DEPARTMENT OF ELECTRONIC AND TELECOMMUNICATION ENGINEERING

UNIVERSITY OF MORATUWA

EN3021 - DIGITAL SYSTEMS DESIGN

Project Report

32-Bit Non-Pipelined Single-Cycle Processor based on RISC-V ISA

N.W.P.R.A. Perera 200462U

This report is submitted as a partial fulfillment for the module EN3021: Digital Systems Design.
Department of Electronic and Telecommunication Engineering, University of Moratuwa.

16th October 2023

Table of Contents

Requirements 4

Basic Implementation 5

RISC-V Instructions that need to be implemented 5

RISC-V Datapath that needs to be implemented 6

Modules Designed & Purpose of each Module 7

Test Bench Module (tb_top.sv) 7

Processor Module (RISC_V.sv) 7

ALU (alu.sv) 7

Data Memory (datamemory.sv) 8

Instruction Memory (instructionmemory.sv) 8

Register File (RegFile.sv) 8

Immediate Generator (imm_Gen.sv) 8

Controller (Controller.sv) 9

ALU Controller (ALUController.sv) 10

RISCV Data Path (Datapath.sv) 10

Data Extractor (data_extract.sv) 10

Adder (adder.sv) 10

2 input Multiplexer (mux2.sv) 11

D flip-flop (flopr.sv) 11

Testbench Simulations 12

Test Program 1 (R-Type Computational / Register-Register) 13

Test Program 2 (I-Type Computational / Short Immediate Operands) 14

Test Program 3 (S-Type Memory Access / Stores) 15

Test Program 4 (I-Type Memory Access / Loads) 16

Test Program 5 (SB-Type Control Transfer / Conditional Branches) 17

Test Program 6 (Adding 2 numbers and saving the result to memory) 19

Test Program 7 (Program that reads the value of n from memory and computes the sum of numbers from 1 to n and stores the sum in memory) 20

Resource Utilization 21

References 23

Appendix A: System Verilog Codes of Modules 24

Test Bench Module (tb_top.sv) 24

Processor Module (RISC_V.sv) 24
ALU (alu.sv) 25
Data Memory (datamemory.sv) 26
Instruction Memory (instructionmemory.sv) 27
Register File (RegFile.sv) 29
Immediate Generator (imm_Gen.sv) 29
Controller (Controller.sv) 30
ALU Controller (ALUController.sv) 31
RISCV Data Path (Datapath.sv) 31
Data Extractor (data_extract.sv) 33
Adder (adder.sv) 34
Adder 32bit (adder_32.sv) 34
2 input Multiplexer (mux2.sv) 35
D flip-flop (flopr.sv) 35
Appendix B: Gate Level Schematics of Modules 36
Test Bench Module (tb_top.sv) 36
Processor Module (RISC_V.sv) 36
ALU (alu.sv) 37
Data Memory (datamemory.sv) 37
Instruction Memory (instructionmemory.sv) 38
Register File (RegFile.sv) 38
Immediate Generator (imm_Gen.sv) 39
Controller (Controller.sv) 39
ALU Controller (ALUController.sv) 40
RISCV Data Path (Datapath.sv) 40
Data Extractor (data_extract.sv) 41
Adder (adder.sv) 41
Adder 32bit (adder_32.sv) 42
2 input Multiplexer (mux2.sv) 42
D flip-flop (flopr.sv) 43

Requirements

Processor design and implementation on a FPGA: Project with 3 stages and Stage 1 is the individual project.

Stage 1 (Individual): 32-bit non-pipelined RISC-V processor using Microprogramming with 3 bus structure. This is RV32I implementation.

You need to implement the following classes of instructions:

1. All computational instructions covered by instruction types R and I.
2. All memory access instructions (load and store): I and S type instructions
3. All Control Flow instructions: SB type

Once you complete the basic implementation, you need to add support to 2 new instructions.

a) MEMCOPY - Copies an array of size N from one location to another. N > 1. Determine the max N that you can use for this instruction.
b) MUL - Unsigned Multiplication (Note that RV32I does not include multiplication instructions). Identify the limits for operands of this instruction.

You need to submit a report which includes resource utilization and synthesizable RTL files.

Basic Implementation

RISC-V Instructions that need to be implemented

The processor should be capable of supporting the following instructions.

	imm[31:12]				rd	0110111	LUI
	imm[31:12]				rd	0010111	AUIPC
	imm[20|10:1|11|19:12]				rd	1101111	JAL
	imm[11:0]		rs1	000	rd	1100111	JALR
SB-Type Control Transfer	imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
	imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
	imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
	imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
	imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
	imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
I-Type Memory Access	imm[11:0]		rs1	000	rd	0000011	LB
	imm[11:0]		rs1	001	rd	0000011	LH
	imm[11:0]		rs1	010	rd	0000011	LW
	imm[11:0]		rs1	100	rd	0000011	LBU
	imm[11:0]		rs1	101	rd	0000011	LHU
S-Type Memory Access	imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
	imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
	imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
I-Type Computational	imm[11:0]		rs1	000	rd	0010011	ADDI
	imm[11:0]		rs1	010	rd	0010011	SLTI
	imm[11:0]		rs1	011	rd	0010011	SLTIU
	imm[11:0]		rs1	100	rd	0010011	XORI
	imm[11:0]		rs1	110	rd	0010011	ORI
	imm[11:0]		rs1	111	rd	0010011	ANDI

0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

I-Type Computational

R-Type Computational

RV32I Computational Instructions

Instruction	Format	Meaning
`add rd, rs1, rs2`	R	Add registers
`sub rd, rs1, rs2`	R	Subtract registers
`sll rd, rs1, rs2`	R	Shift left logical by register
`srl rd, rs1, rs2`	R	Shift right logical by register
`sra rd, rs1, rs2`	R	Shift right arithmetic by register
`and rd, rs1, rs2`	R	Bitwise AND with register
`or rd, rs1, rs2`	R	Bitwise OR with register
`xor rd, rs1, rs2`	R	Bitwise XOR with register
`slt rd, rs1, rs2`	R	Set if less than register, 2's complement
`sltu rd, rs1, rs2`	R	Set if less than register, unsigned
`addi rd, rs1, imm[11:0]`	I	Add immediate
`slli rd, rs1, shamt[4:0]`	I	Shift left logical by immediate
`srli rd, rs1, shamt[4:0]`	I	Shift right logical by immediate
`srai rd, rs1, shamt[4:0]`	I	Shift right arithmetic by immediate
`andi rd, rs1, imm[11:0]`	I	Bitwise AND with immediate
`ori rd, rs1, imm[11:0]`	I	Bitwise OR with immediate
`xori rd, rs1, imm[11:0]`	I	Bitwise XOR with immediate
`slti rd, rs1, imm[11:0]`	I	Set if less than immediate, 2's complement
`sltiu rd, rs1, imm[11:0]`	I	Set if less than immediate, unsigned
`lui rd, imm[31:12]`	U	Load upper immediate
`auipc rd, imm[31:12]`	U	Add upper immediate to pc

RV32I Control Transfer Instructions

Instruction	Format	Meaning
`beq rs1, rs2, imm[12:1]`	SB	Branch if equal
`bne rs1, rs2, imm[12:1]`	SB	Branch if not equal
`blt rs1, rs2, imm[12:1]`	SB	Branch if less than, 2's complement
`bltu rs1, rs2, imm[12:1]`	SB	Branch if less than, unsigned
`bge rs1, rs2, imm[12:1]`	SB	Branch if greater or equal, 2's complement
`bgeu rs1, rs2, imm[12:1]`	SB	Branch if greater or equal, unsigned
`jal rd, imm[20:1]`	UJ	Jump and link
`jalr rd, rs1, imm[11:0]`	I	Jump and link register

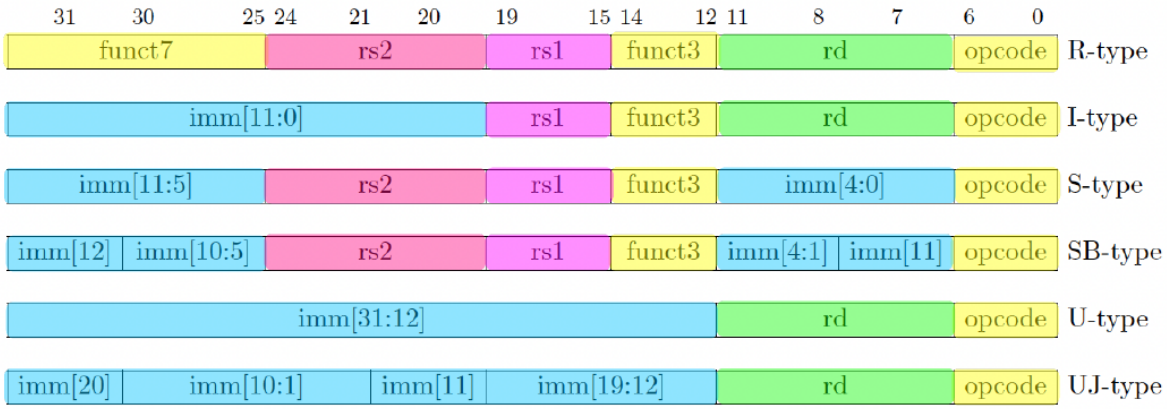
RV32I Memory Access Instructions

Instruction	Format	Meaning
`lb rd, imm[11:0](rs1)`	I	Load byte, signed
`lbu rd, imm[11:0](rs1)`	I	Load byte, unsigned
`lh rd, imm[11:0](rs1)`	I	Load half-word, signed
`lhu rd, imm[11:0](rs1)`	I	Load half-word, unsigned
`lw rd, imm[11:0](rs1)`	I	Load word
`sb rs2, imm[11:0](rs1)`	S	Store byte
`sh rs2, imm[11:0](rs1)`	S	Store half-word
`sw rs2, imm[11:0](rs1)`	S	Store word

Instruction Formats and Opcode Map

Instruction Formats

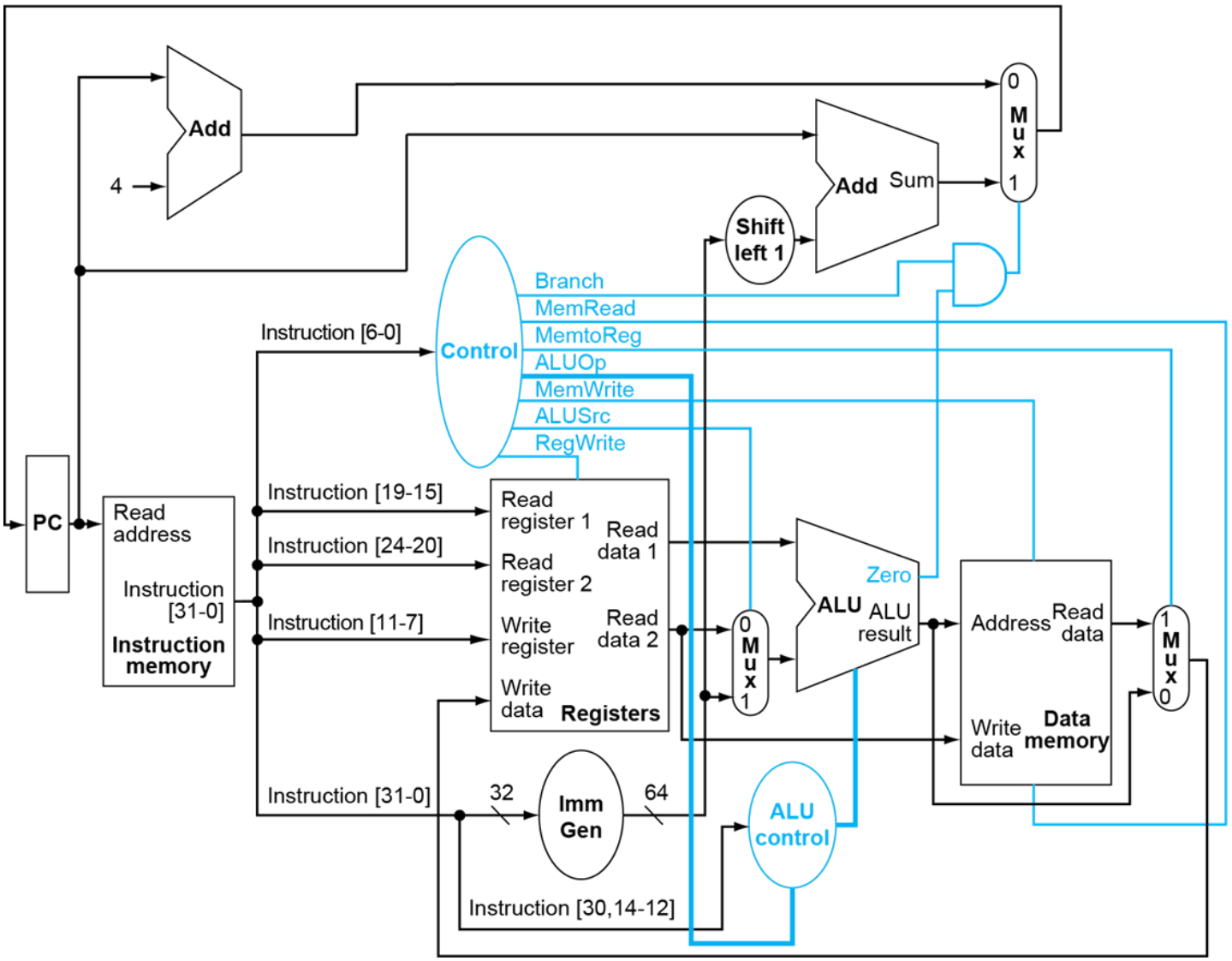
SB and UJ types are referred to as B and J type in RISC-V specifications.



RV32I OpCode Map

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111
00	Loads	*F Ext.*		Fences	Arithmetic	AUIPC	*RV64I*	
01	Stores	*F Ext.*		*A Ext.*	Arithmetic	LUI	*RV64I*	
10	*F Ext.*	*F Ext.*	*F Ext.*	*F Ext.*	*F Ext.*		*RV128I*	
11	Branches	JALR		JAL	System		*RV128I*	

RISC-V Datapath that needs to be implemented



Modules Designed & Purpose of each Module

For a more in-depth understanding of the workings of each module, refer to the System Verilog code relevant to each module in Appendix A and the gate level schematics of each module in Appendix B.

Test Bench Module (tb_top.sv)

Consists of non-synthesizable Verilog code which is used to run a RTL simulation of the designed RISC-V processor. Uses an artificially generated clock signal (of period 10ns) along with a reset signal as inputs.

This RTL simulation was used to verify the functionality of the designed processor before implementing it on the FPGA.

In this basic implementation, no external inputs and outputs were used. But rather, the CPU execution was carried out depending on the instructions and data that were programmed onto the instruction memory and data memory respectively. (Code)

Processor Module (RISC_V.sv)

The designed 32-bit RISC-V processor supports the 37 instructions mentioned in the previous section.

The processor module combines the Controller (control path), ALU controller, and Data path. (Code)

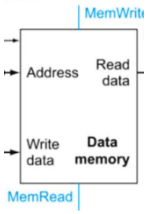
ALU (alu.sv)

Responsible for carrying out the arithmetic operations. The designed ALU gets a 4-bit control signal as input and two 32-bit operands as inputs, and outputs a 32-bit value after carrying out the relevant operation. (Code)

The following table shows the operations that the designed ALU is capable of performing along with the relevant ALU control signals.

ALU Control Signal	Instruction Name	Function
0000	AND	Bitwise AND
0001	OR	Bitwise OR
0011	XOR	Bitwise XOR
0010	ADD	Addition
0110	SUB	Signed Subtraction (Also sets flags: BLT, BGT, Zero)
0111	SUBU	Unsigned Subtraction (Also sets flags: BLT, BGT, Zero)
0100	SLL	Shift Left Logical
1000	SRL	Shift Right Logical
1100	SRA	Shift Right Arithmetic
0101	SLTU	Set (1) on Less than (Unsigned)
1010	SLT	Set (1) on Less than (Signed)
default	-	Set ALUResult = 0 (Invalid ALU Control Signal)

Data Memory (datamemory.sv)



Separate memories were used for instructions and data.

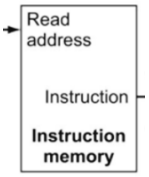
The designed data memory has 512 words (9-bit address bus) with each word of size 32 bits. Load and store instructions access data memory.

Memory reads are combinational/level triggered: Whenever the MemRead control signal is high, the memory word pointed by the address bus is read and transferred to the ReadData bus.

Memory writes are synchronous/edge triggered: At a positive clock edge, if the MemWrite control signal is high, the data at the WriteData bus will be written to the address pointed by the address bus.

For testing purposes, data memory was initialized with different values at different locations that were used by the Test Programs used to verify the functionality of the processor. (Code)

Instruction Memory (instructionmemory.sv)

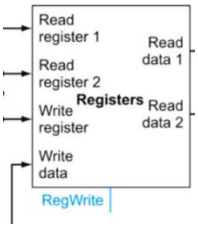


Instruction memory is the memory that instructions are fetched from.

The designed instruction memory takes a 9-bit address as input and outputs a 32-bit instruction.

For testing purposes, instruction memory was initialized with different instructions according to the test programs that were executed. (Code)

Register File (RegFile.sv)



The register file holds 32 registers with each register of size 32 bits. Reg[0], Reg[1],…, Reg[31].

Consists of 3 address buses as inputs,

- Write register **(rd)**
- Read register 1 **(rs1)**
- Read register 2 **(rs2)**

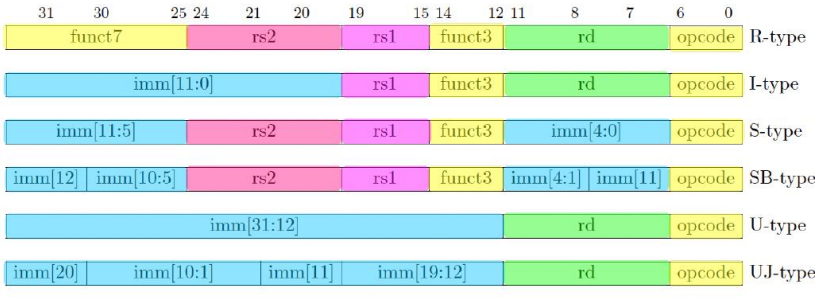
Consists of 1 data bus as input (Write data), and 2 data buses as output (Read data 1 and Read data 2).

Reads can be done at any time but write is done only if RegWrite is high at negative clock edges. (Code)

Immediate Generator (imm_Gen.sv)

Immediate operands are distributed in different ways for different instructions in RISC-V architecture.

So, the immediate generator module was designed to take the instruction as input and, to extract and output the immediate operand depending on the instruction type. (Code)



Controller (Controller.sv)

	imm[31:12]				rd	0110111	LUI
	imm[31:12]				rd	0010111	AUIPC
	imm[20|10:1|11|19:12]				rd	1101111	JAL
	imm[11:0]		rs1	000	rd	1100111	JALR
SB-Type Control Transfer	imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
	imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
	imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
	imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
	imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
	imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
I-Type Memory Access	imm[11:0]		rs1	000	rd	0000011	LB
	imm[11:0]		rs1	001	rd	0000011	LH
	imm[11:0]		rs1	010	rd	0000011	LW
	imm[11:0]		rs1	100	rd	0000011	LBU
	imm[11:0]		rs1	101	rd	0000011	LHU
S-Type Memory Access	imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
	imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
	imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
I-Type Computational	imm[11:0]		rs1	000	rd	0010011	ADDI
	imm[11:0]		rs1	010	rd	0010011	SLTI
	imm[11:0]		rs1	011	rd	0010011	SLTIU
	imm[11:0]		rs1	100	rd	0010011	XORI
	imm[11:0]		rs1	110	rd	0010011	ORI
	imm[11:0]		rs1	111	rd	0010011	ANDI

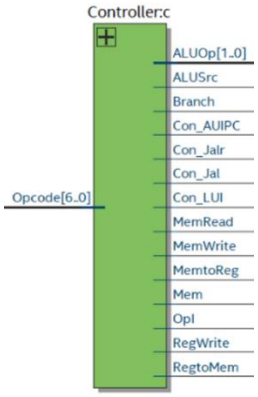
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

I-Type Computational

R-Type Computational

The opcode relevant to each instruction type is shown in the table below.

Opcode	Instruction Type
7'b0110011	R-Type (Computational) : ADD, SUB, AND, OR, XOR, SLT, SLTU, SLL, SRL, SRA
7'b0010011	I-Type (Computational) : ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI
7'b1100011	SB-Type (Control Transfer) / Branches : BEQ, BNE, BLT, BGE, BLTU, BGEU
7'b0000011	I-Type (Memory Access) / Loads : LB, LH, LW, LBU, LHU
7'b0100011	S-Type (Memory Access) / Stores : SB, SH, SW
7'b1101111	JAL
7'b1100111	JALR
7'b0010111	AUIPC
7'b0110111	LUI



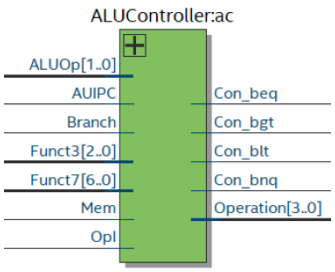
The controller consists of hardware elements that determine control signals. Control signals are the signals which specify what the datapath elements should do to the data.

This module takes the opcode of an instruction as input and sets the control signals accordingly. Controller was designed using a microprogramming approach instead of hardwired approach. Here, control signals are stored in a control store and sent out when required rather than computing control signals in real time using combinational logic gates. Control signals: ALUSrc, MemtoReg, RegtoMem, RegWrite, MemRead, MemWrite, Branch, ALUOp, Con_Jalr, Con_Jal, Mem, OpI, Con_AUIPC, Con_LUI. (Code)

	R TYPE: b'0110011	LW: b'0000011	SW: b'0100011	Rtypel: b'0010011	BR: b'1100011	JALR: b'1100111	JAL: b'1101111	AUIPC: b'0010111	LUI: b'0110111
Arusrc	0	1	1	1	0	0	0	0	0
Memtoreg	0	1	0	0	0	0	0	0	0
Regtomem	0	0	1	0	0	0	0	0	0
Regwrite	1	1	0	1	0	1	1	0	0
Memread	0	1	0	0	0	0	0	0	0
Memwrite	0	0	1	0	0	1	0	0	0
Branch	0	0	0	0	1	0	0	0	0
Aluop [0]	0	0	0	0	1	0	0	0	0
Aluop [1]	1	0	0	0	0	0	0	0	0
Con_jalr	0	0	0	0	0	1	0	0	0
Con_jal	0	0	0	0	0	0	1	0	0
Mem	0	1	1	0	0	0	0	0	0
Opi	0	0	0	1	0	0	0	0	0
Con_auipc	0	0	0	0	0	0	0	1	0
Con_lui	0	0	0	0	0	0	0	0	1
Control Bits	000100001000000	110110000001000	101001000001000	100100000000100	000000110000000	000101000100000	000100000010000	000000000000010	000000000000001

ALU Controller (ALUController.sv)

The main task of the ALU Controller is to generate the ALU control signals which determines which operation the ALU needs to perform. (Code)



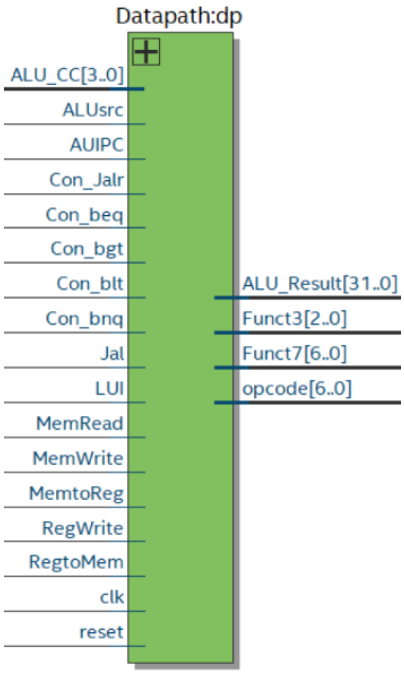
Inputs: ALUOp, Funct3, Funct7, Branch, Mem, OpI, AUIPC.

Outputs: 4-bit ALU control signal, Con_beq, Con_bnq, Con_blt, Con_bgt.

RISCV Data Path (Datapath.sv)

Consists of hardware elements that deal with and transform signals,

- Functional units that operate on data. E.g., ALU, Sign extenders
- Storage units that store data. E.g., Registers
- Hardware structure that enables the flow of data into the functional units and registers. E.g., Wires, Buses, Multiplexers, Decoders.



The major components of the data path are,

- ALU
- Register File
- Instruction Memory
- Data Memory
- Data Extractor Module
- Immediate Generator Module
- PCReg
- Adders
- Multiplexers

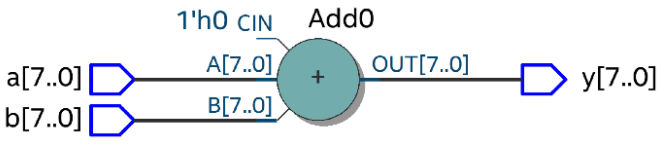
(Code)

Data Extractor (data_extract.sv)

Used by load and store instructions to perform actions such as load byte, load halfword, load word, etc. (Code)

Adder (adder.sv)

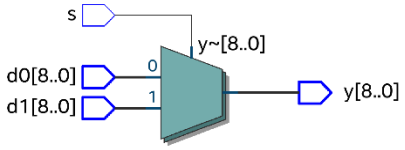
A simple module that takes 2 numbers as input and outputs their sum. By default, the width parameter was set to 8 in this module. So, it takes 8-bit numbers as input and outputs an 8-bit number. But this width parameter can be changed if required. (Code)



Further, a 32-bit adder module was also defined. (Code)

2 input Multiplexer (mux2.sv)

Takes 2 input signals and gives one of them to the output depending on the select control signal (If s==1, output = d1. Else output = d0).
Many of these multiplexers were used in the Datapath implementation.



(Code)

D flip-flop (flopr.sv)

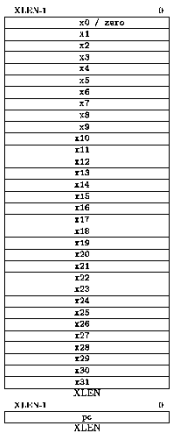
A binary storage element that can store 1 bit, which changes its value depending on the clock signal.
Used in the datapath implementation for PCReg (Program counter).

(Code)

Testbench Simulations

Generally, the 32 registers of RISC-V ISA are designated as follows,

- x0: the constant value 0
- x1: return address
- x2: stack pointer
- x3: global pointer
- x4: thread pointer
- x5 – x7, x28 – x31: temporaries
- x8: frame pointer
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments



	imm[31:12]				rd	0110111	LUI
	imm[31:12]				rd	0010111	AUIPC
	imm[20|10:1|11|19:12]				rd	1101111	JAL
	imm[11:0]		rs1	000	rd	1100111	JALR
SB-Type Control Transfer	imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
	imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
	imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
	imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
	imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
	imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
I-Type Memory Access	imm[11:0]		rs1	000	rd	0000011	LB
	imm[11:0]		rs1	001	rd	0000011	LH
	imm[11:0]		rs1	010	rd	0000011	LW
	imm[11:0]		rs1	100	rd	0000011	LBU
	imm[11:0]		rs1	101	rd	0000011	LHU
S-Type Memory Access	imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
	imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
	imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
I-Type Computational	imm[11:0]		rs1	000	rd	0010011	ADDI
	imm[11:0]		rs1	010	rd	0010011	SLTI
	imm[11:0]		rs1	011	rd	0010011	SLTIU
	imm[11:0]		rs1	100	rd	0010011	XORI
	imm[11:0]		rs1	110	rd	0010011	ORI
	imm[11:0]		rs1	111	rd	0010011	ANDI

0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

I-Type Computational

R-Type Computational

When each test program was tested using RTL simulations, the assembly (machine) instructions relevant to the test program were loaded onto the instruction memory. (Code)

Data memory was initialized with the following set of values that were used by all test programs. (Code)

Mem[0]	Mem[1]	Mem[2]	Mem[3]	Mem[4]	Mem[5]	Mem[6]	Mem[7]	Mem[8]	Mem[9]
0	0	0	0	5_{10}	0	0	0	-9_{10}	0

Mem[10]	Mem[11]	Mem[12]	Mem[13]	Mem[14]	Mem[15]	Mem[16]	Mem[17]	Mem[18]	Mem[19]
0	0	12_{10}	14_{10}	-18_{10}	0	0	0	0	0

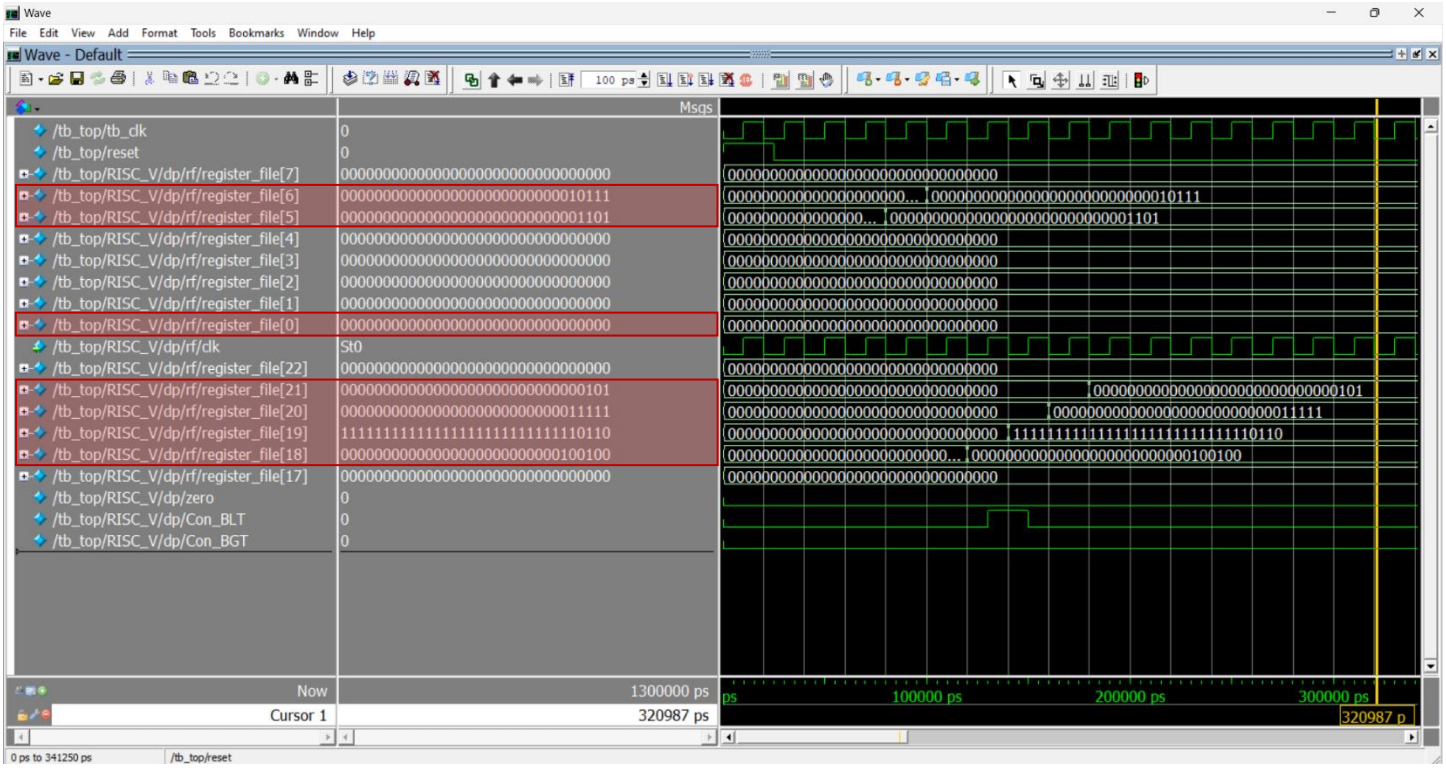
Test Program 1 (R-Type Computational / Register-Register)

Binary	Assembly	Instruction Type	Meaning
00000000110100000000001010010011	ADDI x5, x0, 13_{10}	I-Type	R5 ← R0 + 13_{10}
00000001011100000000001100010011	ADDI x6, x0, 23_{10}	I-Type	R6 ← R0 + 23_{10}
00000000011000101000100100110011	ADD x18, x5, x6	R-Type	R18 ← R5 +R6
01000000011000101000100110110011	SUB x19, x5, x6	R-Type	R19 ← R5 – R6
00000000011000101110101000110011	OR x20, x5, x6	R-Type	R20 ← R5 || R6
00000000011000101111101010110011	AND x21, x5, x6	R-Type	R21 ← R5 && R6

Note: Two I-Type instructions were performed at the start to initialize values in registers R5(x5) and R6(x6) before using those 2 registers to perform R-Type instructions.

Expected values in registers at the end of the simulation considering the instructions of the test program are,

R0	R5	R6	R18	R19	R20	R21
0	13_{10} (1101_2)	23_{10} (10111_2)	36_{10} (100100_2)	-10_{10} ($111…1110110_2$)	11111_2	00101_2



It can be observed from the above RTL simulation that the expected values of registers at the end of the program execution perfectly match with the actual register values in the simulation.

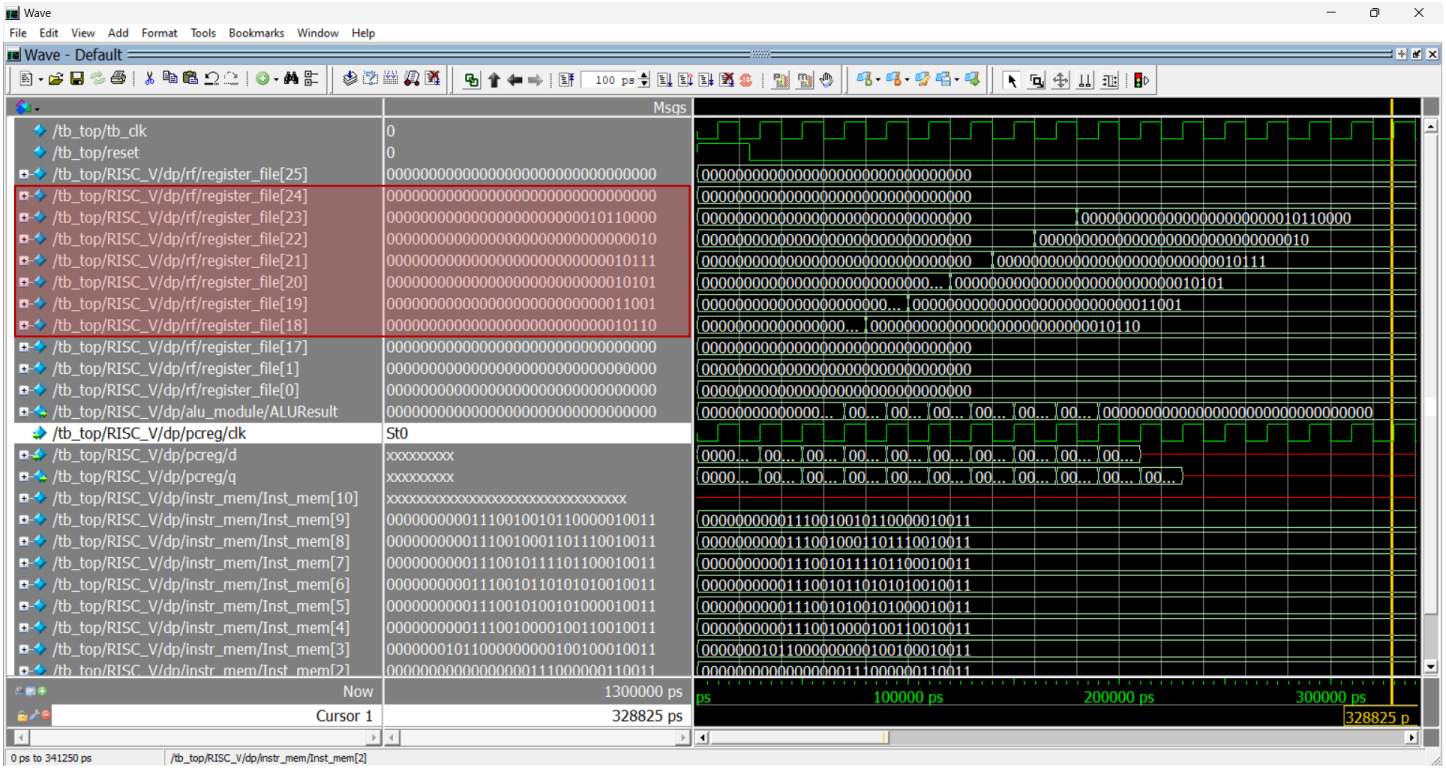
Hence, R-Type computational instructions seem to be working properly.

Test Program 2 (I-Type Computational / Short Immediate Operands)

Binary	Assembly	Instruction Type	Meaning
00000001011000000000100100010011	ADDI x18, x0, 22_{10}	I-Type	R18 ← R0 + 22_{10}
00000000001110010000100110010011	ADDI x19, x18, 3_{10}	I-Type	R19 ← R18 + 3_{10}
00000000001110010100101000010011	XORI x20, x18, 3_{10}	I-Type	R20 ← R18 ^ 3_{10}
00000000001110010110101010010011	ORI x21, x18, 3_{10}	I-Type	R21 ← R18 || 3_{10}
00000000001110010111101100010011	ANDI x22, x18, 3_{10}	I-Type	R22 ← R18 && 3_{10}
00000000001110010001101110010011	SLLI x23, x18, 3_{10}	I-Type	R23 ← R18 SLLI(3_{10})
00000000001110010010110000010011	SLTI x24, x18, 3_{10}	I-Type	R24 ← R18 SLTI(3_{10})

Expected values in registers at the end of the simulation considering the instructions of the test program are,

R18	R19	R20	R21	R22	R23	R24
22_{10} (10110_2)	25_{10} (11001_2)	10101_2	10111_2	00010_2	10110000_2	0



It can be observed from the above RTL simulation that the expected values of registers at the end of the program execution perfectly match with the actual register values in the simulation.

Hence, I-Type computational instructions also seem to be working properly.

Test Program 3 (S-Type Memory Access / Stores)

Binary	Assembly	Instruction Type	Meaning
00100000010100000000001100010011	ADDI x6, x0, 517_{10}	I-Type	R6 ← R0 + 517_{10}
00000000011000000010001010100011	SW x6, 5_{10} (x0)	S-Type	Mem[R0+5_{10}] ← x6 (Word)
00000000011000000000001100100011	SB x6, 6_{10} (x0)	S-Type	Mem[R0+6_{10}] ← x6 (Byte)
00000000011000000001001110100011	SH x6, 7_{10} (x0)	S-Type	Mem[R0+7_{10}] ← x6 (Halfword)

Note: The first I-Type instruction was done to initialize a value in register R6(x6) so that it can be used by the store instructions. Since R0 is the zero register, when R0 is used as base register in base register addressing, the memory location pointed by R0 is Mem[0].

Initial values in data memory are as follows,

Mem[0]	Mem[1]	Mem[2]	Mem[3]	Mem[4]	Mem[5]	Mem[6]	Mem[7]	Mem[8]	Mem[9]
0	0	0	0	5_{10}	0	0	0	-9_{10}	0

$517_{10} = 0010\ 0000\ 0101_2$

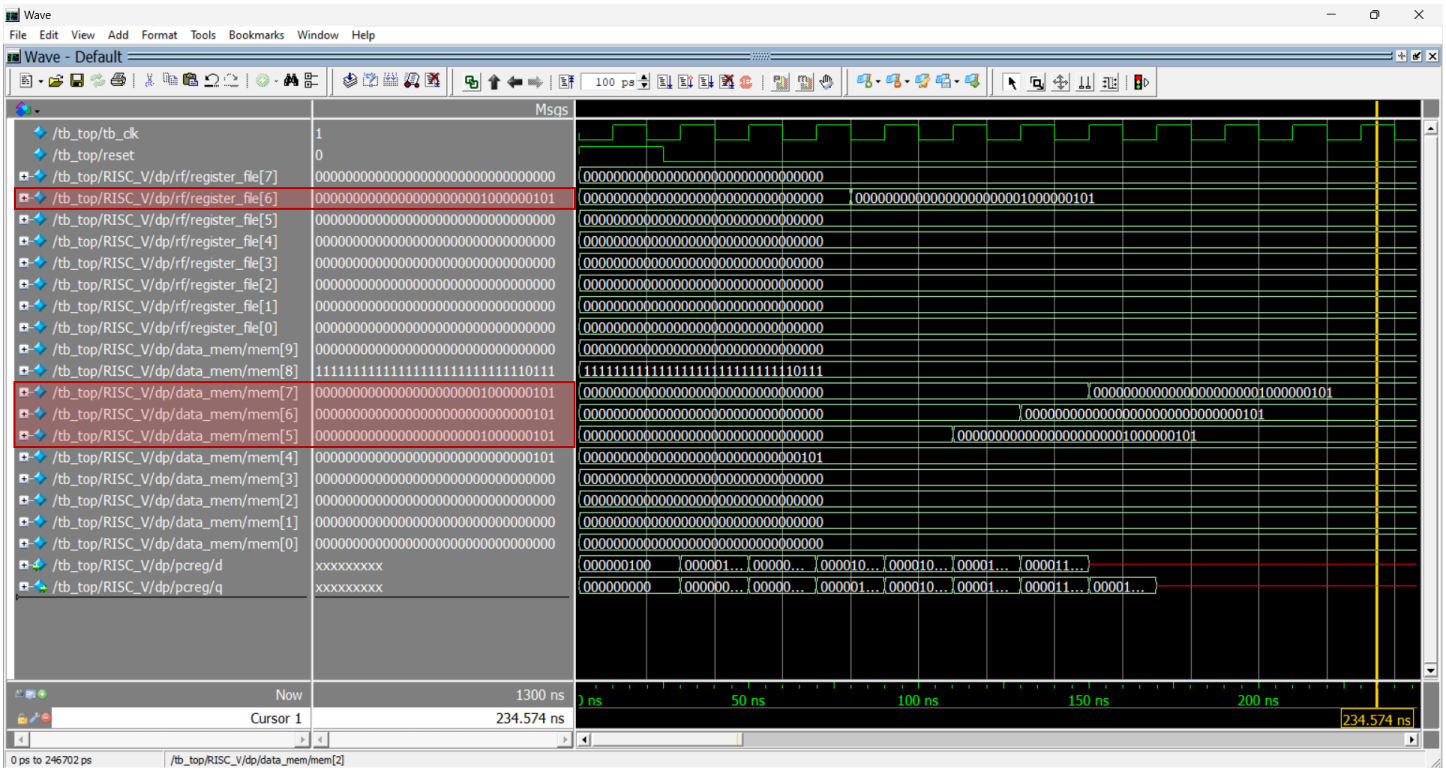
517 will be stored in R6 as a 32-bit number. To represent 517 in binary, at least 10 bits are required.

So, when calling different methods of storing, it may or may not be changed during storage in memory.

- Store word: Word is 32 bits. So, 517 can be stored without any issue.
- Store halfword: Halfword is 16 bits. So, 517 can be stored without any issue.
- Store byte: Byte is 8 bits. So, 517 cannot be stored. The 2 most significant bits will be ignored and the actual value that will be stored is 0000 0101.

Expected values in data memory at the end of the RTL simulation are as follows,

Mem[R0+5] = Mem[5]	Mem[R0+6] = Mem[6]	Mem[R0+7] = Mem[7]
$0010\ 0000\ 0101_2$ (517_{10})	$0000\ 0101_2$ (5_{10})	$0010\ 0000\ 0101_2$ (517_{10})



Test Program 4 (I-Type Memory Access / Loads)

Binary	Assembly	Instruction Type	Meaning
00000000100000000000100100000011	LB x18, 8_{10} (x0)	I-Type	R18 ← Mem[R0+8_{10}] (Byte Signed)
00000000100000000001001100000011	LBU x19, 8_{10} (x0)	I-Type	R19 ← Mem[R0+8_{10}] (Byte Unsigned)
00000000100000000001101000000011	LH x20, 8_{10} (x0)	I-Type	R20 ← Mem[R0+8_{10}] (Halfword Signed)
00000000100000000101101010000011	LHU x21, 8_{10} (x0)	I-Type	R21 ← Mem[R0+8_{10}] (Halfword Unsigned)
00000000100000000010101100000011	LW x22, 8_{10} (x0)	I-Type	R22 ← Mem[R0+8_{10}] (Word)

Initial values in data memory are as follows,

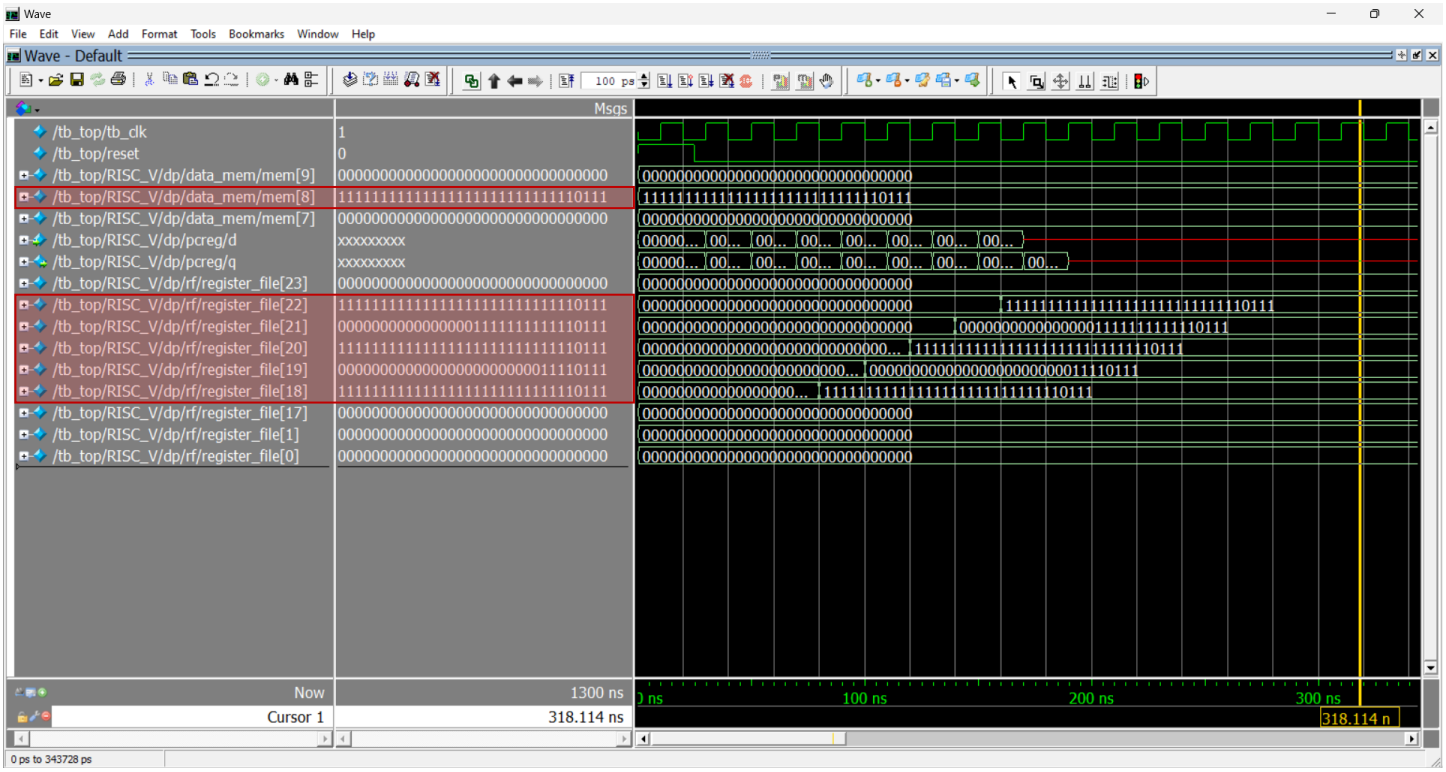
Mem[0]	Mem[1]	Mem[2]	Mem[3]	Mem[4]	Mem[5]	Mem[6]	Mem[7]	Mem[8]	Mem[9]
0	0	0	0	5_{10}	0	0	0	-9_{10}	0

Value at Mem[R0+8] = Value at Mem[8] = -9_{10} = 1111 1111 1111 1111 1111 1111 1111 0111_2

To represent -9_{10}, at least 5 bits are required. 10111_2 (In 2's complement notation). Because of this, load byte, load halfword and, load word does not make any difference when the value in memory to be loaded is -9_{10}.

Expected values in registers at the end of the RTL simulation are as follows,

R18	R19	R20	R21	R22
Load byte	Load byte unsigned	Load halfword	Load halfword unsigned	Load word
1111 1111 1111 1111 1111 1111 1111 0111	0000 0000 0000 0000 0000 0000 1111 0111	1111 1111 1111 1111 1111 1111 1111 0111	0000 0000 0000 0000 1111 1111 1111 0111	1111 1111 1111 1111 1111 1111 1111 0111



It can be observed from the above RTL simulation that the expected values of registers at the end of the program execution perfectly match with the actual register values in the simulation.

Hence, I-Type memory access instructions (Load instructions) also seem to be working properly.

Test Program 5 (SB-Type Control Transfer / Conditional Branches)

Initial values in data memory are as follows,

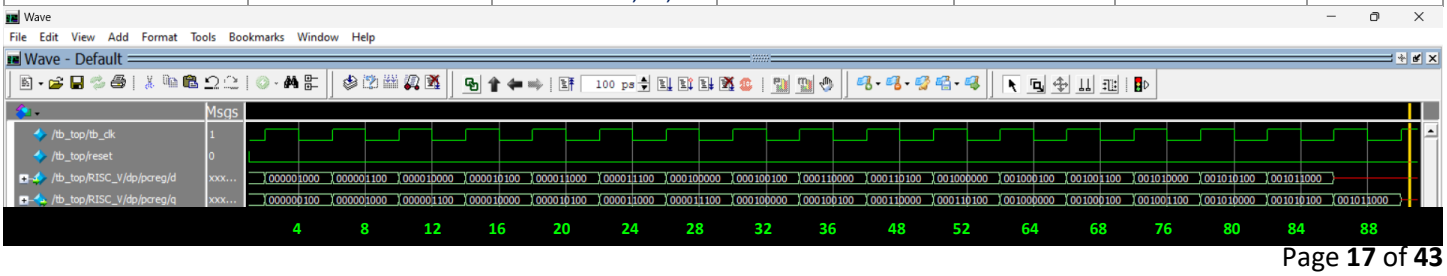
Mem[10]	Mem[11]	Mem[12]	Mem[13]	Mem[14]	Mem[15]	Mem[16]	Mem[17]	Mem[18]	Mem[19]
0	0	12	14	-18	0	0	0	0	0

Instructions of Test Program 5 which will be used to test Branch instructions and their expected behavior is given below,

Binary	Assembly Instruction	Instruction Type	Meaning	Instruction Executed or Not	Branch Taken or Not
00000000110000000010111000000011	LW x28, 12(x0)	I-Type	Load M[R0+12] to R28	Execute	-
00000000110100000010111010000011	LW x29, 13(x0)	I-Type	Load M[R0+13] to R29	Execute	-
00000000111000000010111100000011	LW x30, 14(x0)	I-Type	Load M[R0+14] to R30	Execute	-
00000000110000000010111110000011	LW x31, 12(x0)	I-Type	Load M[R0+12] to R31	Execute	-
00000001110111100000011001100011	BEQ x28, x29, 12	SB-Type	If R28==R29, PC=PC+12	Execute	No
00000001010100000000100100010011	ADDI x18, x0, 21	I-Type	R18 = R0 + 21	Execute	-
00000001111111100000011001100011	BEQ x28, x31, 12	SB-Type	If R28==R31, PC=PC+12	Execute	Yes
00000001011100000000100110010011	ADDI x19, x0, 23	I-Type	R19 = R0 + 23	Should not execute	-
00000001100100000000101000010011	ADDI x20, x0, 25	I-Type	R20 = R0 + 25	Should not execute	-
00000001101100000000101010010011	ADDI x21, x0, 27	I-Type	R21 = R0 + 27	Execute	-
00000001110111100001011001100011	BNE x28, x29, 12	SB-Type	If R28!=R29, PC=PC+12	Execute	Yes
00000001110100000000101100010011	ADDI x22, x0, 29	I-Type	R22 = R0 + 29	Should not execute	-
00000001111100000000101110010011	ADDI x23, x0, 31	I-Type	R23 = R0 + 31	Should not execute	-
00000010000100000000110000010011	ADDI x24, x0, 33	I-Type	R24 = R0 + 33	Execute	-
00000001111011100101010001100011	BGE x28, x30, 8	SB-Type	If R28>=R30, PC=PC+8	Execute	Yes
00000010001100000000110010010011	ADDI x25, x0, 35	I-Type	R25 = R0 + 35	Should not execute	-
00000010010100000000110100010011	ADDI x26, x0, 37	I-Type	R26 = R0 + 37	Execute	-
00000001111011100111010001100011	BGEU x28, x30, 8	SB-Type	If R28>=|R30|, PC=PC+8	Execute	No
00000010011100000000110110010011	ADDI x27, x0, 39	I-Type	R27 = R0 + 39	Execute	-

Program counter values are expected to change during execution of Test program 5 as follows,

Instruction Index	Instruction Address	Assembly Instruction	Instruction Executed or Not	Branch Taken or Not	PC Change after Instruction	Final PC Value
3	12	LW x28, 12(x0)	Execute	-	PC = PC +4	16
4	16	LW x29, 13(x0)	Execute	-	PC = PC +4	20
5	20	LW x30, 14(x0)	Execute	-	PC = PC +4	24
6	24	LW x31, 12(x0)	Execute	-	PC = PC +4	28
7	28	BEQ x28, x29, 12	Execute	No	PC = PC +4	32
8	32	ADDI x18, x0, 21	Execute	-	PC = PC +4	36
9	36	BEQ x28, x31, 12	Execute	Yes	PC = PC +12	48
10	40	ADDI x19, x0, 23	Should not execute	-	-	-
11	44	ADDI x20, x0, 25	Should not execute	-	-	-
12	48	ADDI x21, x0, 27	Execute	-	PC = PC +4	52
13	52	BNE x28, x29, 12	Execute	Yes	PC = PC + 12	64
14	56	ADDI x22, x0, 29	Should not execute	-	-	-
15	60	ADDI x23, x0, 31	Should not execute	-	-	-
16	64	ADDI x24, x0, 33	Execute	-	PC = PC +4	68
17	68	BGE x28, x30, 8	Execute	Yes	PC = PC + 8	76
18	72	ADDI x25, x0, 35	Should not execute	-	-	-
19	76	ADDI x26, x0, 37	Execute	-	PC = PC +4	80
20	80	BGEU x28, x30, 8	Execute	No	PC = PC +4	84
21	84	ADDI x27, x0, 39	Execute	-	PC = PC +4	88

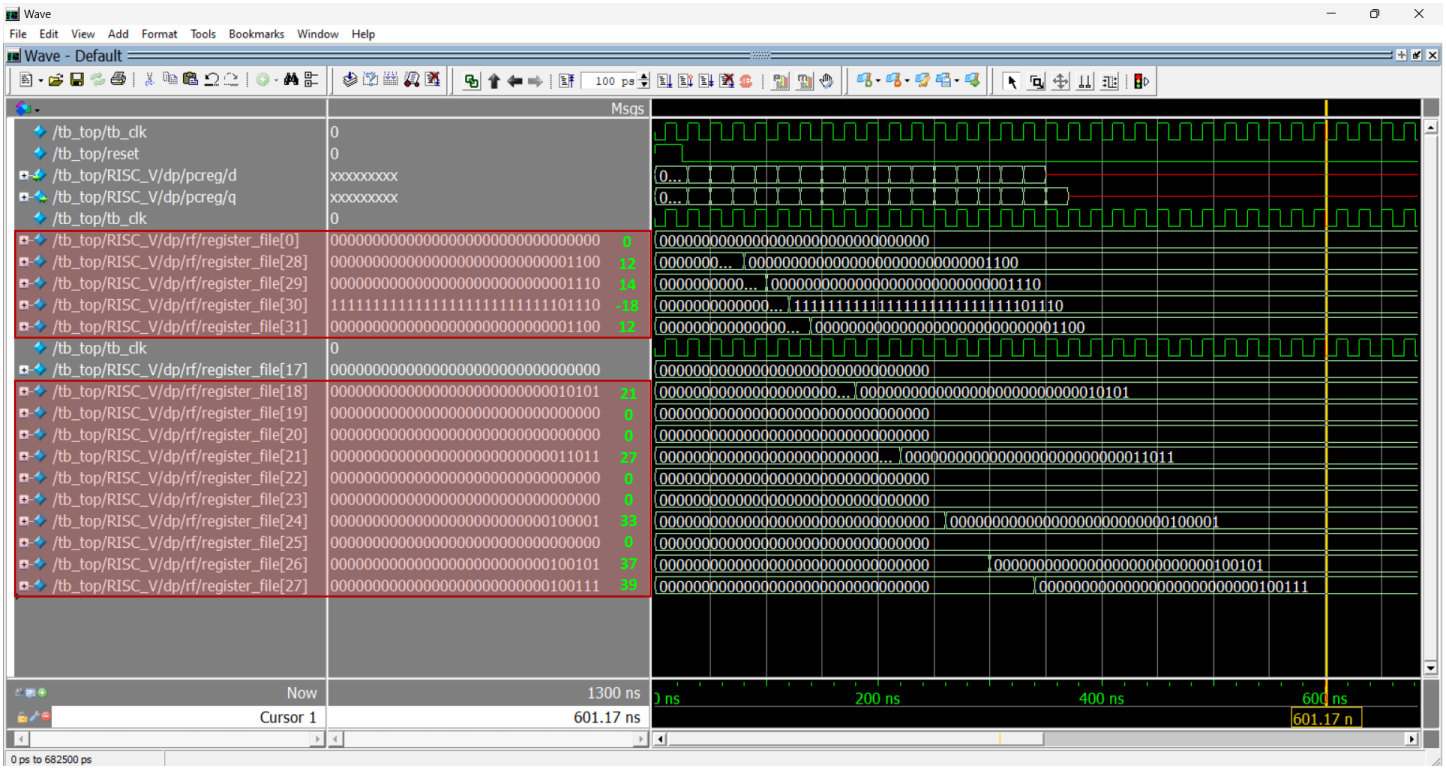


It can be observed from the above RTL simulation that the expected program counter (PC) values perfectly match with the program counter values in the simulation.

Expected values in registers at the end of execution of Test Program 5 are as follows,

R28	R29	R30	R31
12	14	-18	12

R18	R19	R20	R21	R22	R23	R24	R25	R26	R27
21	0	0	27	0	0	33	0	37	39



It can be observed from the above RTL simulation that the expected values of registers at the end of the program execution perfectly match with the actual register values in the simulation.

Hence, SB-Type control transfer instructions also seem to be working properly.

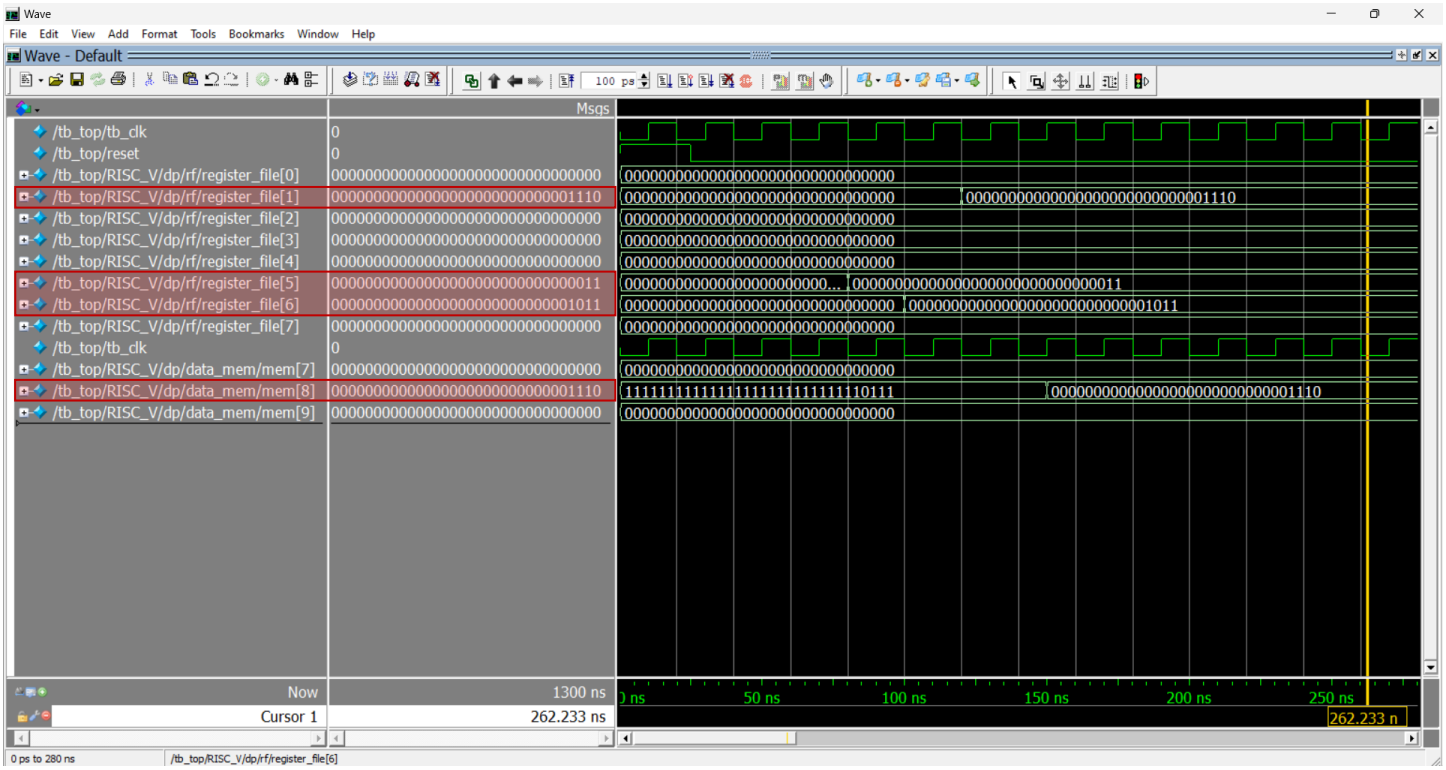
Test Program 6 (Adding 2 numbers and saving the result to memory)

Instructions of Test Program 6 are as follows,

Binary	Assembly	Instruction Type	Meaning
00000000001100000000001010010011	ADDI x5, x0, 3_{10}	I-Type	R5 ← R0 + 3_{10}
00000000101100000000001100010011	ADDI x6, x0, 11_{10}	I-Type	R6 ← R0 + 11_{10}
00000000011000101000000010110011	ADD x1, x5, x6	R-Type	R1 ← R5 + R6
00000000000010000001001000100011	SW x1, 8_{10} (x0)	S-Type	Mem[R0+8] ← R1

Expected values in registers and data memory at the end of the RTL simulation are as follows,

R5	R6	R1	Mem[R0+8] = Mem[8]
3_{10} (11_2)	11_{10} (1011_2)	$3_{10} + 11_{10} = 14_{10}$ (1110_2)	14_{10} (1110_2)



Test Program 7 (Program that reads the value of n from memory and computes the sum of numbers from 1 to n and stores the sum in memory)

Initial values in data memory are as follows,

Mem[11]	Mem[12]
0	12

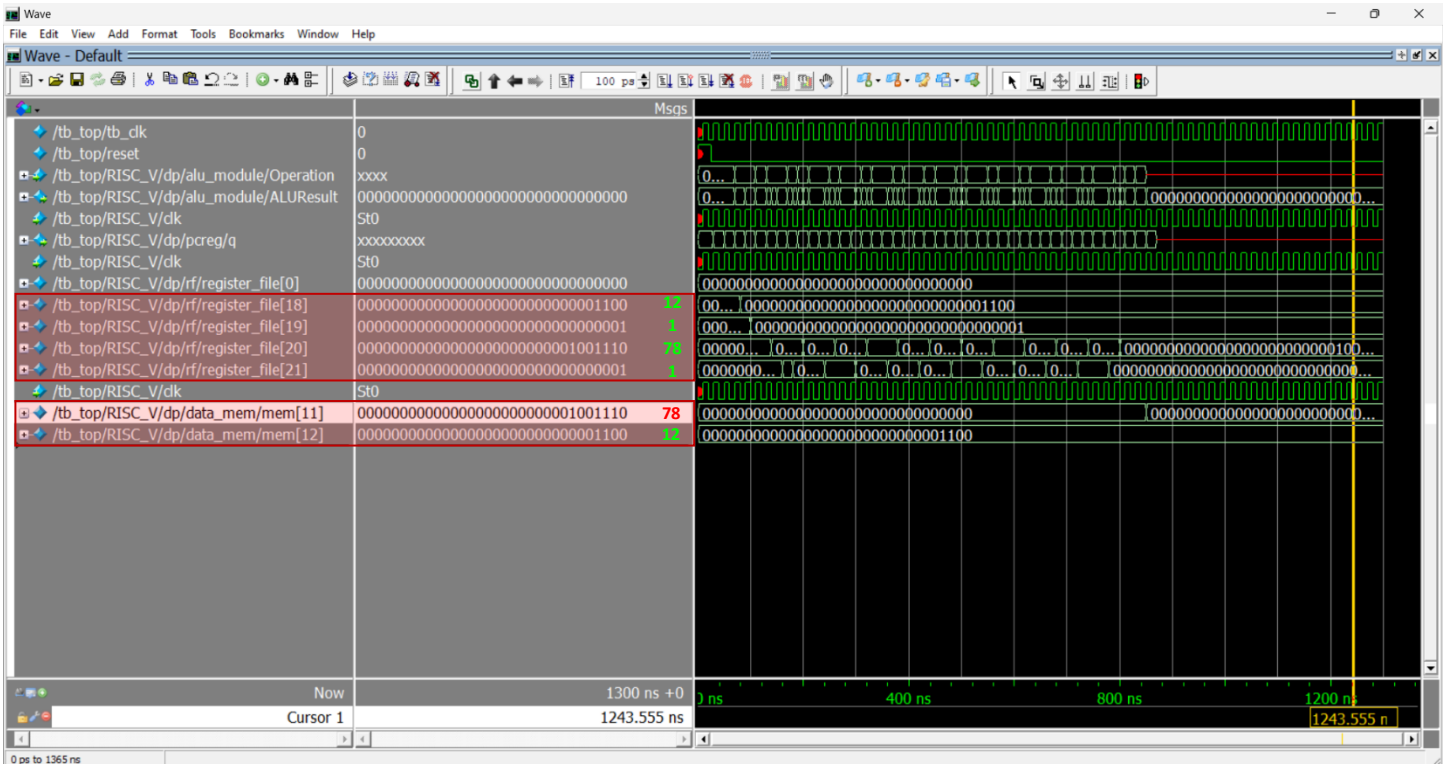
Instructions of Test Program 7 are as follows,

Binary	Assembly	Meaning	Register Usage	Instruction Type
00000000110000000010100100000011	LW x18, 12(x0)	Load M[R0+12] to R18	Storing n	I-Type
00000000000100000000100110010011	ADDI x19, x0, 1	R19 ← 1	Storing 1	I-Type
00000000000010100111101000010011	ANDI x20, x20, 0	R20 ← 0	-	I-Type
00000001001010100000101000110011	ADD x20, x20, x18	R20 ← R20 + R18	Calculating total	R-Type
00000000110000000010101010000011	LW x21, 12(x0)	Load M[R0+12] to R21	-	I-Type
01000001001110101000101010110011	SUB x21, x21, x19	R21 ← R21 – R19	Calculating n-1	R-Type
00000001010110101000001010000110011	ADD x20, x20, x21	R20 ← R20 + R21	-	R-Type
11111111001110101001110011100011	BNE x21, x19, -8	If R21!=R19, PC=PC-8	-	SB-Type
00000001010000000010010110100011	SW x20, 11(x0)	M[R0+11] ← R20	-	S-Type

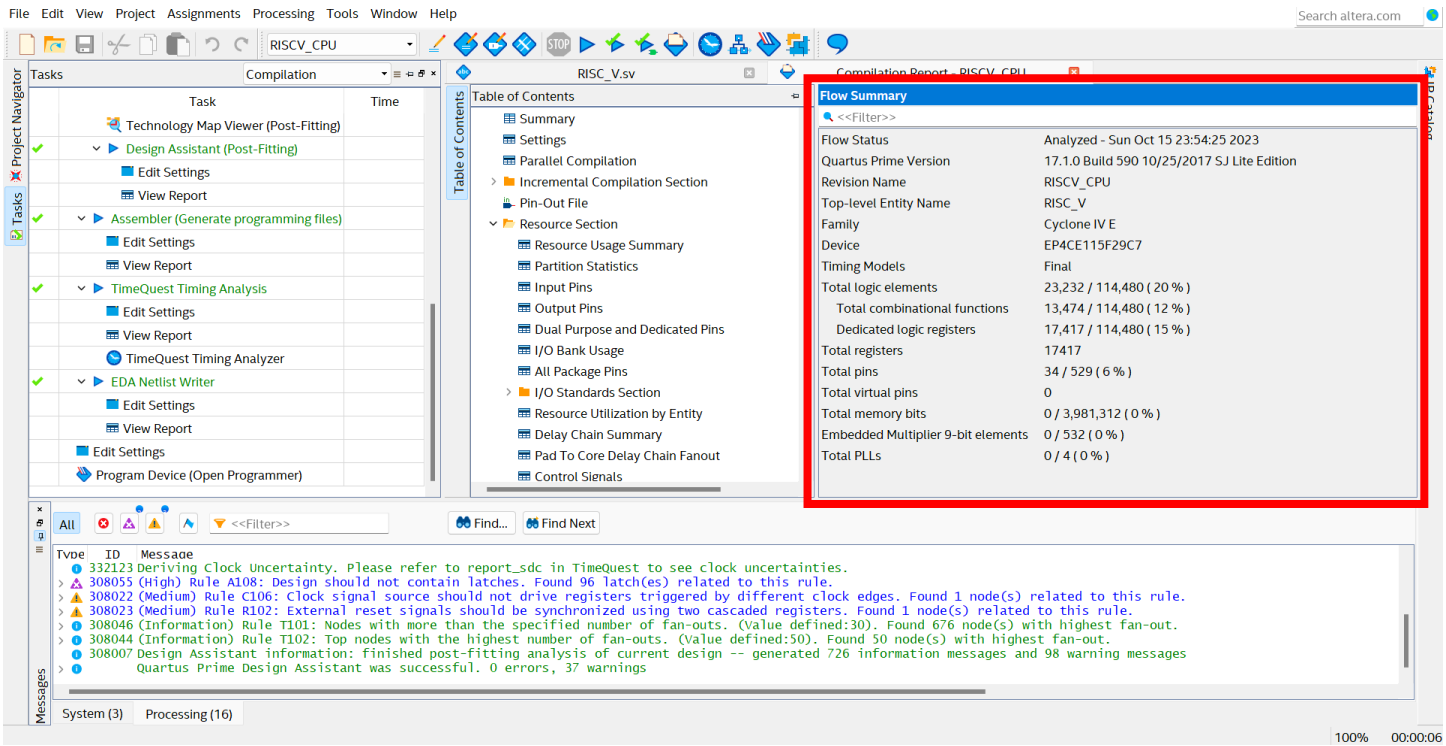
This program loads the value of memory location 12. Let n = Mem[12].
Then it computes the summation of all integers from 1 to n and stores the result in memory location 11.
Since the initial value of Mem[12] is equal to 12, this program finds the sum of integers from 1 to 12.

$$\sum_{n=1}^{12} n = 78$$

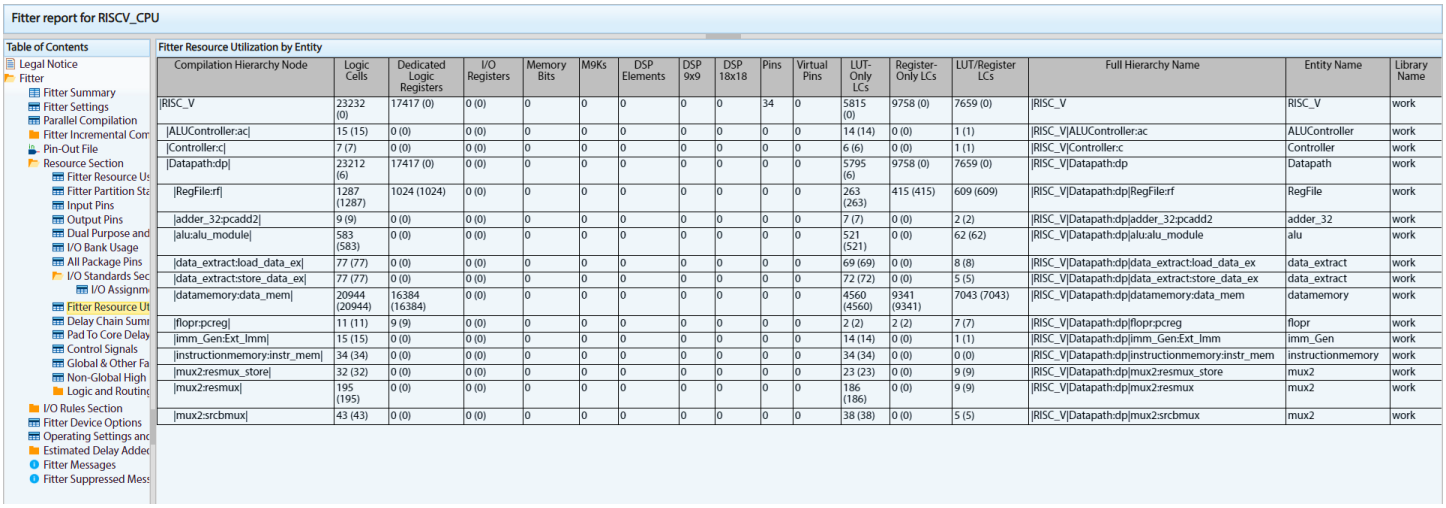
Therefore, the expected value to be stored in Mem[11] after the execution of the program is 78.



Resource Utilization



Flow Summary



Resource Utilization by Entity

Fitter report for RISCV_CPU

Table of Contents

- Legal Notice
- Fitter
  - Fitter Summary
  - Fitter Settings
  - Parallel Compilation
  - Fitter Incremental Com
  - Pin-Out File
  - Resource Section
    - Fitter Resource Us
    - Fitter Partition Sta
    - Input Pins
    - Output Pins
    - Dual Purpose and
    - I/O Bank Usage
    - All Package Pins
    - I/O Standards Sec
      - I/O Assignm
    - Fitter Resource Ut
    - Delay Chain Sumr
    - Pad To Core Delay
    - Control Signals
    - Global & Other Fa
    - Non-Global High
    - Logic and Routing
  - I/O Rules Section
  - Fitter Device Options
  - Operating Settings and
  - Estimated Delay Added
  - Fitter Messages
  - Fitter Suppressed Mess

Fitter Resource Usage Summary

Resource	Usage
Total logic elements	23,232 / 114,480 (20 %)
-- Combinational with no register	5815
-- Register only	9758
-- Combinational with a register	7659
Logic element usage by number of LUT inputs	
-- 4 input functions	12713
-- 3 input functions	662
-- <=2 input functions	99
-- Register only	9758
Logic elements by mode	
-- normal mode	13305
-- arithmetic mode	169
Total registers*	17,417 / 117,053 (15 %)
-- Dedicated logic registers	17,417 / 114,480 (15 %)
-- I/O registers	0 / 2,573 (0 %)
Total LABs: partially or completely used	1,813 / 7,155 (25 %)
Virtual pins	0
I/O pins	34 / 529 (6 %)
-- Clock pins	2 / 7 (29 %)
-- Dedicated input pins	0 / 9 (0 %)
M9Ks	0 / 432 (0 %)
Total block memory bits	0 / 3,981,312 (0 %)
Total block memory implementation bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
PLLs	0 / 4 (0 %)
Global signals	4
-- Global clocks	4 / 20 (20 %)
JTAGs	0 / 1 (0 %)
CRC blocks	0 / 1 (0 %)
ASMI blocks	0 / 1 (0 %)
Oscillator blocks	0 / 1 (0 %)
Impedance control blocks	0 / 4 (0 %)
Average interconnect usage (total/H/V)	19.7% / 19.2% / 20.4%
Peak interconnect usage (total/H/V)	90.2% / 88.2% / 94.3%
Maximum fan-out	17411
Highest non-global fan-out	2067
Total fan-out	115066
Average fan-out	2.83

Resource Usage Summary

References

[1] https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.pdf

[2] https://inst.eecs.berkeley.edu/~cs61c/sp20/pdfs/lectures/lec12.pdf

[3] https://safari.ethz.ch/digitaltechnik/spring2022/lib/exe/fetch.php?media=onur-digitaldesign_comparch-2022-lecture11-microarchitecture-fundamentals-afterelecture.pdf

[4] https://msyksphinz-self.github.io/riscv-isadoc/html/rvi.html

[5] https://luplab.gitlab.io/rvcodecjs/

[6] https://www.google.co.uk/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwirnIGnkuyBAxWcT2wGHTWPBiwQFnoECBQQAQ&url=https%3A%2F%2Fpages.hmc.edu%2Fharris%2Fclass%2Fe85%2Fold%2Ffall21%2Flect18.pptx&usg=AOvVaw2w0DbzbZ3TtuqTZTnzHmSZ&opi=89978449

Appendix A: System Verilog Codes of Modules

Test Bench Module (tb_top.sv)

```
`timescale 1ns / 1ps

module tb_top;

    logic tb_clk, reset;
    logic [31:0] tb_WB_Data;

    always #10 tb_clk = ~tb_clk;   // Clock Signal Generation (10ns period)

    initial                        // Reset Signal (Starts at 1, then goes to 0 after 25ns)
        begin
            tb_clk = 0;
            reset = 1;
            #25 reset = 0;
        end

    RISC_V RISC_V                  // DUT Instantiation
    (
        .clk(tb_clk),
        .reset(reset),
        .WB_Data(tb_WB_Data)
    );

    initial                        // Terminate the simulation after 1200ns
        begin
            #1200;
            $finish;
        end

endmodule
```

Processor Module (RISC_V.sv)

```
`timescale 1ns / 1ps

module RISC_V#(
    parameter DATA_W = 32
    )
    (
    input  logic clk, reset,      // Clock and reset
    output logic [31:0] WB_Data   // The ALU_Result
    );

    logic [6:0] opcode;
    logic ALUSrc, MemtoReg,RegtoMem, RegWrite, MemRead, MemWrite, Con_Jalr;
    logic Con_beq, Con_bnq, Con_bgt, Con_blt, Con_Jal,Branch, Mem,OpI,AUIPC,LUI;

    logic [1:0] ALUop;
    logic [6:0] Funct7;
    logic [2:0] Funct3;
    logic [3:0] Operation;

    Controller c(opcode, ALUSrc, MemtoReg,RegtoMem, RegWrite, MemRead, MemWrite, Branch, ALUop, Con_Jalr,Con_Jal,Mem, OpI, AUIPC, LUI);

    ALUController ac(ALUop, Funct7, Funct3, Branch,Mem,OpI,AUIPC, Operation, Con_beq, Con_bnq, Con_blt, Con_bgt);

    Datapath dp(clk, reset, RegWrite , MemtoReg, RegtoMem, ALUSrc , MemWrite, MemRead, Con_beq, Con_bnq, Con_bgt, Con_blt, Con_Jalr,Con_Jal,
AUIPC, LUI,Operation, opcode, Funct7, Funct3, WB_Data);

endmodule
```

ALU (alu.sv)

```
`timescale 1ns / 1ps

module alu#(
        parameter DATA_WIDTH    = 32,
        parameter OPCODE_LENGTH = 4
        )
        (
        input logic [DATA_WIDTH-1:0]     SrcA,
        input logic [DATA_WIDTH-1:0]     SrcB,
        input logic [OPCODE_LENGTH-1:0]  Operation,

        output logic[DATA_WIDTH-1:0]     ALUResult,

        output logic Con_BLT,
        output logic Con_BGT,
        output logic zero
        );

        integer  i;

        always_comb
            begin
                ALUResult = 'd0;
                Con_BLT   = 'b0;
                Con_BGT   = 'b0;
                zero      = 'b0;

                case(Operation)
                    ////////////////////////////////////////////////////////////
                    4'b0000:        // AND
                            ALUResult = SrcA & SrcB;
                    4'b0001:        // OR
                            ALUResult = SrcA | SrcB;
                    4'b0011:        // XOR
                            ALUResult = SrcA ^ SrcB;
                    ////////////////////////////////////////////////////////////
                    4'b0010:        // ADD
                            ALUResult = SrcA + SrcB;
                    4'b0110:        // SUB (Subtract signed)
                        begin
                            ALUResult = $signed(SrcA)       -  $signed(SrcB) ;
                            Con_BLT   = ($signed(ALUResult) <  $signed(1'd0));
                            Con_BGT   = ($signed(ALUResult) >  $signed(1'd0));
                            zero      = ($signed(ALUResult) == $signed(1'd0));
                        end
                    4'b0111:
                        begin       // SUBU (Subtract unsigned)
                            ALUResult = SrcA - SrcB;
                            Con_BLT   = SrcA < SrcB;
                            Con_BGT   = SrcA > SrcB;
                            zero      = (ALUResult == 1'd0);
                        end
                    ////////////////////////////////////////////////////////////
                    4'b0100:        // SLL (Shift left logical)
                            ALUResult = SrcA << SrcB;
                    4'b1000:        // SRL (Shift right logical)
                            ALUResult = SrcA >> SrcB;
                    4'b1100:        // SRA (Shift right arithmetic)
                            ALUResult = $signed(SrcA) >>> SrcB;
                    ////////////////////////////////////////////////////////////
                    4'b0101:        // SLTU (Set on less than) (unsigned)
                            ALUResult = ($unsigned(SrcA)<$unsigned(SrcB));
                    4'b1010:        // SLT  (Set on less than) (signed)
                            ALUResult = ($signed(SrcA)<$signed(SrcB));
                    ////////////////////////////////////////////////////////////
                    default:        // Default
                            ALUResult = 'b0;
                    ////////////////////////////////////////////////////////////
                endcase
            end
endmodule
```

Data Memory (datamemory.sv)

```
`timescale 1ns / 1ps

module datamemory#(
    parameter DM_ADDRESS = 9 , // 9 bit address bus
    parameter DATA_W = 32      // 32 bit data bus
    )
    (
    input logic clk,           // Clock
     input logic MemRead ,      // Read signal from the control unit
    input logic MemWrite ,     // Write signal from the control unit

    input  logic [DM_ADDRESS-1:0] a ,   // Address bus (Read/Write address) = 9 LSB bits of the ALU output
    input  logic [DATA_W-1:0] wd    ,   // Write data bus
    output logic [DATA_W-1:0] rd        // Read data bus
    );

    logic [DATA_W-1:0] mem [(2**DM_ADDRESS)-1:0];     // Data memory array with capacity of 512 words with
each word of size 32 bits

    // Data memory will be initialized with the following values
    initial
        begin
            mem[0]  = 32'h00000000;  // 0
            mem[1]  = 32'h00000000;  // 0
            mem[2]  = 32'h00000000;  // 0
            mem[3]  = 32'h00000000;  // 0
            mem[4]  = 32'h00000005;  // 5
            mem[5]  = 32'h00000000;  // 0
            mem[6]  = 32'h00000000;  // 0
            mem[7]  = 32'h00000000;  // 0
            mem[8]  = 32'hfffffff7;  // -9
            mem[9]  = 32'h00000000;  // 0
            mem[10] = 32'h00000000;  // 0
            mem[11] = 32'h00000000;  // 0
            mem[12] = 32'h0000000C;  // 12
            mem[13] = 32'h0000000E;  // 14
            mem[14] = 32'hFFFFFFEE;  // -18
            mem[15] = 32'h00000000;  // 0
            mem[16] = 32'h00000000;  // 0
            mem[17] = 32'h00000000;  // 0
            mem[18] = 32'h00000000;  // 0
            mem[19] = 32'h00000000;  // 0
        end

    always @(*)
        begin
            if (MemRead)       // Whenever MemRead is high, the data at mem[a] is read and stored in rd
(Level Triggered)
                rd = mem[a];
        end

    always @(posedge clk)      // At postive edge of the clock, if MemWrite is high, wd is wrriten to mem[a]
(Edge Triggered)
        begin
            if (MemWrite)
                mem[a] = wd;
        end

endmodule
```

Instruction Memory (instructionmemory.sv)

```
`timescale 1ns / 1ps

module instructionmemory#(
    parameter INS_ADDRESS = 9,  // 9 bit address bus
    parameter INS_W = 32        // 32 bit data bus
    )
    (
    input  logic [INS_ADDRESS-1:0] ra ,   // Read address of the instruction memory (Comes from PC)
    output logic [INS_W-1:0] rd           // Data output of the instruction memory (Instruction)
    );

    logic [INS_W-1 :0] Inst_mem [(2**(INS_ADDRESS-2))-1:0];

    ////////////////////////////////////////////////////////////////////////////////////////////////////
//

    // Test Program 1 (R-Type Computational / Register-Register)
    /*
    assign Inst_mem[0]   = 32'h00007033;                          // NOP
    assign Inst_mem[1]   = 32'h00007033;                          // NOP
    assign Inst_mem[2]   = 32'h00007033;                          // NOP
    assign Inst_mem[3]   = 32'b00000000110100000000001010010011;  // ADDI x5, x0, 13
    assign Inst_mem[4]   = 32'b00000001011100000000001100010011;  // ADDI x6, x0, 23
    assign Inst_mem[5]   = 32'b00000000001100010100010100110011;  // ADD x18, x5, x6
    assign Inst_mem[6]   = 32'b01000000001100010100010110110011;  // SUB x19, x5, x6
    assign Inst_mem[7]   = 32'b00000000001100010111010100110011;  // OR  x20, x5, x6
    assign Inst_mem[8]   = 32'b00000000001100010111110101010011;  // AND x21, x5, x6
    */

    // Test Program 2 (I-Type Computational / Short Immediate Operands)
    /*
    assign Inst_mem[0]   = 32'h00007033;                          // NOP
    assign Inst_mem[1]   = 32'h00007033;                          // NOP
    assign Inst_mem[2]   = 32'h00007033;                          // NOP
    assign Inst_mem[3]   = 32'b00000001011000000000100100010011;  // ADDI x18,  x0, 22
    assign Inst_mem[4]   = 32'b00000000000111001000100110010011;  // ADDI x19, x18,  3
    assign Inst_mem[5]   = 32'b00000000000111001010010100010011;  // XORI x20, x18,  3
    assign Inst_mem[6]   = 32'b00000000000111001011010101010011;  // ORI  x21, x18,  3
    assign Inst_mem[7]   = 32'b00000000000111001011111100010011;  // ANDI x22, x18,  3
    assign Inst_mem[8]   = 32'b00000000000111001000110110010011;  // SLLI x23, x18,  3
    assign Inst_mem[9]   = 32'b00000000000111001001010000010011;  // SLTI x24, x18,  3
    */

    // Test Program 3 (S-Type Memory Access / Stores)
    /*
    assign Inst_mem[0]   = 32'h00007033;                          // NOP
    assign Inst_mem[1]   = 32'h00007033;                          // NOP
    assign Inst_mem[2]   = 32'h00007033;                          // NOP
    assign Inst_mem[3]   = 32'b00100000010100000000001100010011;  // ADDI x6, x0, 517
    assign Inst_mem[4]   = 32'b00000000011000000010001010100011;  // SW x6, 5(x0)
    assign Inst_mem[5]   = 32'b00000000011000000000001100100011;  // SB x6, 6(x0)
    assign Inst_mem[6]   = 32'b00000000011000000001001110100011;  // SH x6, 7(x0)
    */

    // Test Program 4 (I-Type Memory Access / Loads)
    /*
    assign Inst_mem[0]   = 32'h00007033;                          // NOP
    assign Inst_mem[1]   = 32'h00007033;                          // NOP
    assign Inst_mem[2]   = 32'h00007033;                          // NOP
    assign Inst_mem[3]   = 32'b00000000100000000000100100000011;  // LB  x18, 8(x0)
    assign Inst_mem[4]   = 32'b00000000100000000100100110000011;  // LBU x19, 8(x0)
    assign Inst_mem[5]   = 32'b00000000100000000001101000000011;  // LH  x20, 8(x0)
    assign Inst_mem[6]   = 32'b00000000100000000101101010000011;  // LHU x21, 8(x0)
    assign Inst_mem[7]   = 32'b00000000100000000010101100000011;  // LW  x22, 8(x0)
    */
```

```
    // Test Program 5 (SB-Type Control Transfer / Conditional Branches)
    /*
    assign Inst_mem[0]    = 32'h00007033;                          // NOP
    assign Inst_mem[1]    = 32'h00007033;                          // NOP
    assign Inst_mem[2]    = 32'h00007033;                          // NOP
    assign Inst_mem[3]    = 32'b00000000110000000010111000000011;  // LW x28, 12(x0)
    assign Inst_mem[4]    = 32'b00000000110100000010111010000011;  // LW x29, 13(x0)
    assign Inst_mem[5]    = 32'b00000000111000000010111100000011;  // LW x30, 14(x0)
    assign Inst_mem[6]    = 32'b00000000110000000010111110000011;  // LW x31, 12(x0)
    assign Inst_mem[7]    = 32'b00000000111011110000001001100011;  // BEQ x28, x29, 12
    assign Inst_mem[8]    = 32'b00000000101010000000000100100010011;  // ADDI x18, x0, 21
    assign Inst_mem[9]    = 32'b00000000111111110000001001100011;  // BEQ x28, x31, 12
    assign Inst_mem[10]   = 32'b00000000101110000000000100110010011;  // ADDI x19, x0, 23
    assign Inst_mem[11]   = 32'b00000000110010000000000101000010011;  // ADDI x20, x0, 25
    assign Inst_mem[12]   = 32'b00000000110110000000000101010010011;  // ADDI x21, x0, 27
    assign Inst_mem[13]   = 32'b00000000111011110000101100100011;  // BNE x28, x29, 12
    assign Inst_mem[14]   = 32'b00000000111010000000000101100010011;  // ADDI x22, x0, 29
    assign Inst_mem[15]   = 32'b00000000111110000000000101110010011;  // ADDI x23, x0, 31
    assign Inst_mem[16]   = 32'b00000001000010000000000110000010011;  // ADDI x24, x0, 33
    assign Inst_mem[17]   = 32'b00000000111101110010101000100011;  // BGE x28, x30, 8
    assign Inst_mem[18]   = 32'b00000001000110000000000110010010011;  // ADDI x25, x0, 35
    assign Inst_mem[19]   = 32'b00000001001010000000000110100010011;  // ADDI x26, x0, 37
    assign Inst_mem[20]   = 32'b00000000111101110011101000100011;  // BGEU x28, x30, 8
    assign Inst_mem[21]   = 32'b00000001001110000000000110110010011;  // ADDI x27, x0, 39
    */

    // Test Program 6 (Adding 2 numbers and saving the result to memory)
    /*
    assign Inst_mem[0]   = 32'h00007033;                          // NOP
    assign Inst_mem[1]   = 32'h00007033;                          // NOP
    assign Inst_mem[2]   = 32'h00007033;                          // NOP
    assign Inst_mem[3]   = 32'b00000000000110000000000001010010011;  // ADDI R5, R0, 3
    assign Inst_mem[4]   = 32'b00000000101100000000000001100010011;  // ADDI R6, R0, 11
    assign Inst_mem[5]   = 32'b00000000001100010100000010110011;  // ADD R1, R5, R6
    assign Inst_mem[6]   = 32'b00000000000010000001001000100011;  // SW R1, 8, R0
    */

    // Test Program 7 (Program that reads the value of n from memory and computes the sum of numbers from 1
to n and stores the sum in memory)
    /**/
    assign Inst_mem[0]   = 32'h00007033;                          // NOP
    assign Inst_mem[1]   = 32'h00007033;                          // NOP
    assign Inst_mem[2]   = 32'h00007033;                          // NOP
    assign Inst_mem[3]   = 32'b00000000110000000010100100000011;  // LW x18, 12(x0)
    assign Inst_mem[4]   = 32'b00000000000010000000000100110010011;  // ADDI x19, x0, 1
    assign Inst_mem[5]   = 32'b00000000000001010011111010000010011;  // ANDI x20, x20, 0
    assign Inst_mem[6]   = 32'b00000001001010100000101000110011;  // ADD x20, x20, x18
    assign Inst_mem[7]   = 32'b00000000110000000010101010000011;  // LW x21, 12(x0)
    assign Inst_mem[8]   = 32'b01000001001110101000101010110011;  // SUB x21, x21, x19
    assign Inst_mem[9]   = 32'b00000001010110101000001010001100011;  // ADD x20, x20, x21
    assign Inst_mem[10]  = 32'b11111111001110101001110011100011;  // BNE x21, x19, -8
    assign Inst_mem[11]  = 32'b00000001010000000001001011010011;  // SW x20, 11(x0)
    /**/

    ////////////////////////////////////////////////////////////////////////////////////////////////////////////
//

    // PC is incremented by 4, but our instruction memory is word addressable.
    // So, we need to divide the PC value by 4 to get the correct address.
    // So, the 2 LSBs of the PC are ignored.
    assign rd =  Inst_mem [ra[INS_ADDRESS-1:2]];

endmodule
```

Register File (RegFile.sv)

```
`timescale 1ns / 1ps

module RegFile#(
    parameter DATA_WIDTH = 32,    // Number of bits in each register
    parameter ADDRESS_WIDTH = 5,  // Number of registers = 2^ADDRESS_WIDTH
    parameter NUM_REGS = 32       // Number of registers
    )
    (
    input  clk,         // Clock signal
    input  rst,         // Synchronous reset; if it is asserted (rst=1), all registers are reseted to 0 at a falling clock edge
    input  rg_wrt_en,   // Write enable signal

    input  logic [ADDRESS_WIDTH-1:0] rg_wrt_dest,  // Address of Write Register
    input  logic [ADDRESS_WIDTH-1:0] rg_rd_addr1,  // Address of Read Register 1
    input  logic [ADDRESS_WIDTH-1:0] rg_rd_addr2,  // Address of Read Register 2

    input  logic [DATA_WIDTH-1:0] rg_wrt_data,  // Write Data Port  (Data to be written to reg_file[rg_wrt_dest])
    output logic [DATA_WIDTH-1:0] rg_rd_data1,  // Read Data Port 1 (Content of reg_file[rg_rd_addr1] is given to the output port rg_rd_data1)
    output logic [DATA_WIDTH-1:0] rg_rd_data2   // Read Data Port 2 (Content of reg_file[rg_rd_addr2] is given to the output port rg_rd_data2)
    );

    logic [DATA_WIDTH-1:0] register_file [NUM_REGS-1:0];   // Register File
    integer i;

    /*      Notes :-
            A Read can be done at any time (Combinational).
            A Write is performed at the falling clock edge if it is enabled (rg_wrt_en=1).      */

    assign rg_rd_data1 = register_file[rg_rd_addr1];  // Read rg_rd_addr1 from register_file and assign it to rg_rd_data1
    assign rg_rd_data2 = register_file[rg_rd_addr2];  // Read rg_rd_addr2 from register_file and assign it to rg_rd_data2

    always @(negedge clk)
        begin
            if (rst==1'b1)
                for (i=0;i<NUM_REGS;i=i+1)
                    register_file[i] <= 0;                // Reset all registers to 0
            else if (rst==1'b0 && rg_wrt_en==1'b1)
                register_file[rg_wrt_dest] <=rg_wrt_data;  // Write rg_wrt_data to register_file[rg_wrt_dest]
        end

endmodule
```

Immediate Generator (imm_Gen.sv)

```
`timescale 1ns / 1ps

module imm_Gen
(
    input  logic [31:0] inst_code, // instruction code
    output logic [31:0] Imm_out);  // immediate operand

    logic [4:0] srai;
    assign srai = inst_code[24:20];

    /* Takes the instruction code as input, performs various operations based on the instruction code and
    generate the immediate value as output by concatenating the various bits of the instruction code.  */
    always_comb
        case(inst_code[6:0])
            7'b0000011 : /*I-type Memory Access     */
                Imm_out = {inst_code[31]? {20{1'b1}}:20'b0 , inst_code[31:20]};
            7'b0010011 : /*I-type Computational     */
                begin
                    if ((inst_code[31:25]==7'b0100000&&inst_code[14:12]==3'b101)||(inst_code[14:12]==3'b001)||inst_code[14:12]==3'b101)
                        Imm_out = {srai[4]? {27{1'b1}}:27'b0,srai};
                    else
                        Imm_out = {inst_code[31]? 20'b1:20'b0 , inst_code[31:20]};
                end
            7'b0100011 : /*S-type Memory Access     */
                Imm_out = {inst_code[31]? 20'b1:20'b0 , inst_code[31:25], inst_code[11:7]};
            7'b1100011 : /*SB-type Control Transfer */
                Imm_out = {inst_code[31]? 20'b1:20'b0 , inst_code[7], inst_code[30:25],inst_code[11:8],1'b0};
            7'b1100111 : /*I-JALR  (Unique Opcode)  */
                Imm_out = {inst_code[31]? 20'b1:20'b0 , inst_code[30:25], inst_code[24:21], inst_code[20]};
            7'b0110111 : /*U-LUI   (Unique Opcode)  */
                Imm_out = {inst_code[31:12], 12'b0};
            7'b0010111 : /*U-AUIPC (Unique Opcode)  */
                Imm_out = {inst_code[31:12], 12'b0};
            7'b1101111 : /*UJ-JAL  (Unique Opcode)  */
                Imm_out = {inst_code[31]? 12'b1:12'b0 , inst_code[19:12] ,inst_code[20], inst_code[30:25],inst_code[24:21],1'b0};
            default    :
                Imm_out = {32'b0};
        endcase
endmodule
```

Controller (Controller.sv)

```
`timescale 1ns / 1ps

module Controller     // Controller is designed using microprogrammed approach
(

    // Input to Controller is the 7-bit opcode from instruction
    input logic [6:0] Opcode,

    // Output from Controller are the control signals
    output logic ALUSrc,
    output logic MemtoReg,
    output logic RegtoMem,
    output logic RegWrite,
    output logic MemRead,
    output logic MemWrite,
    output logic Branch,
    output logic [1:0] ALUOp,
    output logic Con_Jalr, Con_Jal, Mem, OpI, Con_AUIPC, Con_LUI

);

    logic [14:0] rom_data [0:512];

    // Control bits for each instruction type
    logic [14:0] control_bits_for_R_TYPE   = 15'b000100010000000;
    logic [14:0] control_bits_for_LW       = 15'b110110000001000;
    logic [14:0] control_bits_for_SW       = 15'b101001000001000;
    logic [14:0] control_bits_for_RI_TYPE  = 15'b100100000000100;
    logic [14:0] control_bits_for_BR_TYPE  = 15'b000000101000000;
    logic [14:0] control_bits_for_JALR     = 15'b000101000100000;
    logic [14:0] control_bits_for_JAL      = 15'b000100000010000;
    logic [14:0] control_bits_for_AUIPC    = 15'b000000000000010;
    logic [14:0] control_bits_for_LUI      = 15'b000000000000001;

    always_comb
        begin
            rom_data[7'b0110011] <= control_bits_for_R_TYPE;
            rom_data[7'b0000011] <= control_bits_for_LW;
            rom_data[7'b0100011] <= control_bits_for_SW;
            rom_data[7'b0010011] <= control_bits_for_RI_TYPE;
            rom_data[7'b1100011] <= control_bits_for_BR_TYPE;
            rom_data[7'b1100111] <= control_bits_for_JALR;
            rom_data[7'b1101111] <= control_bits_for_JAL;
            rom_data[7'b0010111] <= control_bits_for_AUIPC;
            rom_data[7'b0110111] <= control_bits_for_LUI;
        end

    assign ALUSrc    =  rom_data[Opcode][14];
    assign MemtoReg  =  rom_data[Opcode][13];
    assign RegtoMem  =  rom_data[Opcode][12];
    assign RegWrite  =  rom_data[Opcode][11];
    assign MemRead   =  rom_data[Opcode][10];
    assign MemWrite  =  rom_data[Opcode][9];
    assign Branch    =  rom_data[Opcode][8];
    assign ALUOp     =  rom_data[Opcode][7:6];
    assign Con_Jalr  =  rom_data[Opcode][5];
    assign Con_Jal   =  rom_data[Opcode][4];
    assign Mem       =  rom_data[Opcode][3];
    assign OpI       =  rom_data[Opcode][2];
    assign Con_AUIPC =  rom_data[Opcode][1];
    assign Con_LUI   =  rom_data[Opcode][0];

endmodule
```

ALU Controller (ALUController.sv)

```
`timescale 1ns / 1ps

module ALUController
(

    input logic [1:0] ALUOp,
    input logic [6:0] Funct7,
    input logic [2:0] Funct3,
    input Branch,
    input Mem,OpI,AUIPC,

    output logic [3:0] Operation,   // Control signal that determines the ALU operation
    output logic Con_beq, Con_bnq, Con_blt, Con_bgt

);

    assign Con_beq = (Branch)&&(Funct3==3'b000);
    assign Con_bnq = (Branch)&&(Funct3==3'b001);
    assign Con_blt = (Branch)&&(Funct3==3'b100)||(Funct3==3'b110);
    assign Con_bgt = (Branch)&&(Funct3==3'b101)||(Funct3==3'b111);

    assign Operation[0] = ((ALUOp[1]==1'b1) && (Funct7==7'b0000000) && ((Funct3==3'b110) ||
(Funct3==3'b100)||(Funct3==3'b011)))||(Branch&&(Funct3==3'b110||Funct3==3'b111))||(OpI&&(Funct3==3'b100))||(OpI&&(Funct3==
3'b110)) || (OpI&&Funct3==3'b011);

    assign Operation[1] = (ALUOp==2'b00&&(!OpI)) || ((ALUOp[1]==1'b1) && (Funct7==7'b0000000) &&
((Funct3==3'b000)||(Funct3==3'b100)||(Funct3==3'b010)))||((ALUOp[1]==1'b1) && (Funct7==7'b0100000) && (Funct3==3'b000)) ||
(ALUOp[0]==1'b1) || Mem || (OpI&&(Funct3==3'b100)) || (OpI&&(Funct3==3'b000)) || (OpI&&Funct3==3'b010);

    assign Operation[2] = (ALUOp[0]==1'b1) || ((ALUOp[1]==1'b1) && (Funct7==7'b0100000) &&
(Funct3==3'b000))||((ALUOp[1]==1'b1) && (Funct7==7'b0000000) && ((Funct3==3'b001)||(Funct3==3'b011))) ||(OpI&&Funct7 ==
7'b0100000&&Funct3 == 3'b101) || (ALUOp[1]&&Funct7 == 7'b0100000&&Funct3 == 3'b101) || (OpI&&Funct3 == 3'b001) ||
(OpI&&Funct3==3'b011);

    assign Operation[3] = (Funct3 == 3'b010&&(!Mem)) || (OpI&&Funct3 == 3'b101&&Funct7 == 7'b0000000) ||
(ALUOp[1]&&(Funct3 == 3'b101)) || (OpI&&Funct7 == 7'b0100000&&Funct3 == 3'b101) || (ALUOp[1]&&Funct7 == 7'b0100000&&Funct3
== 3'b101);

endmodule
```

RISCV Data Path (Datapath.sv)

```
`timescale 1ns / 1ps

module Datapath#(

    parameter PC_W = 9,          // Program Counter Width
    parameter INS_W = 32,        // Instruction Width
    parameter RF_ADDRESS = 5,    // Register File Address
    parameter DATA_W = 32,       // Data WriteData
    parameter DM_ADDRESS = 9,    // Data Memory Address
    parameter ALU_CC_W = 4       // ALU Control Code Width

    )
    (

    // Control Signals used to control the datapath modules
    input logic
        clk ,                    // Global clock
        reset ,                  // Reset
        RegWrite ,               // RegFile write enable
        MemtoReg ,               // MUX Select : 1 = Load Data  , 0 = ALU Result
        RegtoMem,                // MUX Select : 1 = Store Data , 0 = RegFile ReadData2
        MemWrite ,               // Data Memory Write Enable
        MemRead ,                // Data Memory Read  Enable
        ALUsrc ,

    // Control signals used for branching, jumping, and ALU operations
    input logic Con_beq,         // Branch on equal condition signal.
    input logic Con_bnq,         // Branch on not equal condition signal.
    input logic Con_bgt,         // Branch on greater than condition signal.
```

```
    input logic Con_blt,         // Branch on less than condition signal.
    input logic Con_Jalr,        // Jump and link register instruction signal.
    input logic Jal,             // Jump and link instruction signal.
    input logic AUIPC, LUI,      // AUIPC and LUI instruction signals.
    input logic [ALU_CC_W-1:0] ALU_CC,     // ALU Control Code

    // Output signals that will carry information about the instruction being executed
    output logic [6:0] opcode,             // Instruction opcode
    output logic [6:0] Funct7,             // Instruction funct7
    output logic [2:0] Funct3,             // Instruction funct3

    // Result of the ALU operation
    output logic [31:0] ALU_Result         // Result of the ALU operation

    );

    // Defining the datapath signals (ports)
    logic [8:0]  PC, PCPlus4, PCValue, BranchPC;
    logic [31:0] Instr, PCPlusImm, PCJalr, LD,ST, Store_data;
    logic [31:0] Result;
    logic [31:0] Reg1, Reg2;
    logic [31:0] ReadData;
    logic [31:0] SrcB, ALUResult;
    logic [31:0] ExtImm;
    logic [31:0] PC_unsign_extend;
    logic [31:0] Read_Alu_Result, Jal_test,aui_data, lui_data;
    logic [1:0] PCSel;
    logic zero, Con_BLT, Con_BGT, Jalr, Branch;

    assign PC_unsign_extend = {23'b0, PC};
    assign Branch = (Con_beq&&zero)||(Con_bnq&&!zero)||(Con_bgt&&Con_BGT)||(Con_blt&&Con_BLT)||Jal;
    assign Jalr = Con_Jalr;

    // Extracting the instruction opcode, funct7, and funct3 from the instruction
    assign opcode = Instr[6:0];
    assign Funct7 = Instr[31:25];
    assign Funct3 = Instr[14:12];

    // PCPlus4 = PC + 4                          (9 bit operation)
    adder #(9) pcadd1 (PC, 9'b100, PCPlus4);

    // PCPusImm = PC_unsign_extend + ExtImm      (32 bit operation)
    adder_32 #(32) pcadd2 (PC_unsign_extend, ExtImm, PCPlusImm);

    // PCJalr = ExtImm + Reg1                    (32 bit operation)
    adder_32 #(32) pcadd3 (ExtImm, Reg1, PCJalr);

    // BranchPC = Branch ? PCPlusImm[8:0] : PCPlus4
    mux2 next_pc1(PCPlus4,PCPlusImm[8:0], Branch, BranchPC);

    // PCValue = Jalr ? PCJalr[8:0] : BranchPC
    mux2 next_pc2(BranchPC, PCJalr[8:0], Jalr, PCValue);

    // At positive clock edges, PC is updated with the new PCValue
    flopr #(9) pcreg(clk, reset, PCValue, PC);

    // Give PC as input to the instr_mem and get Instr(Instruction) as output
    instructionmemory instr_mem (PC, Instr);

    // Use Instr(Instruction from instruction memory) and Reg2(Data from data memory) to extract the information and
assign it to ST
    data_extract store_data_ex(Instr, Reg2, ST);

    // Store_data = RegtoMem ? ST : Reg2
    mux2 #(32) resmux_store(Reg2, ST, RegtoMem, Store_data);

    /*      Define register file (rf)
    clk = Clock Signal
    reset = Reset Signal
    RegWrite = Write Enable Signal
    Instr[11:7]  = Address of Write Register   (rd)
    Instr[19:15] = Address of Read Register 1  (rs1)
    Instr[24:20] = Address of Read Register 2  (rs2)
    Result = Write Data Port  (Data to be stored in rd)
    Reg1   = Read Data Port 1 (Gets value of rs1)
```

```
    Reg2    = Read Data Port 2 (Gets value of rs2)        */
    RegFile rf(clk, reset, RegWrite, Instr[11:7], Instr[19:15], Instr[24:20], Result, Reg1, Reg2);

    // Use Instr(Instruction from instruction memory) and ReadData(Data from data memory) to extract the information and
assign it to LD
    data_extract load_data_ex(Instr, ReadData, LD);

    // Read_ALU_Result = MemtoReg ? LD : ALUResult
    mux2 #(32) resmux(ALUResult, LD, MemtoReg, Read_Alu_Result);

    // Jal_test = (Jal OR Jalr) ? {23'b0,PCPlus4} : Read_Alu_Result
    mux2 #(32) resmux_jal(Read_Alu_Result, {23'b0, PCPlus4}, (Jal||Jalr), Jal_test);

    // aui_data = AUIPC ? PCPlusImm : Jal_test
    mux2 #(32) resmux_auipc(Jal_test, PCPlusImm, AUIPC, aui_data);

    // Use Instr(Instruction) to generate ExtImm(Extended Immediate Operand)
    imm_Gen Ext_Imm (Instr,ExtImm);

    // SrcB = (ALUsrc OR Jal OR Jalr) ? ExtImm : Reg2
    mux2 #(32) srcbmux(Reg2, ExtImm, (ALUsrc||Jal||Jalr), SrcB);

    /*   Define ALU (alu_module)
    SrcA       = Reg1
    SrcB       = SrcB
    Operation  = ALU_CC
    ALUResult  = ALUResult
    Con_BLT    = Con_BLT
    Con_BGT    = Con_BGT
    zero       = zero         */
    alu alu_module(Reg1, SrcB, ALU_CC, ALUResult, Con_BLT, Con_BGT, zero);

    // Result = LUI ? ExtImm : aui_data
    mux2 #(32) resmux_lui(aui_data, ExtImm, LUI, Result);

    assign ALU_Result = Result;

    /*      Define data memory module (data_mem)
    Clock Signal   = clk
    Memory Read Enable Signal  = MemRead
    Memory Write Enable Signal = MemWrite
    Address Bus    = ALUResult[8:0]
    WriteData Bus = Store_data
    ReadData Bus  = ReadData                  */
    datamemory data_mem (clk, MemRead, MemWrite, ALUResult[8:0], Store_data, ReadData);

endmodule
```

Data Extractor (data_extract.sv)

```
`timescale 1ns / 1ps

module data_extract#(
    parameter WIDTH = 32
    )
    (
    input  logic [WIDTH-1:0] inst,    // 32-bit input port that contains the instruction to be parsed
    input  logic [WIDTH-1:0] data,    // 32-bit input port that contains the data to be parsed
    output logic [WIDTH-1:0] y        // 32-bit output port that contains the extracted data
    );

    logic [31:0] Imm_out;    // 32-bit signal that contains the immediate value extracted from the instruction
    logic [15:0] s_bit;      // 16-bit signal that contains the signed bit of the extracted data
    logic [7:0]  e_bit;      //  8-bit signal that contains the exponent bit of the extracted data

    assign s_bit = data[15:0];
    assign e_bit = data[7:0];

    //always_comb
    always
        begin
```

```
            Imm_out = {inst[31]? {20{1'b1}}:{20{1'b0}}, inst[31:20]};

            if (inst[6:0] == 7'b0000011)     // Load instructions
                begin
                    if (inst[14:12] == 3'b000)              // LB
                        y = {e_bit[7]? {24{1'b1}}:{24{1'b0}}, e_bit};
                    else if (inst[14:12] == 3'b001)         // LH
                        y = {s_bit[15]? {16{1'b1}}:{16{1'b0}}, s_bit};
                    else if (inst[14:12] == 3'b100)         // LBU
                        y = {24'b0, e_bit};
                    else if (inst[14:12] == 3'b101)         // LHU
                        y = {16'b0, s_bit};
                    else if (inst[14:12] == 3'b010)         // LW
                        y = data;
                end

            else if(inst[6:0] == 7'b0100011) // Store instructions
                begin
                    if (inst[14:12] == 3'b000)              // SB
                        y = {e_bit[7]? {24{1'b1}}:{24{1'b0}}, e_bit};
                    else if (inst[14:12] == 3'b001)         // SH
                        y = {s_bit[15]? {16{1'b1}}:{16{1'b0}}, s_bit};
                    else if (inst[14:12] == 3'b010)         // SW
                        y = data;
                end

        end

endmodule
```

Adder (adder.sv)

```
`timescale 1ns / 1ps

module adder#(
    parameter WIDTH = 8
    )
    (
    input  logic [WIDTH-1:0] a, b,
    output logic [WIDTH-1:0] y
    );

    assign y = a + b;

endmodule
```

Adder 32bit (adder_32.sv)

```
`timescale 1ns / 1ps

module adder_32#(
    parameter WIDTH = 32
    )
    (
    input  logic [WIDTH-1:0] a, b,
    output logic [WIDTH-1:0] y
    );

    assign y = a + b;

endmodule
```

2 input Multiplexer (mux2.sv)

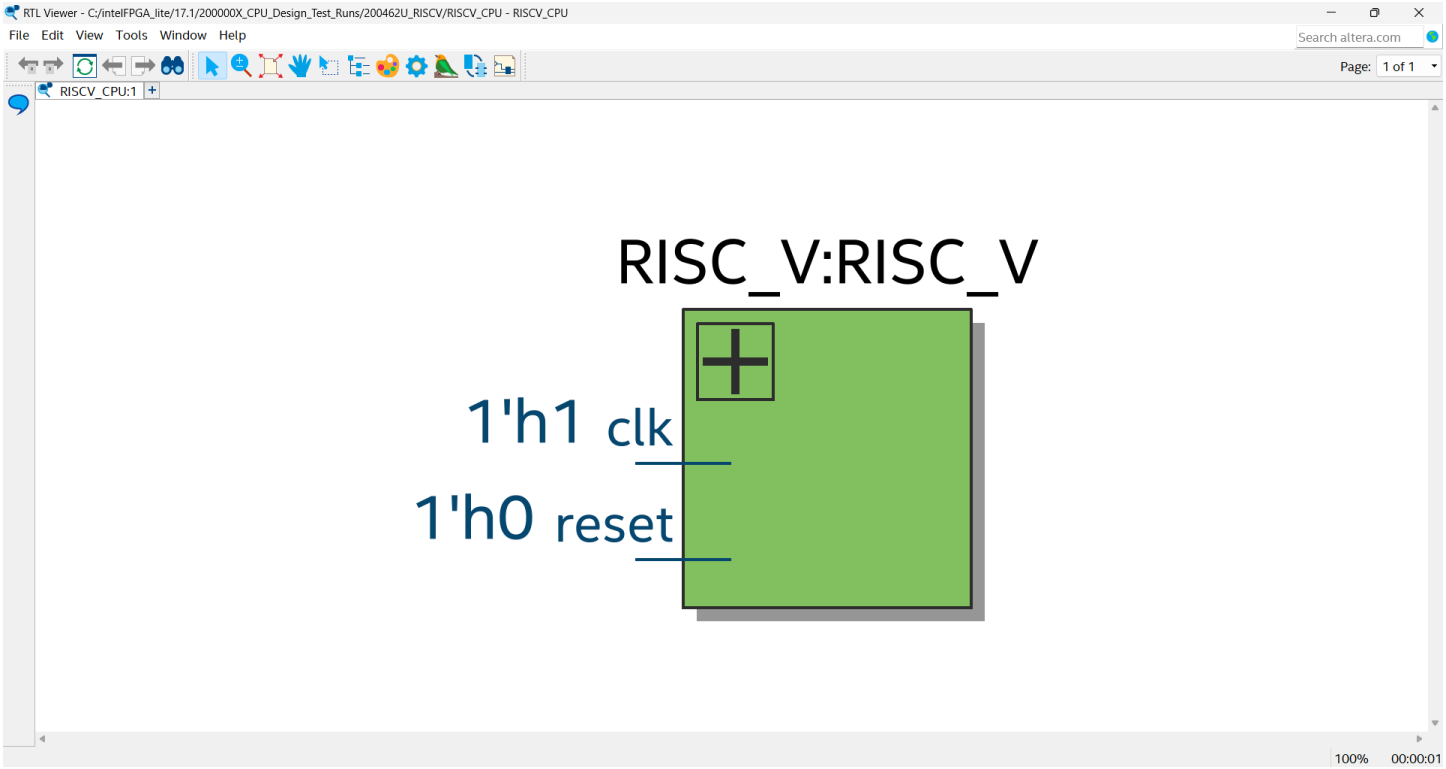
```
`timescale 1ns / 1ps

module mux2#(
    parameter WIDTH = 9
    )
    (
    input  logic [WIDTH-1:0] d0, d1,  // Mux data inputs
    input  logic s,                   // Mux select input
    output logic [WIDTH-1:0] y        // Mux data output
    );

    assign y = s ? d1 : d0;

endmodule
```

D flip-flop (flopr.sv)

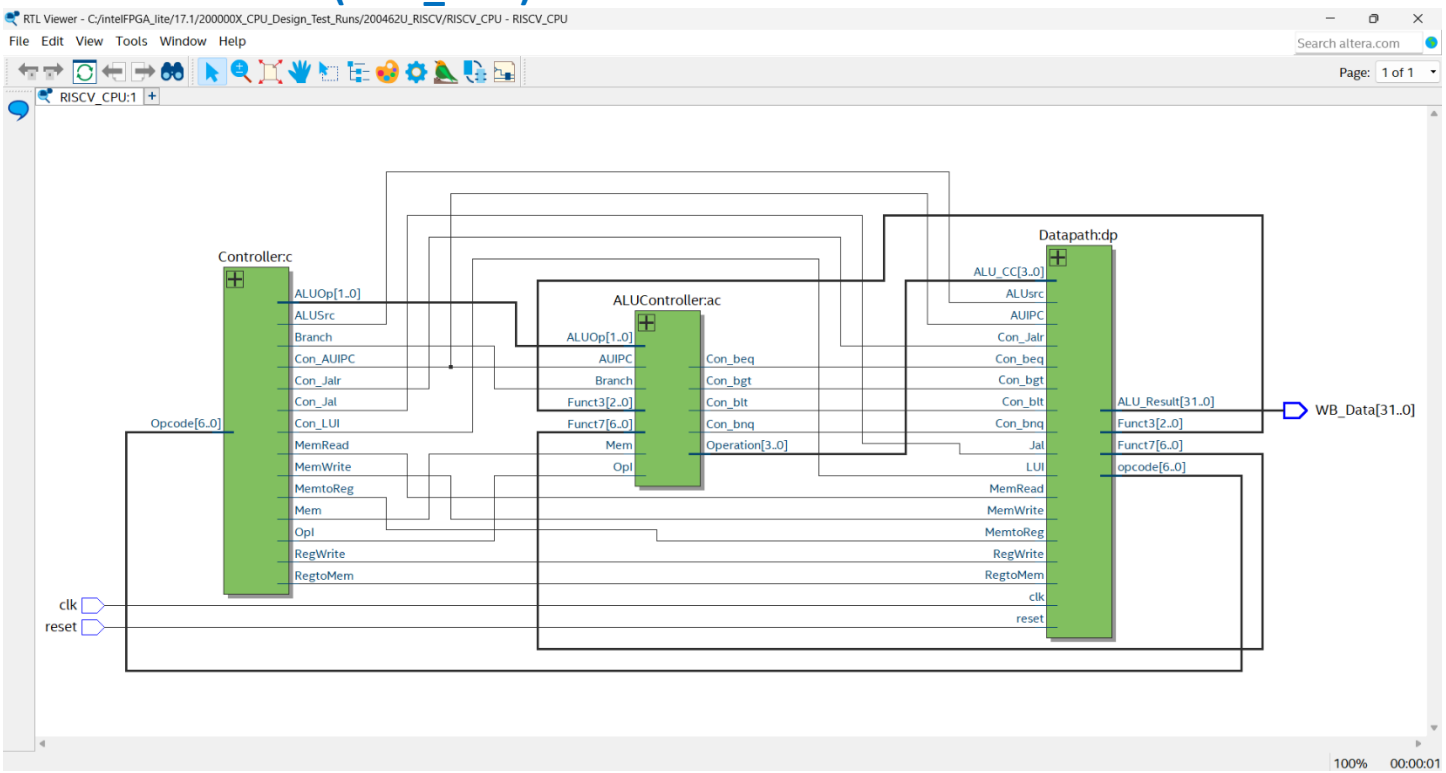
```
`timescale 1ns / 1ps

module flopr#(
    parameter WIDTH = 8
    )
    (
    input  logic clk, reset,
    input  logic [WIDTH-1:0] d,
    output logic [WIDTH-1:0] q
    );

    always_ff @(posedge clk, posedge reset)
        if (reset)
            q <= 0;
        else
            q <= d;

endmodule
```

Appendix B: Gate Level Schematics of Modules

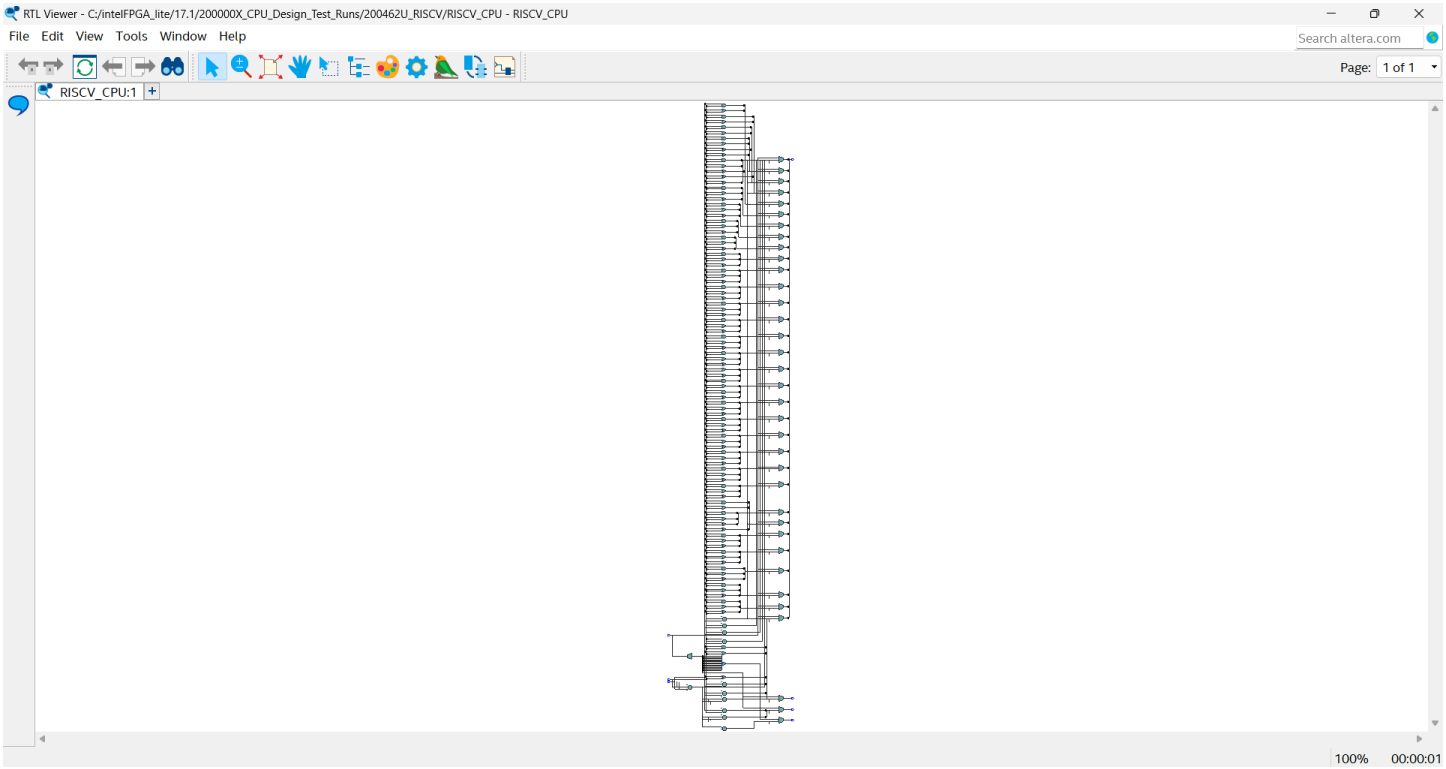
Test Bench Module (tb_top.sv)



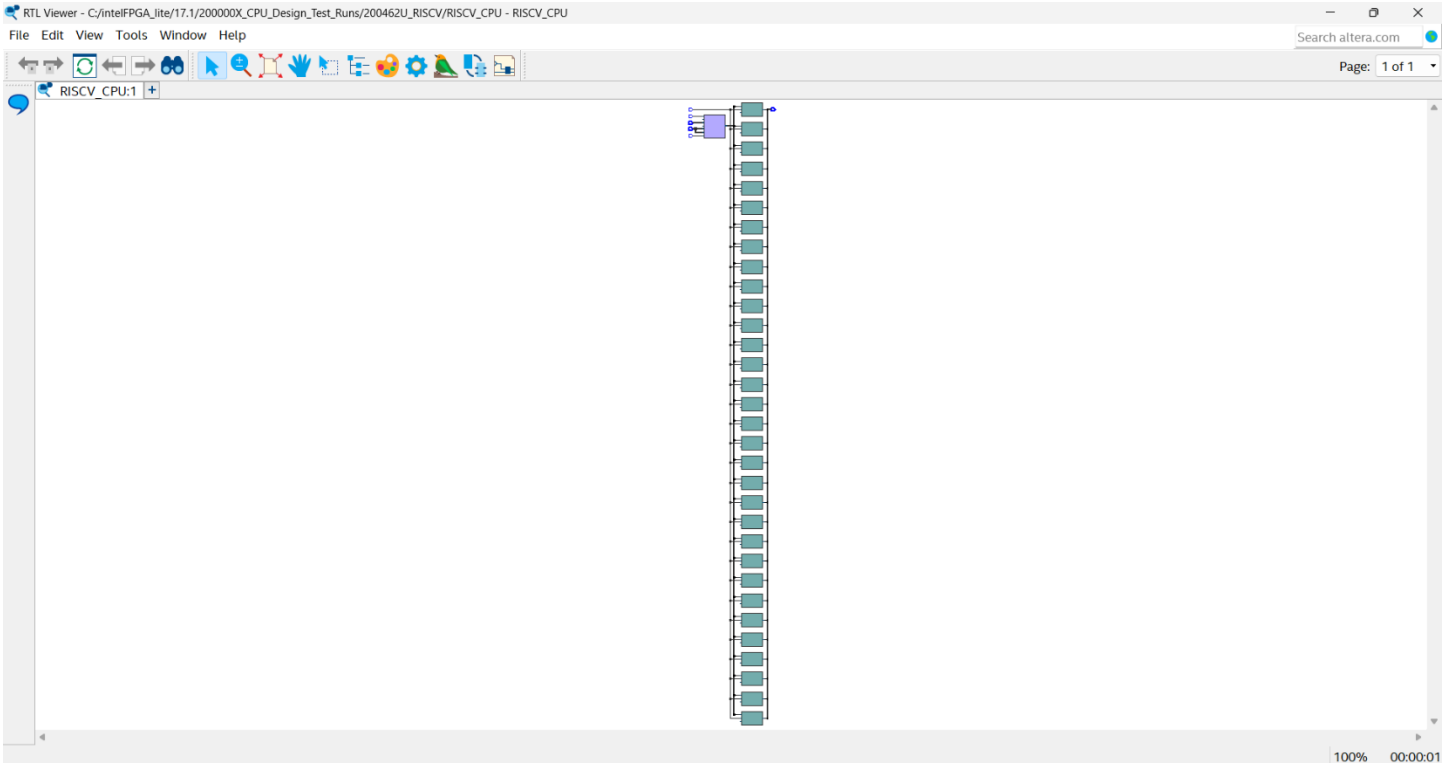
Processor Module (RISC_V.sv)



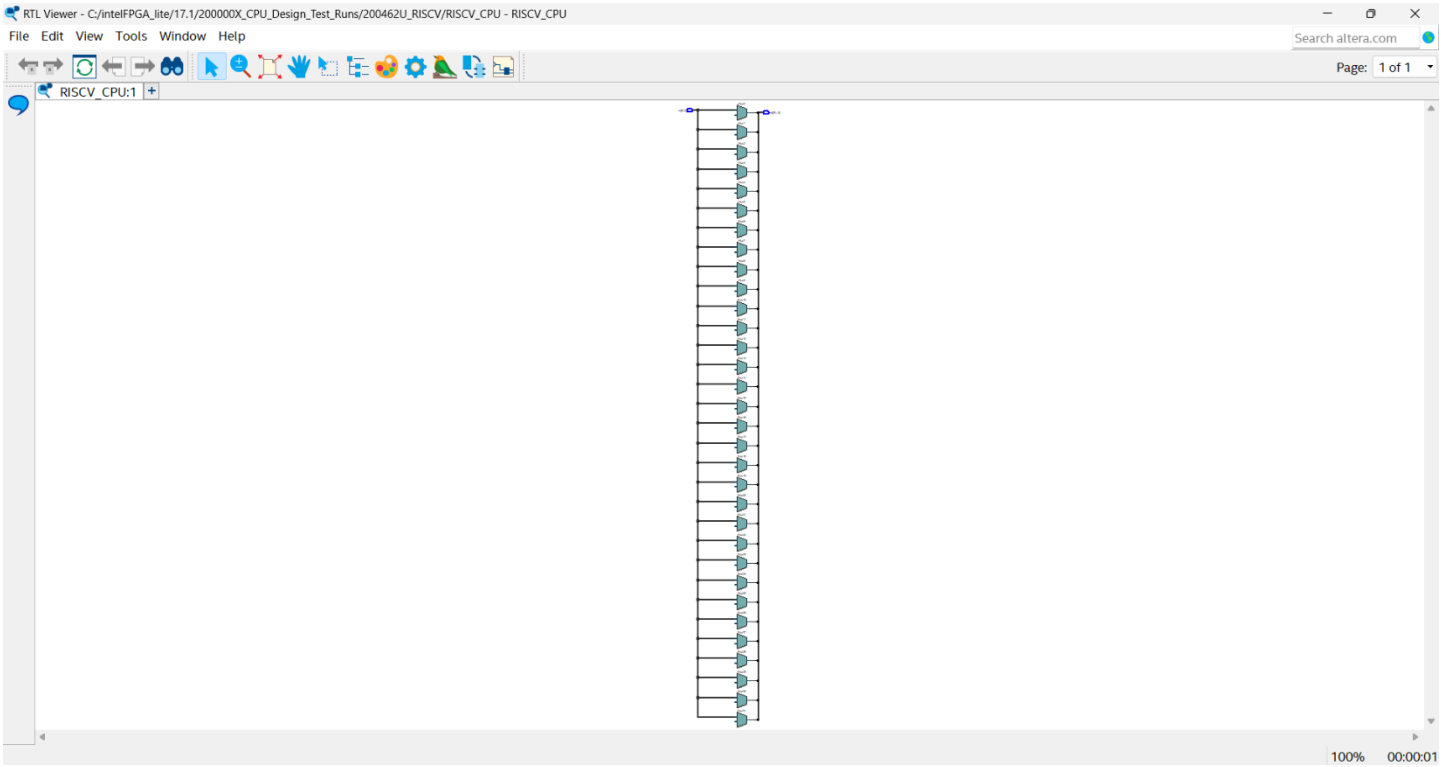
ALU (alu.sv)



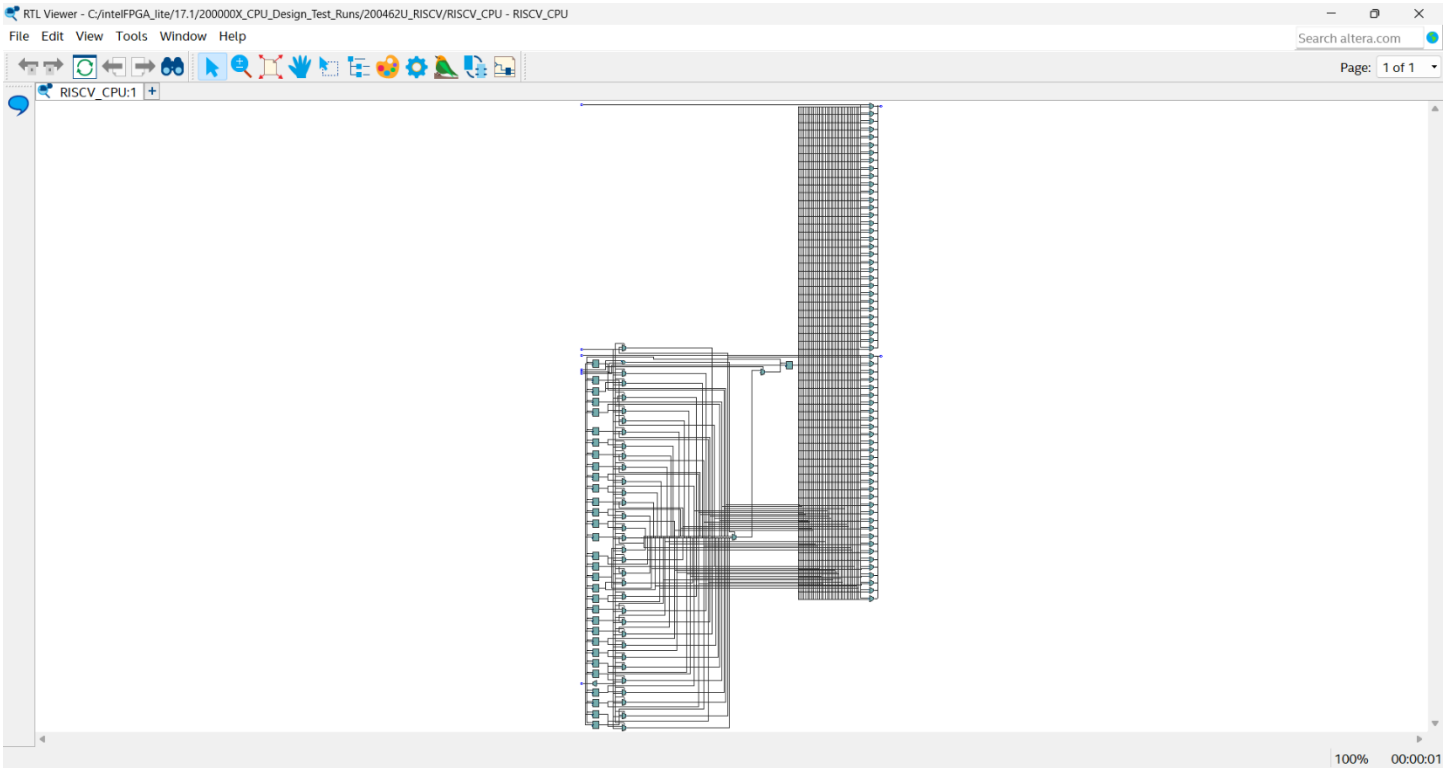
Data Memory (datamemory.sv)



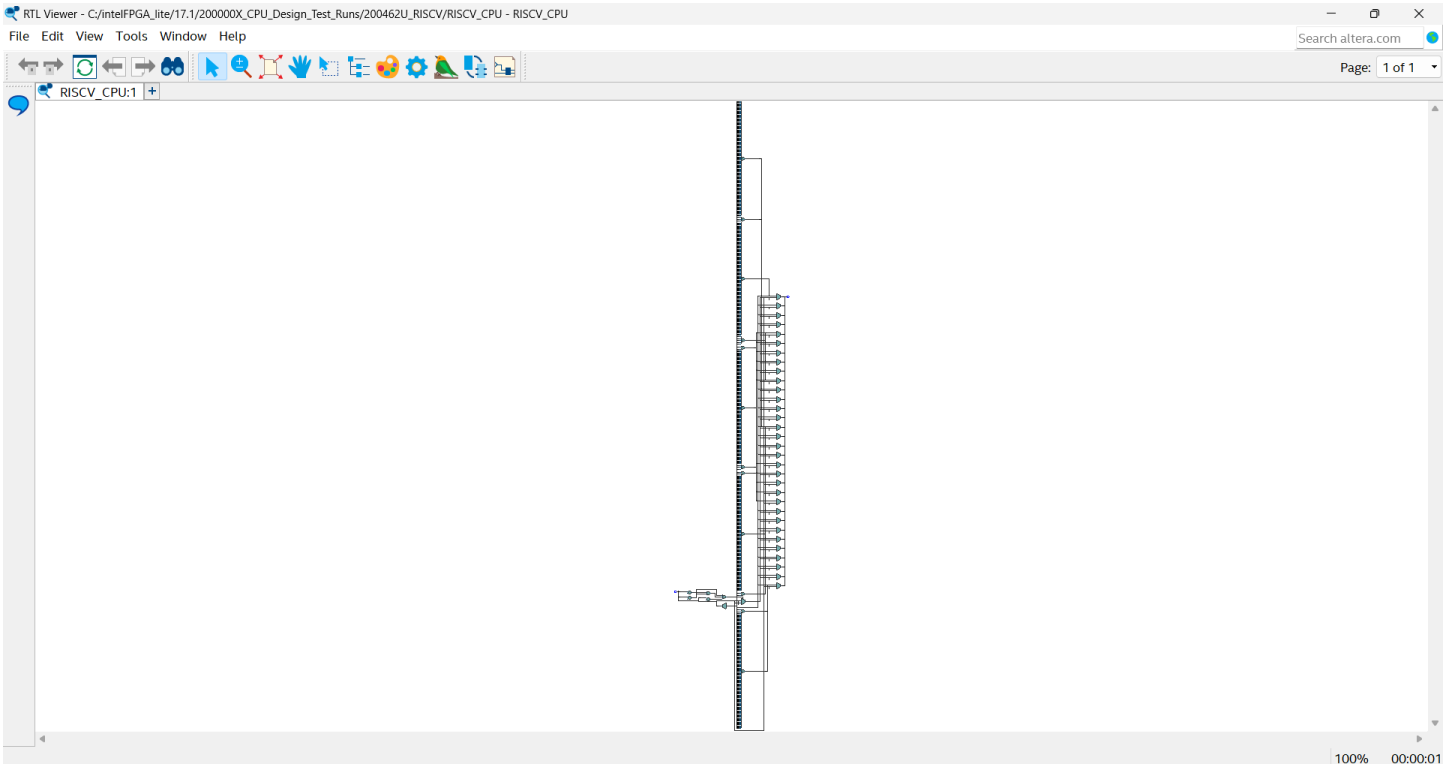
Instruction Memory (instructionmemory.sv)



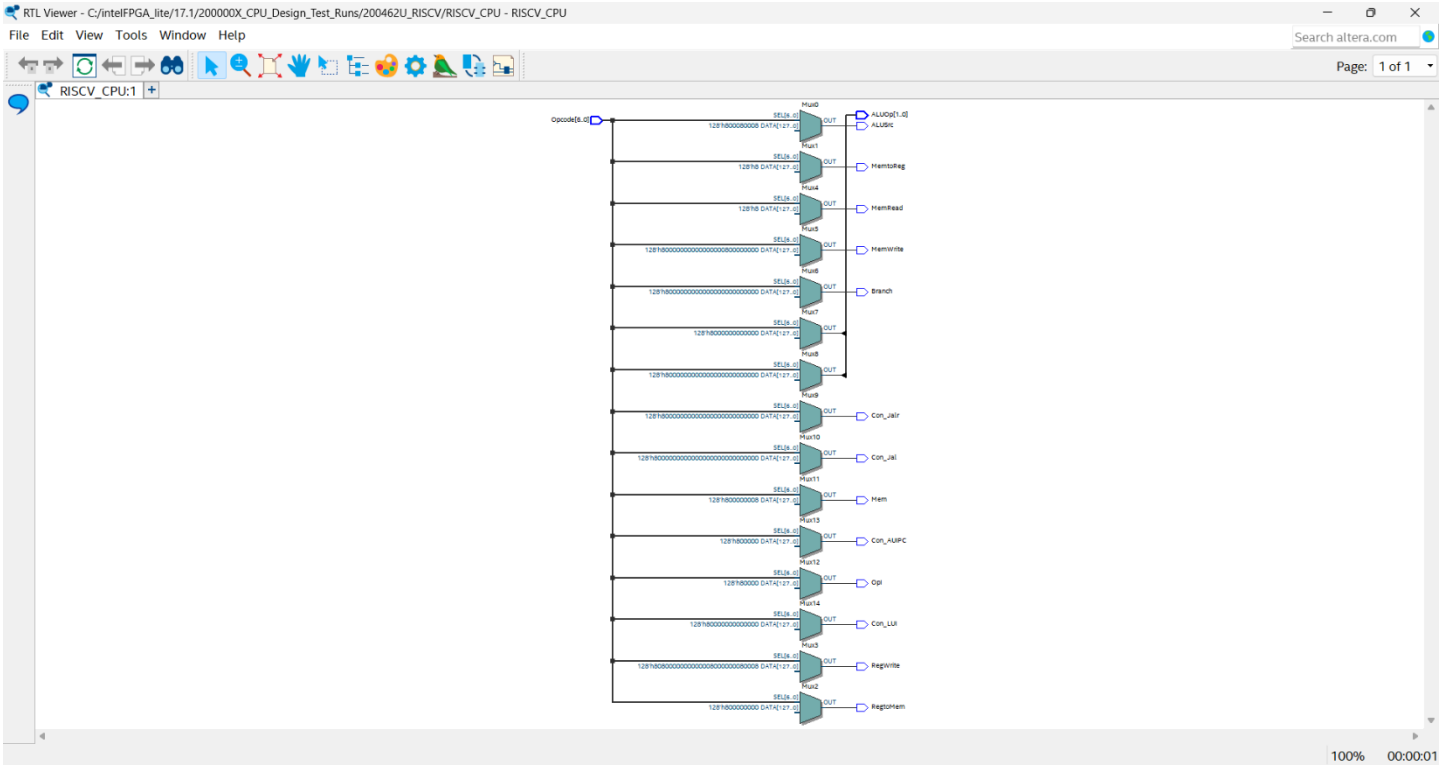
Register File (RegFile.sv)



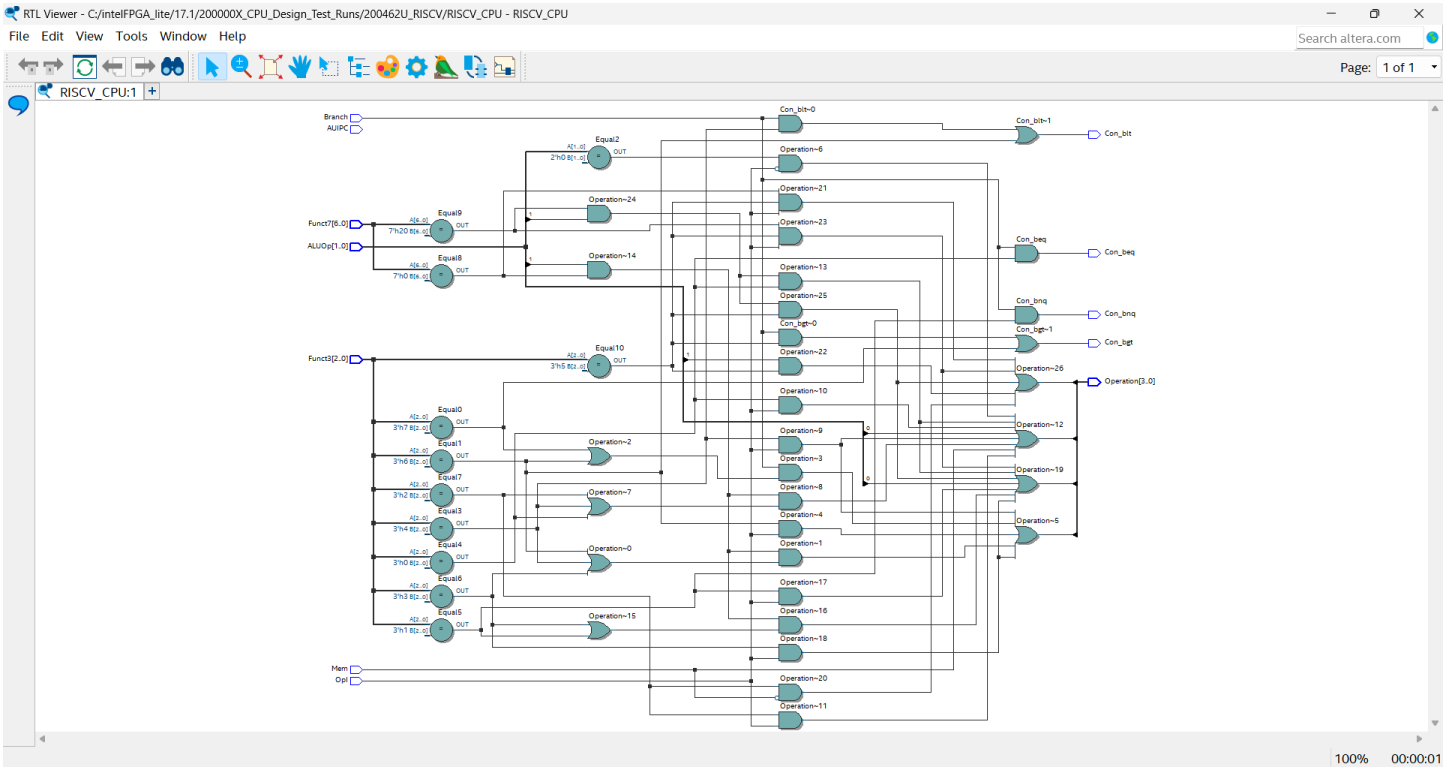
Immediate Generator (imm_Gen.sv)



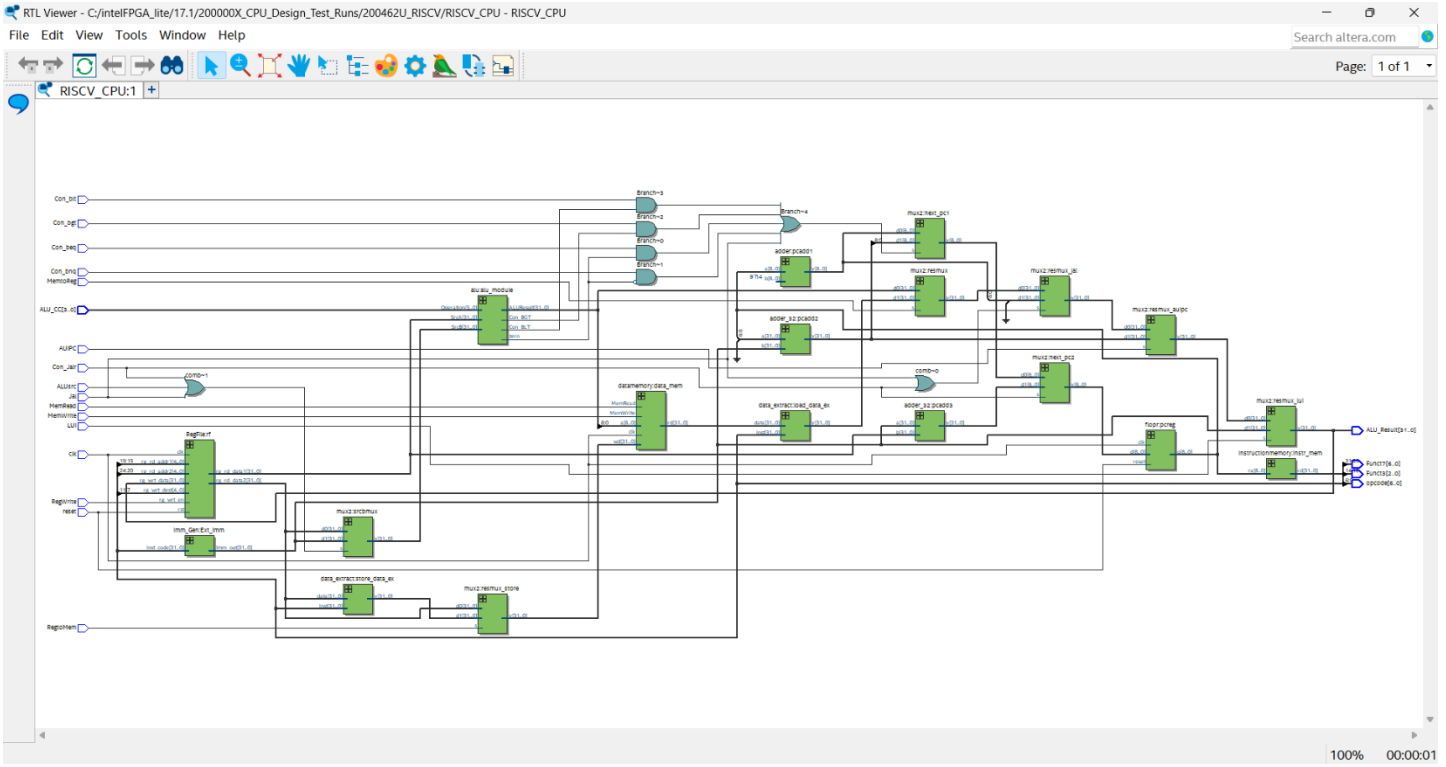
Controller (Controller.sv)



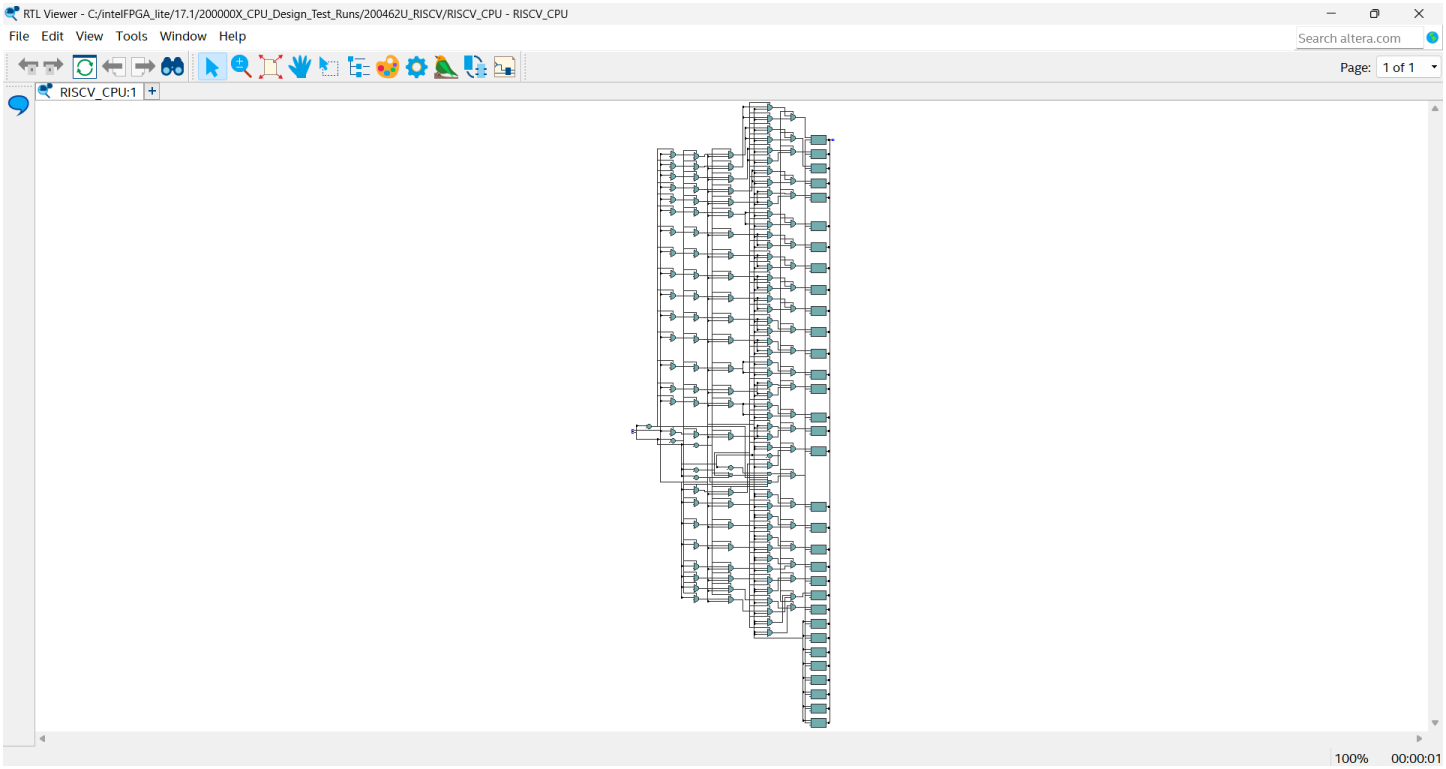
ALU Controller (ALUController.sv)



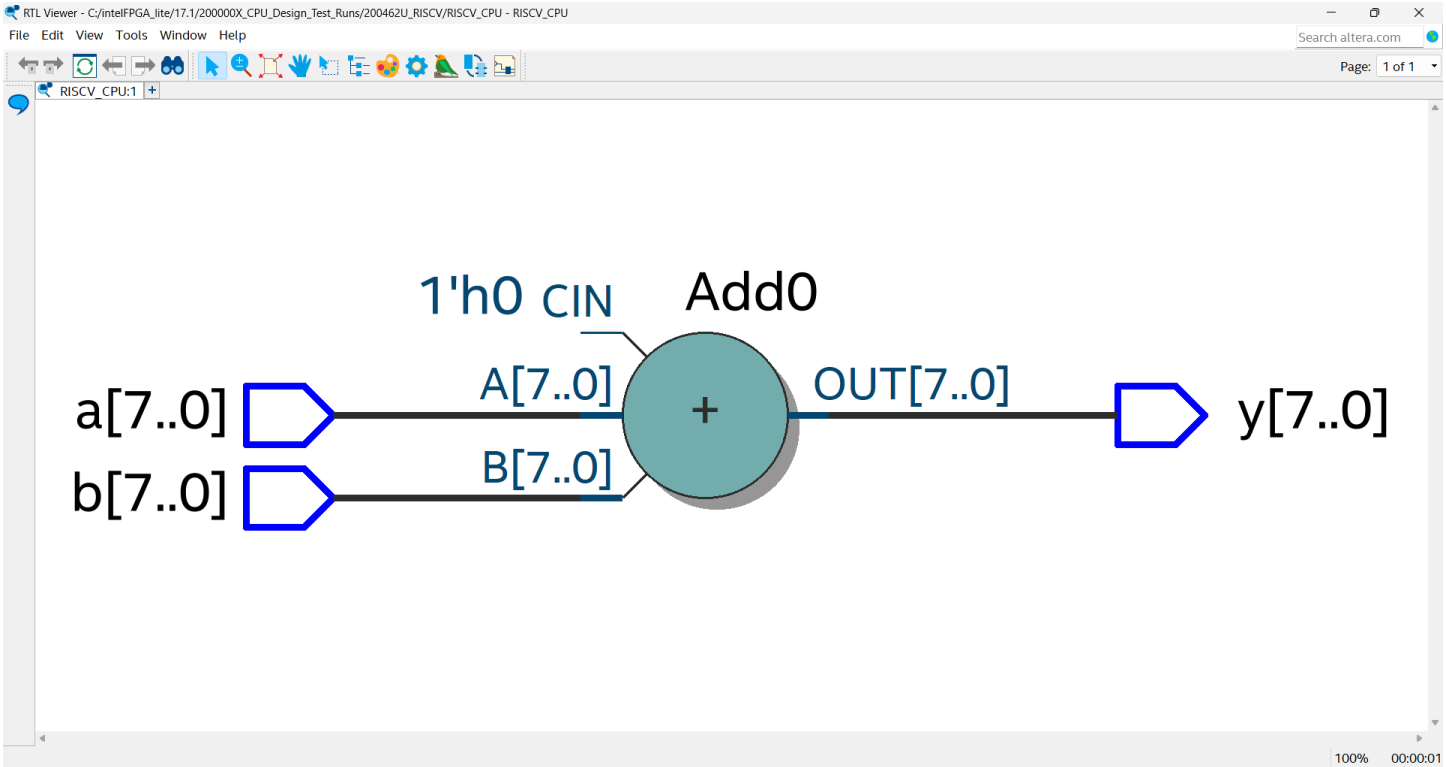
RISCV Data Path (Datapath.sv)



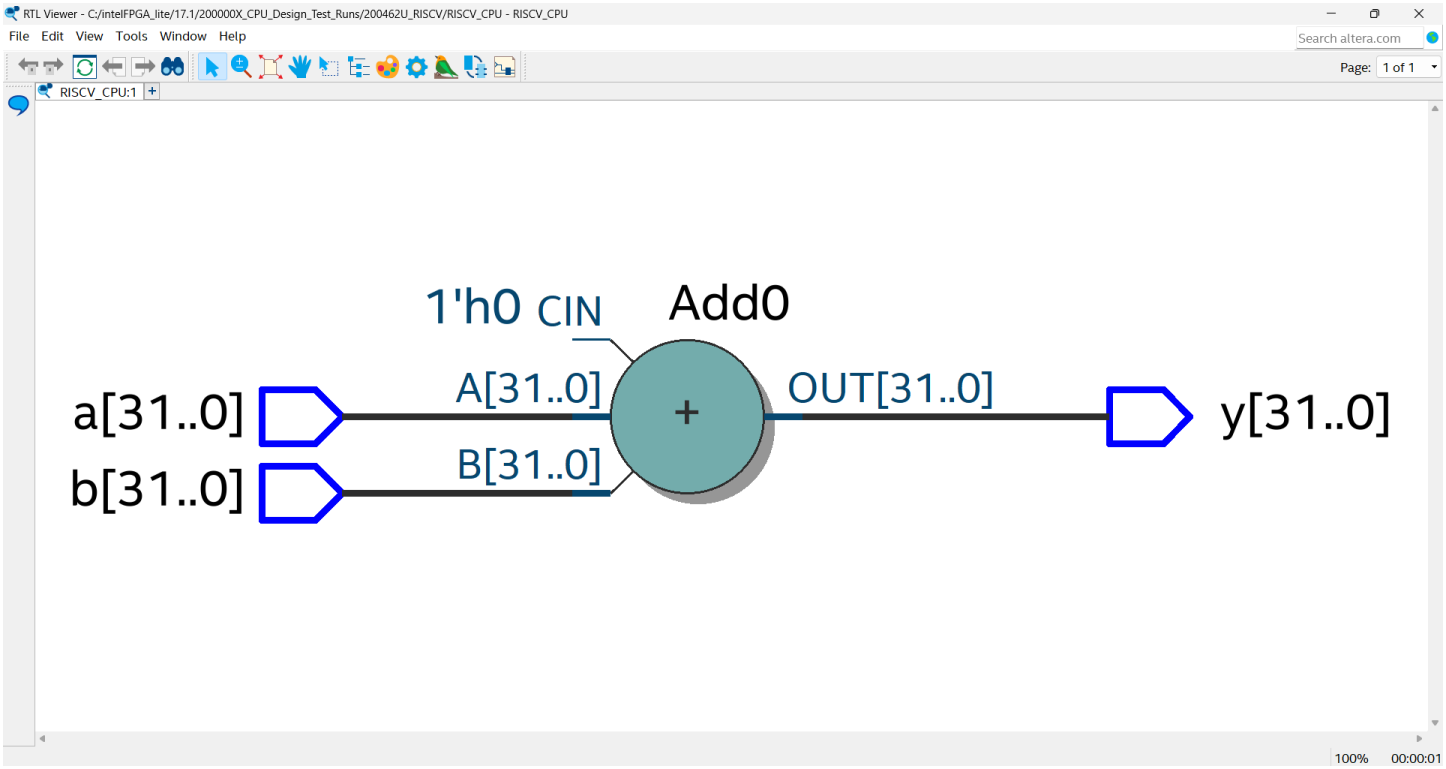
Data Extractor (data_extract.sv)



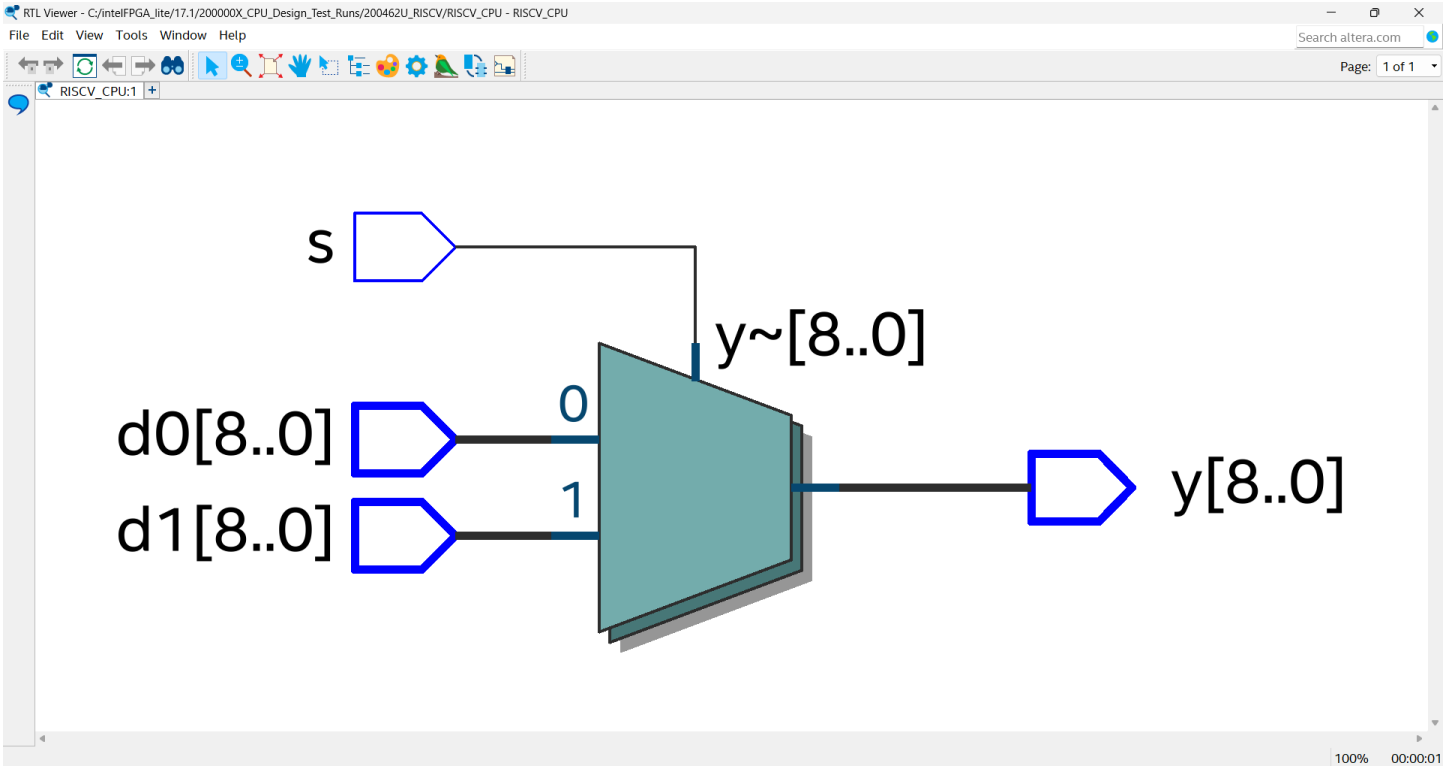
Adder (adder.sv)



Adder 32bit (adder_32.sv)



2 input Multiplexer (mux2.sv)



D flip-flop (flopr.sv)