ALUControl = 3'b000;
    endcase
    end
endmodule
```

Small modules

1. Sign extend



Sign extension simply copies the sign bit (most significant bit) of a short input (16 bits) into all the upper bits of the longer output (32 bits).

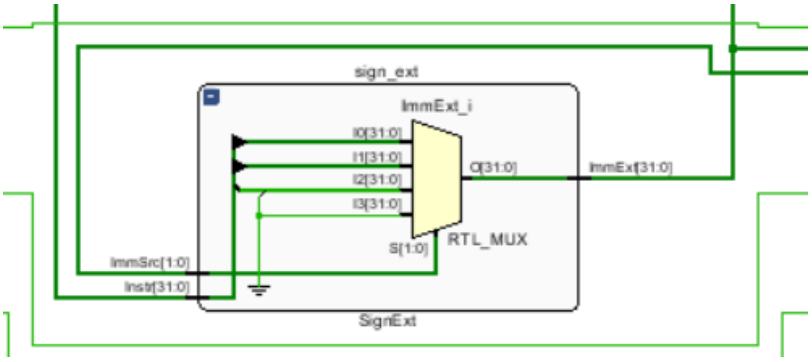
→example

- Inst [15:0] = **0**000 0000 0011 1101

  ImmExt = **0000 0000 0000 0000** 0000 0000 0011 1101

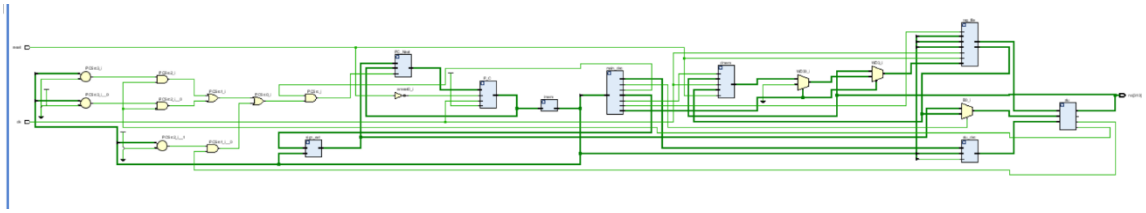
- Inst [15:0] = **1**000 0000 0011 1101

  ImmExt = **1111 1111 1111 1111** 1000 0000 0011 1101




```systemverilog
`timescale 1ns / 1ps
module SignExt (
    input [31:0] Instr,       // Full 32-bit instruction
    input [1:0] ImmSrc,       // Control signal from Table 7.1
    output reg [31:0] ImmExt  // 32-bit sign-extended immediate
);

wire [11:0] immI = Instr[31:20];
wire [11:0] immS = {Instr[31:25], Instr[11:7]};
wire [12:0] immB = {Instr[31], Instr[7], Instr[30:25], Instr[11:8], 1'b0};

always @(*) begin
    case (ImmSrc)
        2'b00: ImmExt = {{20{immI[11]}}, immI};
        2'b01: ImmExt = {{20{immS[11]}}, immS};
        2'b10: ImmExt = {{19{immB[12]}}, immB};
        default: ImmExt = 32'b0;
    endcase
end
endmodule
```

Final Processor top module and full schematic



```
`timescale 1ns/1ps

module RISC_V_Processor(
    input clk,
    input reset,
    output[31:0]no
);

    //Wires
    wire [31:0] PC;
    wire PCSrc;
    wire [31:0] DataMemory_out;
    wire [31:0] ImmExt;
    wire [31:0] RD1, RD2, WD3;
    wire [31:0] ALUResult;
    wire [31:0] instruction_out;
    wire Zero, Sign;

    // Control signals
    wire RegWrite,ALUSrc, MemWrite,Branch;
    wire [1:0]ResultSrc;
    wire [1:0] ImmSrc, ALUOp;
    wire [2:0] ALUControl;
    wire[31:0]PCNext_from_PCNext_Calc;

    // ControlUnit
   Control_Unit main_dec (
        .opcode(instruction_out[6:0]),
        .RegWrite(RegWrite),
        .ImmSrc(ImmSrc),
        .ALUSrc(ALUSrc),
        .MemWrite(MemWrite),
        .ResultSrc(ResultSrc),
        .Branch(Branch),
        .ALUOp(ALUOp)
    );
        // Sign Extend
          SignExt sign_ext(
              .Instr(instruction_out),
              .ImmSrc(ImmSrc),
              .ImmExt(ImmExt)
          );

         //PC Logic
 NextPCLogic PC_Next (
            .PC(PC),
            .ImmExt(ImmExt),
            .PCSrc(PCSrc),
            .PCNext(PCNext_from_PCNext_Calc)
        );

        ProgramCounter P_C (
            .clk(clk),
            .areset(~reset),
            .load(1'b1),
            .PCNext(PCNext_from_PCNext_Calc),
            .PC(PC)
        );

   // Instruction Memory (ROM)
ROM imem(
    .addr(PC),
    .instr(instruction_out)
);

    // Register File
    RegFile reg_file(
        .clk(clk),
        .reset(reset),
        .RegWrite(RegWrite),
        .a1(instruction_out[19:15]),
        .a2(instruction_out[24:20]),
        .a3(instruction_out[11:7]),
        .wd3(WD3),
        .rd1(RD1),
        .rd2(RD2)
    );

    // ALU Decoder
    Dec alu_dec (
        .ALUOp(ALUOp),
        .funct3(instruction_out[14:12]),
        .funct7_5(instruction_out[30]),
        .ALUControl(ALUControl)
    );


    //Additional overflow flag

    wire overflowflag;

    // ALU
    ALU32 alu(
        .A(RD1),
        .B(ALUSrc ? ImmExt : RD2),
        .operation(ALUControl),
        .ALUResult(ALUResult),
        .zeroflag(Zero),
        .signflag(Sign),
        .overflowflag(overflowflag)
    );

     //Data Memory
   DataMemory dmem(
     .clk(clk),
          .addr(ALUResult),
          .din(RD2),
          .WE(MemWrite),
          .dout(DataMemory_out)
);


   //PCSrc Logic

    assign PCSrc = Branch && (
    (instruction_out[14:12] == 3'b000 && Zero)   ||
    (instruction_out[14:12] == 3'b001 && ~Zero)  ||
    (instruction_out[14:12] == 3'b100 && Sign)
);


   assign WD3 = (ResultSrc == 2'b00) ? ALUResult :
            (ResultSrc == 2'b01) ? DataMemory_out :
             32'b0;

    // For synthesis prop
    assign no=ALUResult;
```

Final results and simulations

FIBONACCI series

Machine code

```
00004033
00000093
00100113
00100193
00100213
00000293
00a00313
00000393
00418c63
00110133
404181b3
00229393
0023a023
00420a63
002080b3
004181b3
00229393
0013a023
00128293
fc62cae3
00000000
```

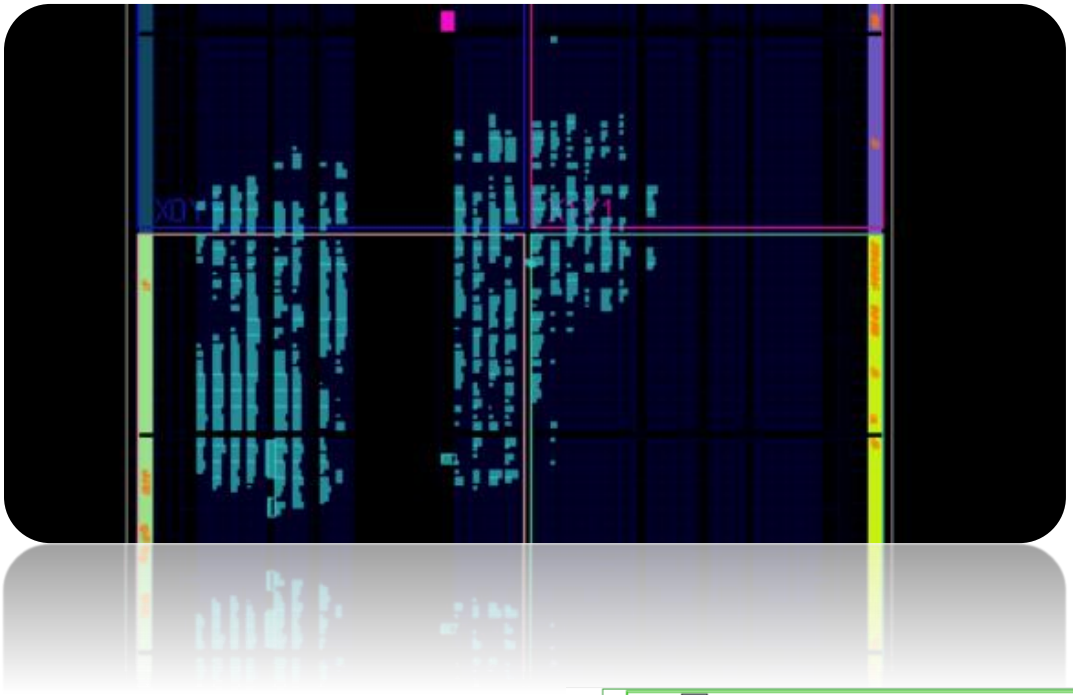
C-code

```
#include <iostream>
using namespace std;
int main()
  { int x = 0;
   int y = 1;
   int sel = 1;
   for (int i = 0; i < 10; i++)
   {
      if (sel == 1)
        { x = x + y;
         sel = 2;
       cout << x << endl;  }
        else
       {  y = x + y;
         sel = 1;
      cout << y << endl;}
  }
}
```

Assembly code

```
0:  xor  x0,x0,x0      # reference register, always = 0
4:  addi x1,x0,0       # data register (containing one of the last two values in FIBONACCI series)
8:  addi x2,x0,1       # data register (containing the other value)
12: addi x3,x0,1       # selector (select the oldest register from R1 and R2 that had been refreshed)
16: addi x4,x0,1       # always constant and = 1
20: addi x5,x0,0       # loop counter
24: addi x6,x0,10      # total number of loops
28: addi x7,x0,0       # address of the data memory that will receive the latest evaluated result at
loop:
32: beq  x3,x4,eq      # you can understand the code well from the attached c-code
36: add  x2,x2,x1
40: sub  x3,x3,x4
44: slli x7,x5,2
48: sw   x2,0(x7)
52: beq  x4,x4,endloop
eq:
56: add  x1,x1,x2
60: add  x3,x3,x4
64: slli x7,x5,2
68: sw   x1,0(x7)
endloop:
72: addi x5,x5,1
76: blt  x5,x6,loop
80: halt               # you can make it to prevent PC from increasing after the code had been finished
                       # But it is optional
```

Simulation Result



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.169 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.8°C**
Thermal Margin:	74.2°C (14.7 W)
Effective ϑJA:	5.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

On-Chip Power