SAMPLE RISC-V SINGLE STAGE PROCESSOR - TANU 1.0

➢ Objective:

Design a single stage RISC-V Processor and do functional verification for the at least two instructions in behavioral simulation.

➢ Design Approach:

Refer the chapter 4 from the reference.

Consider the simple architecture / datapath for the processor -

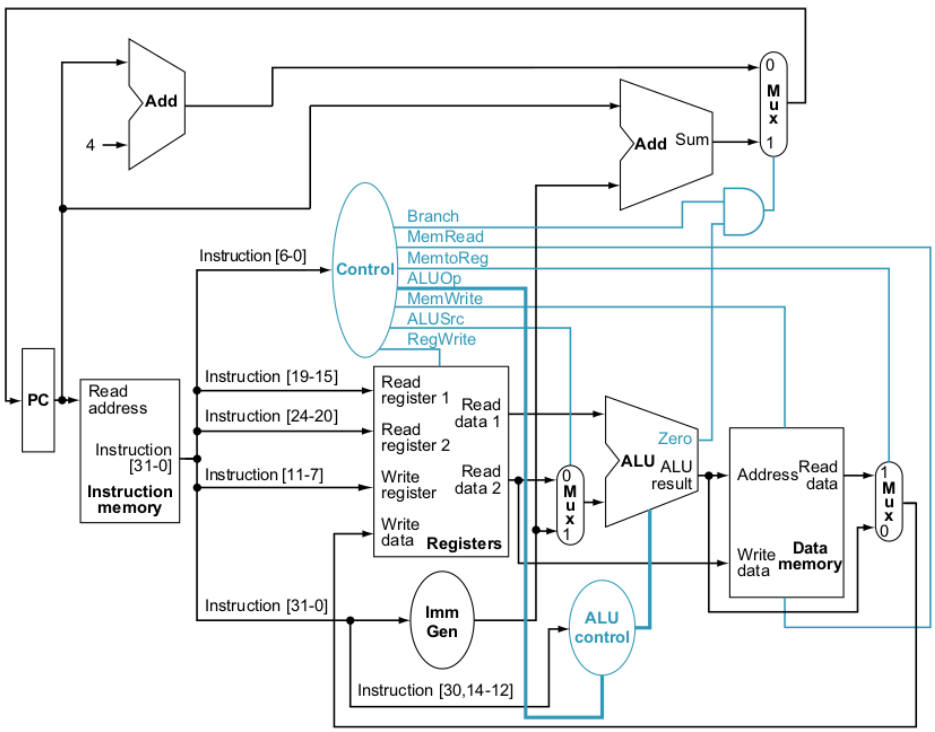


Figure 1: The simple datapath for the core RISC-V architecture with the control unit

For the design of Immediate Generator, refer the use of the Immediate Bits in the Instruction Formatting in the below table:

Name (Bit position)	Fields					
	31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

Figure 2: The four Instruction Classes (arithmetic, load, store and Conditional Brach) use four different instruction formats.

Type of instruction based on opcode and Output pertaining the type of instruction:

Input or output	Signal name	R-format	lw	sw	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Figure 3: Control function for simple single cycle implementation.

ALU Implementation -

The below table shows the operation performed by ALU for different combinations of its control inputs.

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

Implementation of ALU Control i.e., it decides the control signals to be sent to the ALU Depending upon the instruction fields it decoded.

Use the table below that uses the instruction fields to generate the control signals. Refer the block diagram representation for the Single Cycle CPU Implementation.

ALUOp		Funct7 field							Funct3 field			
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

➢ Simulation Verification:

As we are using Vivado for designing the CPU, lets go through the design verification steps to be perfomed for the Xilinx Simulator (Xsim) using CLI.

Simulation Run Guidelines

Step 1: Open Terminal and Souce Executable

Open the terminal and source the settings64.sh file to be able to run the Xsim simulator via CLI. Sourcing the settings64.sh file in the vivado project directory will make the environment variables visible to the OS so that the libraries, compilers actual position can be known in order to execute the desired commands over them. This is handled intenally, when the same is tried by opening the GUI Icon.

Step 2: Compile all the scirpts in the Projects

Compile all the project files including the design sources and the simulation sources and resolve any errors or issues in compilation.

Format: **xvlog <file_path>**
xvlog - compile scripts with extension .v, .sv, .vh, .svh
xvhdl - compile scripts with extension .vhdl, .vhd

Step 3: Elaborate the Testbench

To create the simulation snapshot, which can be run in simulation window, use the below command. If theres any bug/syntax issue that can be captured in the terminal screen log.

xelab sim_entity -debug all -s top_sim

sim_entity - Name of simulation module entity name in the testbench script
top_sim - Name of snapshot to be saved.

Step 4: Run the simulation window

To see the waveforms, we need to run the stored snapshot in the simulator. For running the simulation -

xsim top_sim - gui

top_sim – Name of the stored snapshot while elaborating the testbench

A) Initialization of Instruction Memory

The instructions to be executed are stored in the Instruction Memory. Lets update the content of the Instruction Memory to keep the desired instructions in it for verifying the funtionality.

In order to verify the different instructions, we will keep the instruction types R, I, L, S, SB in the memory.

R Type

1. add x13, x16, x25
2. sub x5, x8, x3
3. and x1, x2, x3
4. or x4, x3, x5

I Type

1. addi x22, x21, 3
2. ori x9, x8, 1

L Type

1. 1w x8, 15(x5)
2. 1w x9, 3(x3)

SB Type

beq x9, x9, 12

The instruction memory is updated with the same instruction format as mentioned in Instruction Formatting Table above.

B) Initialize the Register File

Lets initialize the contents of Reg Fiel with random decimal values.

- **Results:**

The below are the contents of the Register File and the Instruction Memory of the RISC CPU.

Contents of Instruction Memory:

```
begin

    //  R -tType
    IMemory[0]  =   32'b0000000_00000_00000_000_00000_0000000;      //  NOP
    IMemory[4]  =   32'b0000000_11001_10000_000_01101_0110011;      //  add x13, x16, x25
    IMemory[8]  =   32'b0100000_00011_01000_000_00101_0110011;      //  sub x5, x8, x3
    IMemory[12] =   32'b0000000_00011_00010_111_00001_0110011;      //  and x1, x2, x3
    IMemory[16] =   32'b0000000_00101_00011_110_00100_0110011;      //  or x4, x3, x5

    //  I - type
    IMemory[20] =   32'b000000000011_10101_000_10110_0010011;       //  addi x22, x21, 3
    IMemory[24] =   32'b000000000001_01000_110_01001_0010011;       //  ori x9, x8, 1

    //  L - type
    IMemory[28] =   32'b000000001111_00101_010_01000_0000011;       //  1w x8, 15(x5)
    IMemory[32] =   32'b000000000011_00011_010_01001_0000011;       //  1w x9, 3(x3)

    //  S-type
    IMemory[36] =   32'b0000000_01111_00101_010_01100_0100011;      //  sw x15, 12(x5)
    IMemory[40] =   32'b0000000_01110_00110_010_01010_0100011;      //  sw x14. 10(x6)

    //  SB-type
    IMemory[44] =   32'h00948663;       //  beq x9, x9, 12

end
```

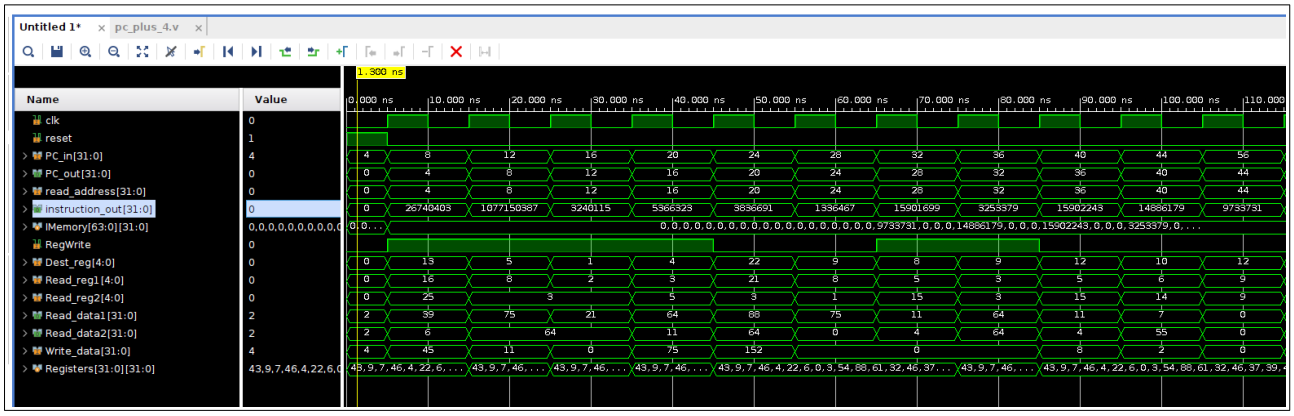
Contents of Register File:

```
//  Initialize the Reg File with Random Decimal Values
initial begin

    Registers[0]    =   2;
    Registers[1]    =   4;
    Registers[2]    =   21;
    Registers[3]    =   64;
    Registers[4]    =   33;
    Registers[5]    =   23;
    Registers[6]    =   7;
    Registers[7]    =   45;
    Registers[8]    =   75;
    Registers[9]    =   4;
    Registers[10]   =   3;
    Registers[11]   =   9;
    Registers[12]   =   5;
    Registers[13]   =   75;
    Registers[14]   =   55;
    Registers[15]   =   4;
    Registers[16]   =   39;
    Registers[17]   =   37;
    Registers[18]   =   46;
    Registers[19]   =   32;
    Registers[20]   =   61;
    Registers[21]   =   88;
    Registers[22]   =   54;
    Registers[23]   =   3;
    Registers[24]   =   0;
    Registers[25]   =   6;
    Registers[26]   =   22;
    Registers[27]   =   4;
    Registers[28]   =   46;
    Registers[29]   =   7;
    Registers[30]   =   9;
    Registers[31]   =   43;

end
```

Simulation Results for Instruction Execution:



➢ Reference:

1. Computer Organization and Design – RISCv Edition – Hennessy and Patterson – 2nded
2. Designing a RISC-V Single Cycle Processor: Step-by-Step Tutorial - Youtube