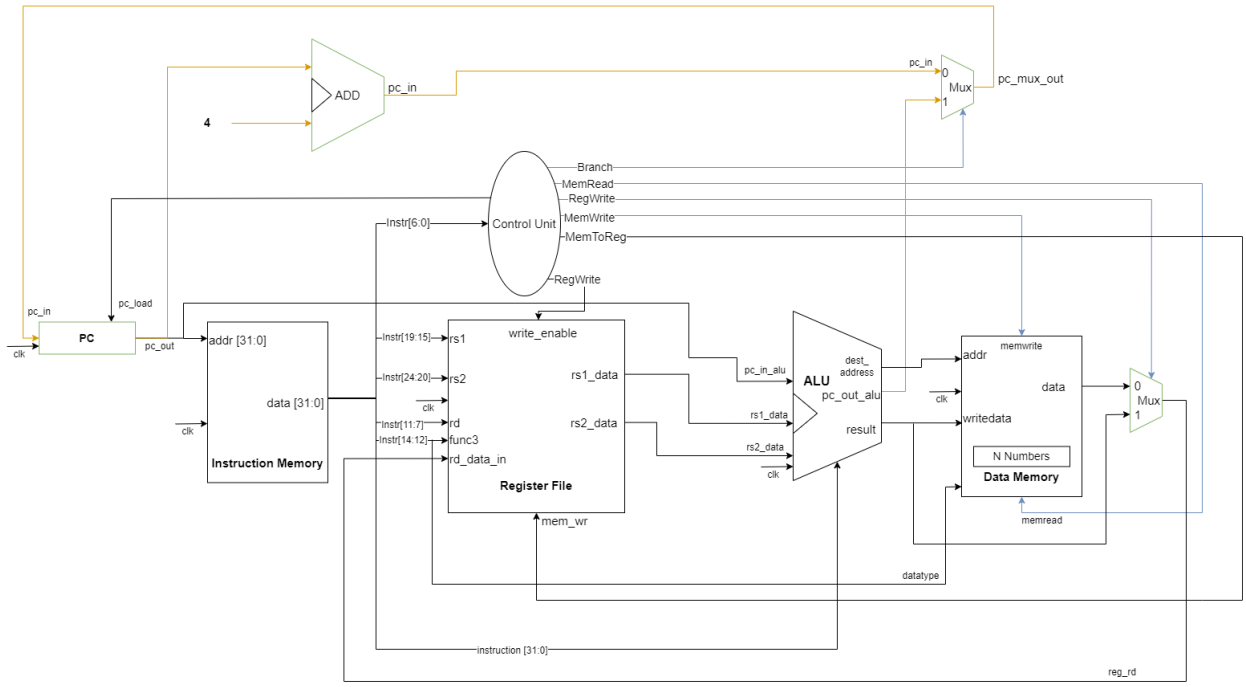
NYU-6463-RV32I processor : Milestone 3

Team: Parishi Naik (pmn7106), Varadraj Sinai Kakodkar (vns2008), Kartik Saxena(ks6028)

The NYU-6463-RV32I processor is a 32-bit architecture that executes a subset of the open-source RISC-V RV32I instruction set. The RISC-V processor is made of 5 main modules: Program Counter, Instruction Memory, Register File, Arithmetic Logic Unit, and Data Memory. There are other supplementary modules used such as adders, 2x1 multiplexers, and a Control Unit that controls the state of the processor at any given time.

Datapath



The NYU-6463-RV32I processor has 5 stage multicycle operation. It goes through 5 stages of Instruction Fetch, Instruction Decode, Execution, Memory and Write Back.

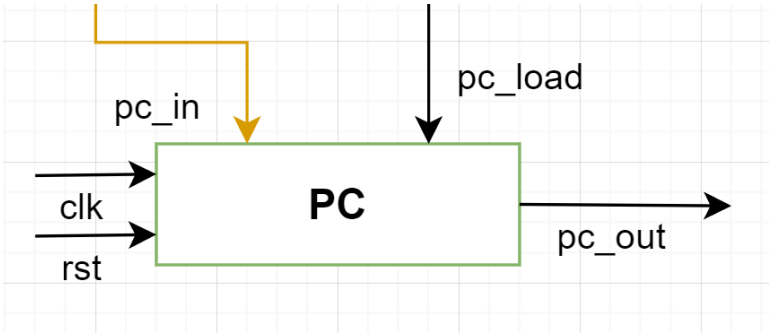
The Control Unit sends the PC_load signal to the Program Counter to start the processor in the IF stage which is when it loads the first PC value of 0x10000000. For the ID stage, the Instruction Memory gets the instruction from the PC value and sends it to the Register File and ALU. The Register File reads the source register addresses from the instruction and sends the data at the source registers to the ALU. In the Execute stage, the ALU performs the operation

specified by the instruction and sends the destination address for the load/store instructions to the Data Memory, the PC value for jump instructions to the PC_mux and output results to the Data Memory. The result from the ALU is stored during the Memory stage. Finally, the Data Memory sends either the result to the Register File or writes some data onto the Register File, depending on the type of instruction during the Write Back stage. For the next cycles, the PC_adder increments the PC value by 4 and sends it to PC_mux. The PC_mux decides which value of PC is to be loaded depending on the type of instruction.

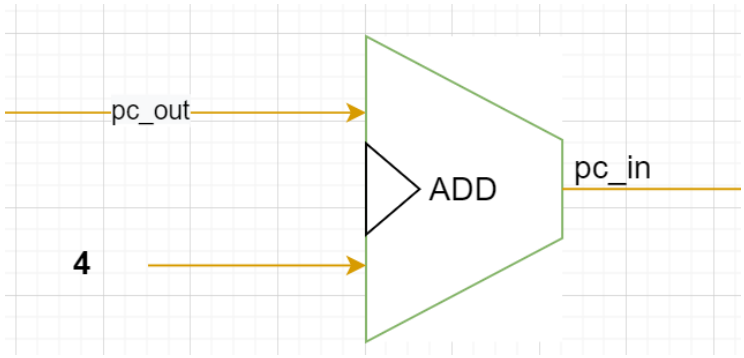
The Control Unit sends various memory read and write signals based on the load or store instruction as well as the stage of the FSM.

Module Overview

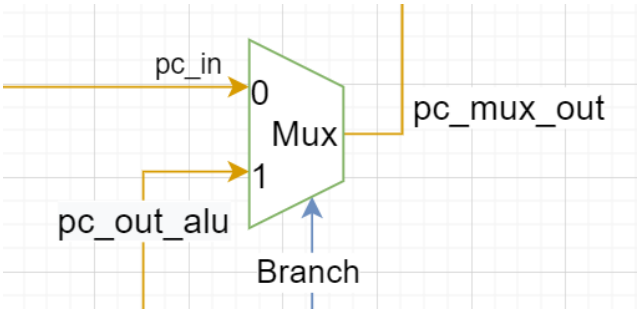
Program Counter: This module is a 32-bit register that holds the current value of the program counter that is given to the instruction memory to fetch the instructions. The Control Unit sends it a PC_Load signal to load the value of the PC. This is only before the Instruction fetch cycle and outputs the value in the next cycle. The program count is either incremented by 4 or jumps to the PC value given by the ALU depending on the type of instruction (branch/not branch). Upon receiving the rst signal, the PC is equal to the start address of the instruction memory (0x10000000).



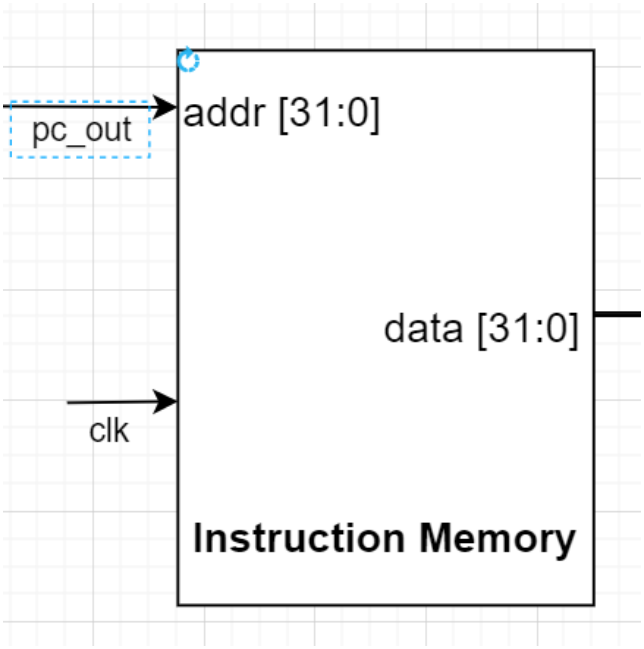
PC_Adder: This is a 32-bit adder that increments the value of PC by 4.



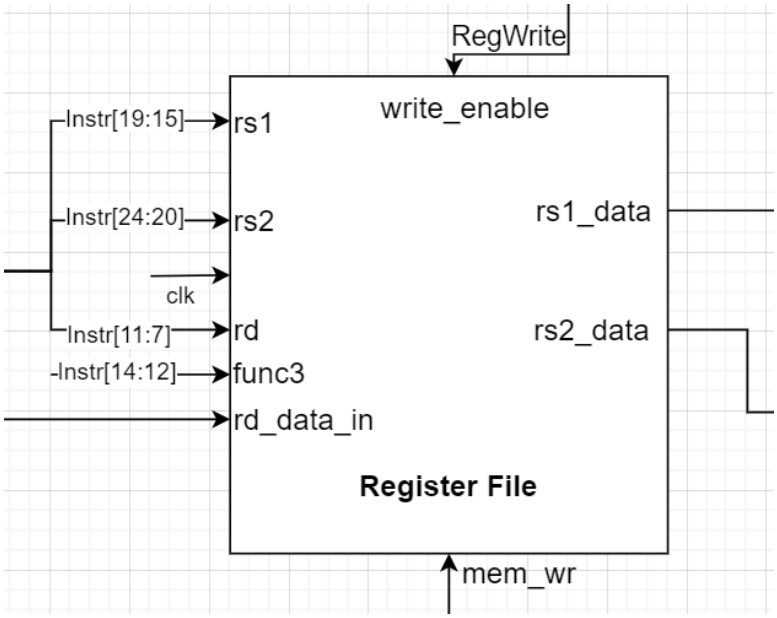
PC_Mux_2x1: This 2x1 multiplexer is used during the branch instructions. It takes the Branch signal from the Control Unit as a select signal and decides the PC value between the PC value from the ALU and the PC value from the adder.



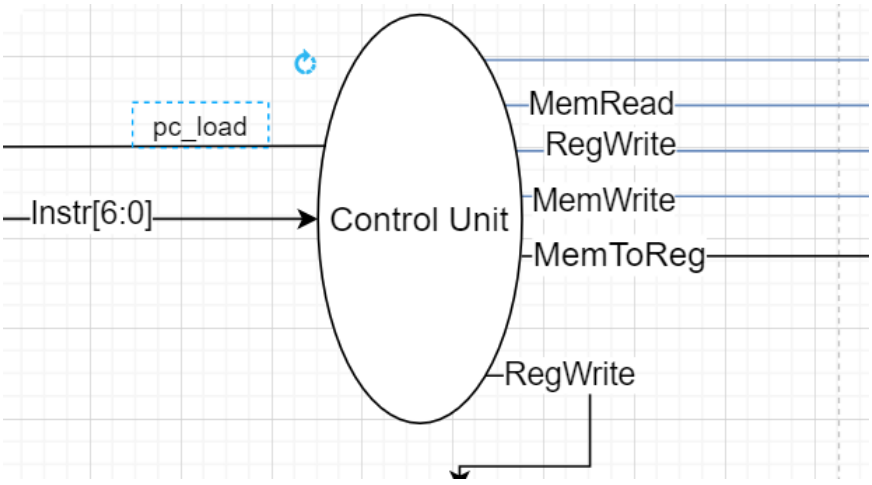
Instruction Memory: The instruction memory contains the instruction to be executed. Its width is 4 bytes (32-bits), although it is byte-addressed. Its output data is the 32-bit instruction which is sent to the other modules. Its address starts at 0x10000000.



Register File: This module has 32 32-bit registers with R0 being a read-only register having its value hardcoded to 0. The write_enable signal coming from the Control Unit controls the read and write functionality of the module. It receives the 32-bit instruction from the IM and extracts the rs1, rs2, rd, func3 values from it. The func3 value is used for load instructions where it can decide whether to load a byte, half-word, or word based on the instruction. This module also gets a mem_wr signal from the Control Unit to write values during the Write Back stage.

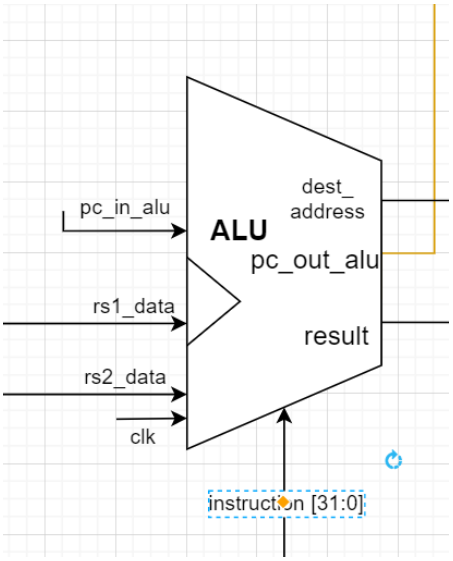


Control Unit: The Control Unit ensures coordination between all the other modules of the processor. This component contains the FSM that runs the 5 stage RiscV pipeline or multi cycle implementation of the processor. The control signals going to various stages decide whether the stage is active at any particular time or not. The Branch signal goes to the PC_mux when there is a branch instruction being executed. The MemRead signal is active when data is being from the data memory. The RegWrite goes to the data_mux to decide whether the result from the ALU or the data from the data memory is being written to the Register File. It is also sent to the Register File as write_enable signal to specify whether data is being read from or written to the Register File. The MemWrite signal is active when data is being stored on the data memory.

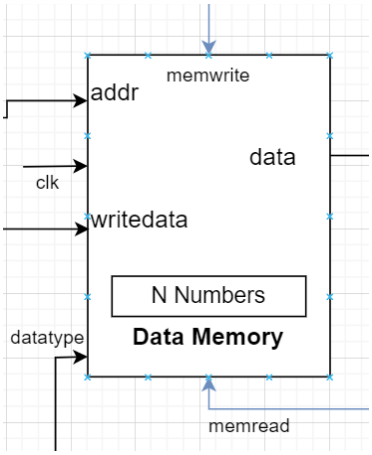


Arithmetic Logic Unit: The ALU performs arithmetic, load/store and various other functions. It implements 40 instructions that are I-type, B-type, S-type, U-type and J-type. This module gets inputs in the form of the 32 bit instruction, data from the 2 source registers and the PC value. Instead of placing a multiplexer at the first source of the ALU, a separate pc_in_alu value is

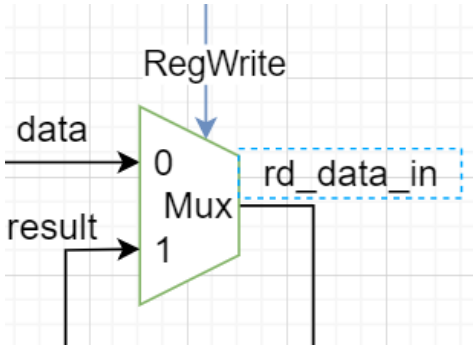
being taken as input for the jump instructions. Also, immediate values are being extracted from the instruction within the ALU itself instead of there being a separate immediate generation unit. Since immediate is being extracted within the ALU, no multiplexer is placed at the second source of the ALU. The ALU generates the result value at the end of the instruction or gives a new PC value for jump instructions. It also computes and gives the destination address for the load/store instructions.



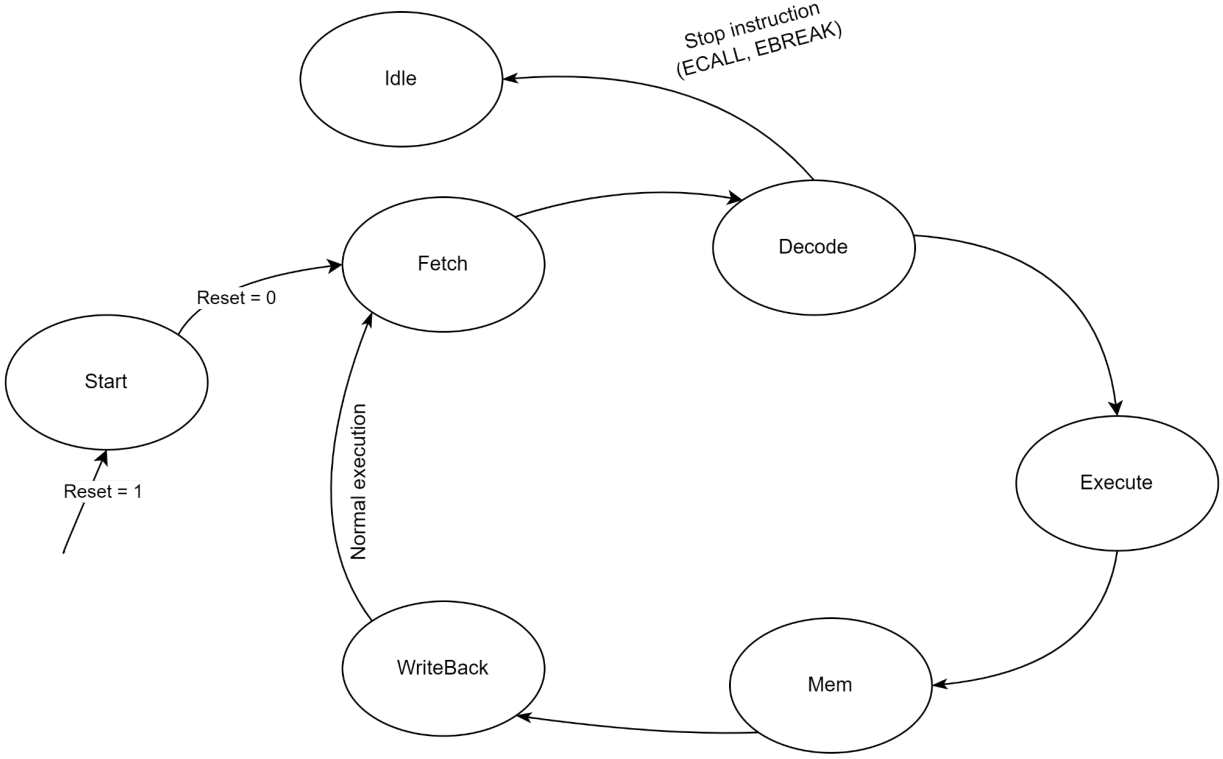
Data Memory: The data memory stores the data and is accessed using load and store instructions. It is 4-bytes (32-bits) wide but it is byte addressable and 4KBytes in size. Its address starts at 0x80000000. The data memory has been implemented using 4 interleaved sets of 8-bit (one-byte) wide memories. Further, as per the project specifications, a read-only memory has been made with the N numbers of the members of the group mapped at address 0x00100000, 0x00100004, 0x00100008. The read-only switches start at address 0x00100010 and read/write LEDs at address 0x00100014. The data memory takes the result from the ALU as well as memwrite and memread signals from the Control Unit to decide whether load or store is being executed.



Data_mux_2x1: This multiplexer takes the RegWrite signal from the Control Unit and sends either the ALU result or some value from the Data Memory depending on whether it is a load instruction or not.



FSM for Multi-Cycle Implementation



The FSM has been implemented in 7 stages: Fetch, Decode, Execute, Mem, WriteBack, Idle, Start.

The FSM starts at the Start state then enters Fetch state. From the Fetch state, the FSM moves to Decode, Execute, Mem and WriteBack and back to Fetch. In the event a stop instruction (ECALL, EBREAK) is to be executed, the FSM enters the Idle state and remains there.

Depending on the OPCODE defined in the Instr[6:0] various control signals are asserted in the Fetch, Decode, Execute, Mem, WriteBack stages. There are 9 different types of instructions namely Rtype,Itype,Stype,Btype,Utype,UJtype,Ltype,Stop,NOP.

The processor would go through each of the 5 stages of execution regardless of the instruction type, with each stage being one clock cycle long. Even if an instruction could be finished within lesser time, it would take 5 cycles to be executed on this processor.

Testing

Execution of each of the instruction takes 5 cycles and any test case would require 5*n where n is the number of instructions in the test case.

The design was tested for correctness by testing individual modules during milestone 1 using testbenches for individual modules.

The integrated design was tested for each of the 40 instructions using low level test cases eg. 8,16,32 bit load/store, both positive and negative immediate value loading for I type of instructions, both forward and backward jump for Jtype and Btype instructions etc.

The low level test cases for individual instructions are described in the following sections

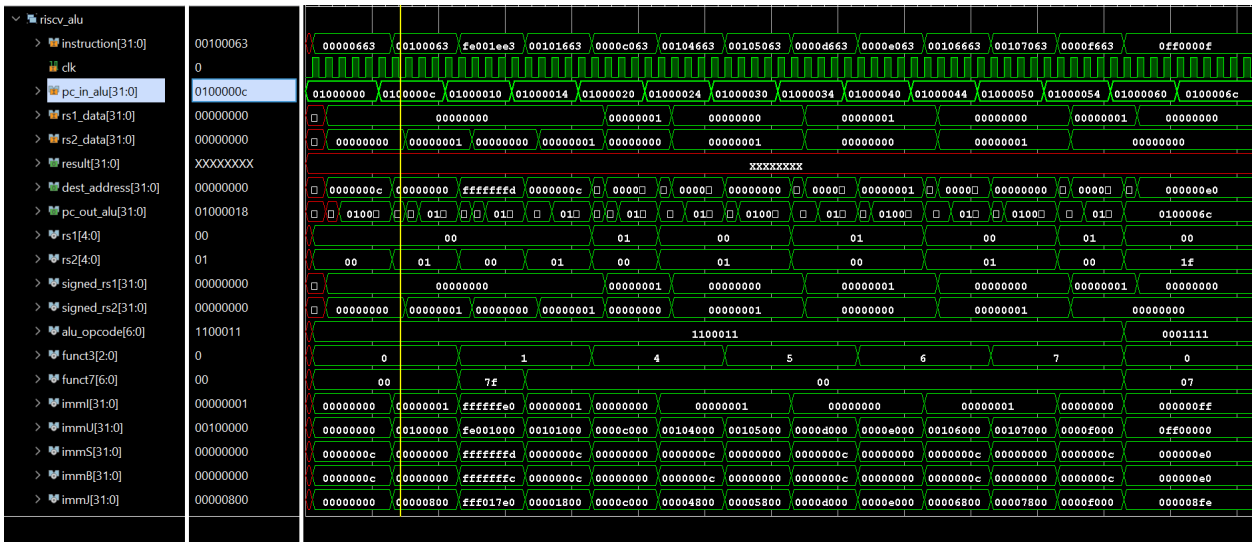
The design was also verified by using 2 high level test programs that included finding the factorial of a number and finding the sum of cubes of odd numbers between +M and -N.

The riscV assembly codes were written and assembled into hex encoding generated by the gcc toolchain as well as the Ripes simulator. These instructions were loaded into the memory.

B-type: Each of the branch instructions was tested by making various labels in between the assembly code. Each instruction was tried for both true and false cases (e.g. branch equal and branch not equal for BEQ), as well as forward and backward jumps were tested for each instruction.

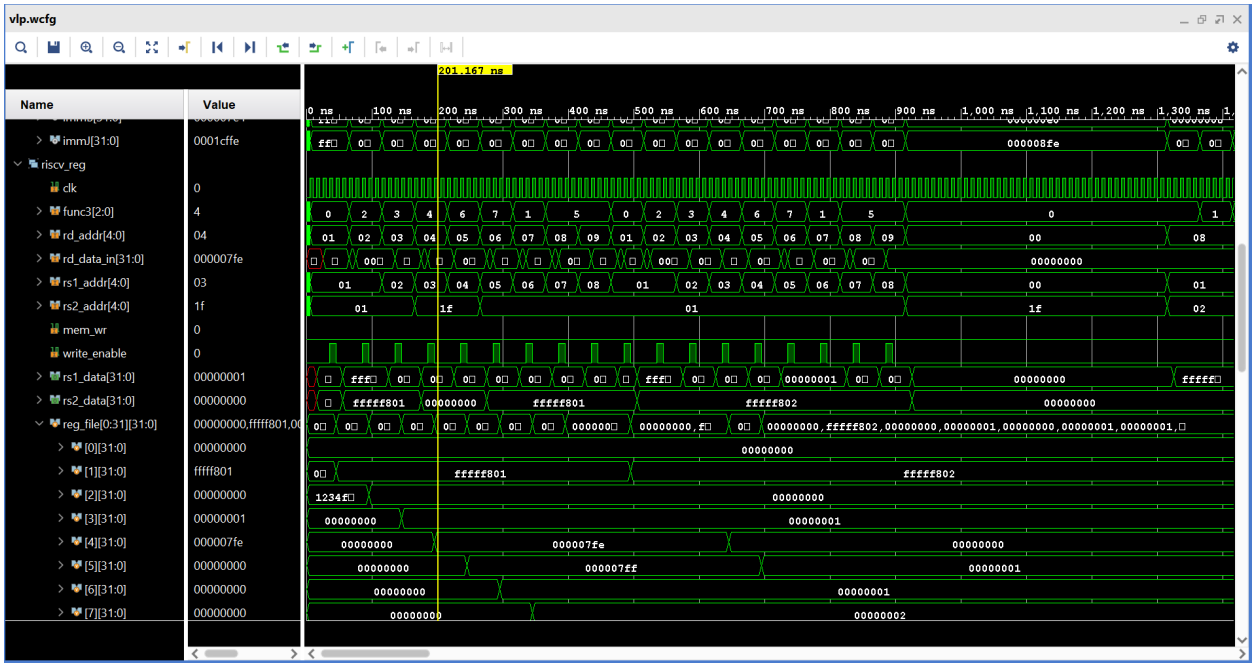
```
   0:       00000663        beq x0 x0 12 <next1>
   4:       01450533        add x10 x10 x20
   8:       41450533        sub x10 x10 x20

0000000c <next1>:
   c:       00100063        beq x0 x1 0 <next1>
  10:       fe001ee3        bne x0 x0 -4 <next1>
  14:       fe101ce3        bne x0 x1 -8 <next1>
  18:       01450533        add x10 x10 x20
  1c:       41450533        sub x10 x10 x20

00000020 <next2>:
  20:       0000c063        blt x1 x0 0 <next2>
  24:       00104663        blt x0 x1 12 <next3>
  28:       01450533        add x10 x10 x20
  2c:       41450533        sub x10 x10 x20

00000030 <next3>:
  30:       00105063        bge x0 x1 0 <next3>
  34:       0000d663        bge x1 x0 12 <next4>
  38:       01450533        add x10 x10 x20
  3c:       41450533        sub x10 x10 x20

00000040 <next4>:
  40:       0000e063        bltu x1 x0 0 <next4>
  44:       00106663        bltu x0 x1 12 <next5>
  48:       01450533        add x10 x10 x20
```

```
00000030 <next3>:
  30:       00105063        bge x0 x1 0 <next3>
  34:       0000d663        bge x1 x0 12 <next4>
  38:       01450533        add x10 x10 x20
  3c:       41450533        sub x10 x10 x20

00000040 <next4>:
  40:       0000e063        bltu x1 x0 0 <next4>
  44:       00106663        bltu x0 x1 12 <next5>
  48:       01450533        add x10 x10 x20
  4c:       41450533        sub x10 x10 x20

00000050 <next5>:
  50:       00107063        bgeu x0 x1 0 <next5>
  54:       0000f663        bgeu x1 x0 12 <next6>
  58:       01450533        add x10 x10 x20
  5c:       41450533        sub x10 x10 x20

00000060 <next6>:
  60:       00000073        ecall
```

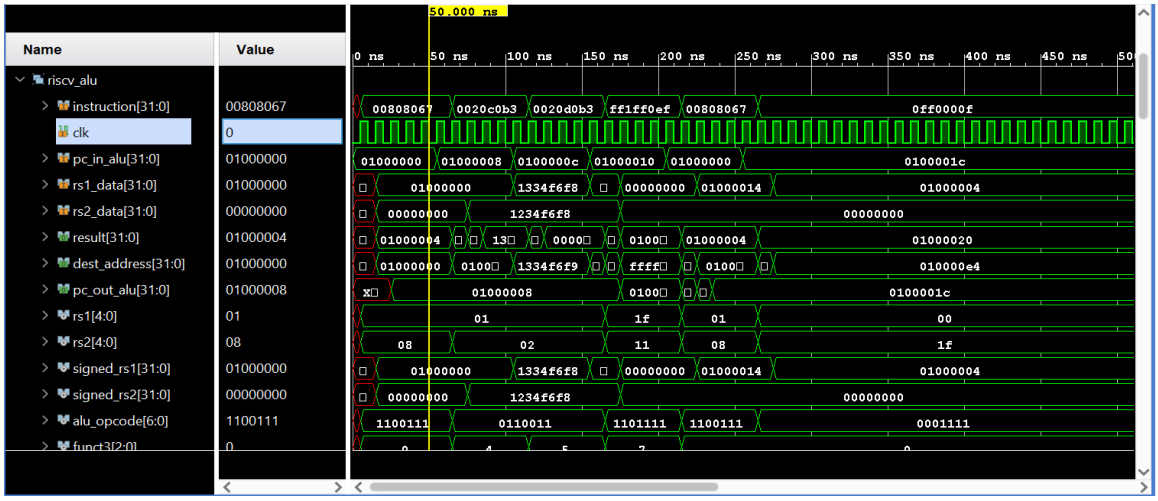


I-type: For the I-type instructions, one instant of each instruction was tested. Moreover, for the JALR instruction, both forward and backward jumps were tested.

```
80108093    addi x1,x1,-0x7ff
0010a113    slti x2,x1,1
00113193    sltiu x3,x2,1
7ff1c213    xori x4,x3,0x7ff
7ff26293    ori x5,x4,0x7ff
0012f313    andi x6,x5,1
00131393    slli x7,x6,1
0013d413    srli x8,x7,1
40145493    srai x9,x8,1
00108093    addi x1,x1,1
0010a113    slti x2,x1,1
00113193    sltiu x3,x2,1
0011c213    xori x4,x3,1
00126293    ori x5,x4,1
0012f313    andi x6,x5,1
00131393    slli x7,x6,1
0013d413    srli x8,x7,1
40145493    srai x9,x8,1
```

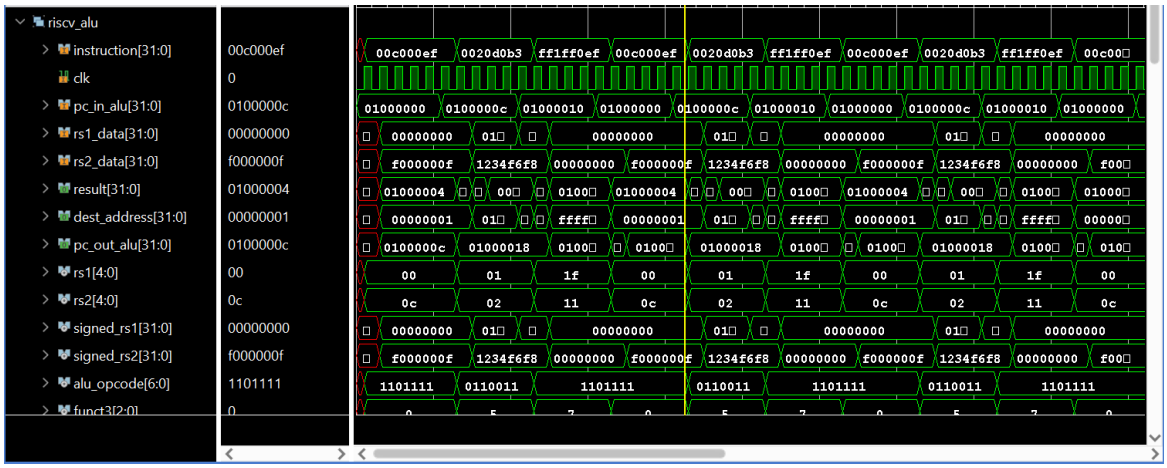



```
00000000 <start>:
    0:      008080e7        jalr x1 x1 8          IF
    4:      002080b3        add x1 x1 x2
    8:      0020c0b3        xor x1 x1 x2

0000000c <end>:
    c:      0020d0b3        srl x1 x1 x2
    10:     ff1ff0ef        jal x1 0x0 <start>
```


50.000 ns
Name
Value
riscv_alu
instruction[31:0]
00808067
clk
0
pc_in_alu[31:0]
01000000
rs1_data[31:0]
01000000
rs2_data[31:0]
00000000
result[31:0]
01000004
dest_address[31:0]
01000000
pc_out_alu[31:0]
01000008
rs1[4:0]
01
rs2[4:0]
08
signed_rs1[31:0]
01000000
signed_rs2[31:0]
00000000
alu_opcode[6:0]
1100111
funct3[2:0]
0
0 ns
50 ns
100 ns
150 ns
200 ns
250 ns
300 ns
350 ns
400 ns
450 ns
00808067
0020c0b3
0020d0b3
ff1ff0ef
00808067
0ff0000f
01000000
01000008
0100000c
01000010
01000000
0100001c
01000000
1334f6f8
00000000
01000014
01000004
00000000
1234f6f8
00000000
01000004
13
0000
0100
01000004
01000020
01000000
0100
1334f6f9
ffff
0100
010000e4
x
01000008
0100
0100001c
01
1f
01
00
08
02
11
08
1f
01000000
1334f6f8
00000000
01000014
01000004
00000000
1234f6f8
00000000
1100111
0110011
1101111
1100111
0001111

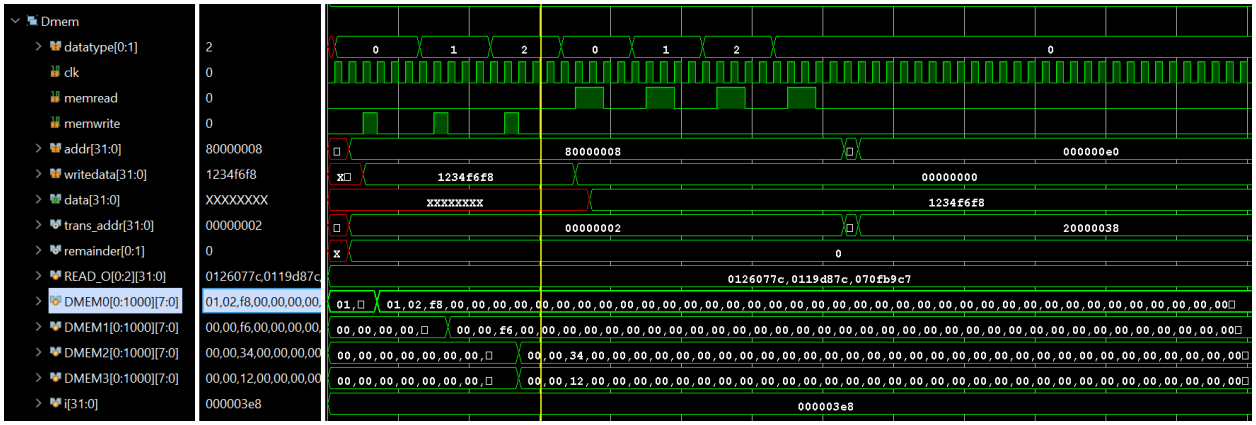
J-type: For the JAL instruction, instances were made in which both forward and backward jumps were tested.

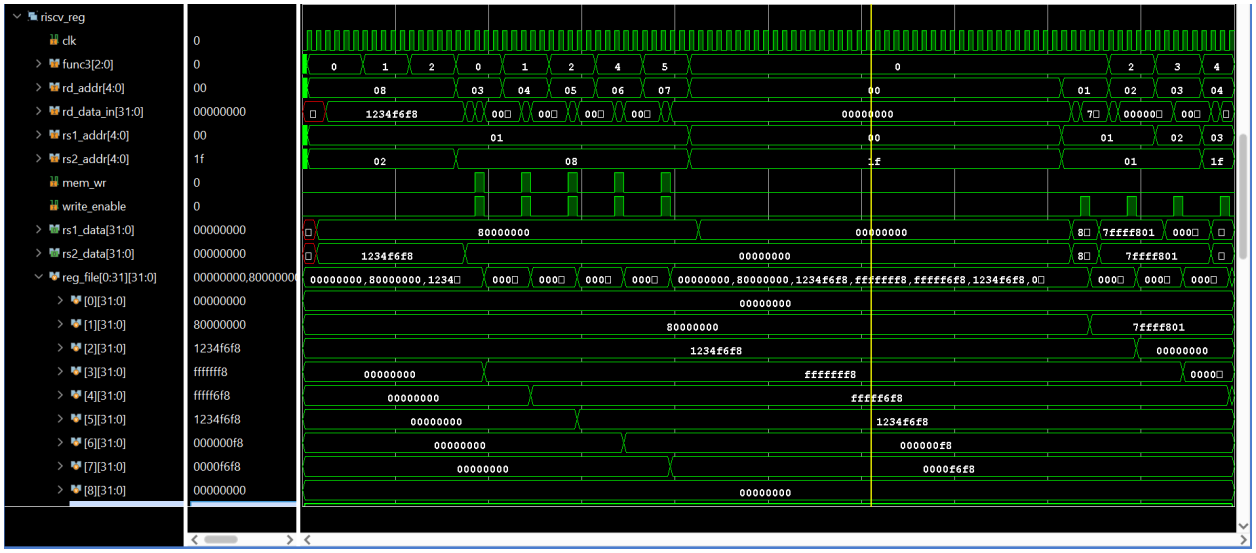
```
00000000 <start>:
    0:      00c000ef        jal x1 0xc <end>    IF
    4:      002080b3        add x1 x1 x2
    8:      0020c0b3        xor x1 x1 x2

0000000c <end>:
    c:      0020d0b3        srl x1 x1 x2
    10:     ff1ff0ef        jal x1 0x0 <start>
```



Load/Store Instructions: For the load and store instructions, instances of all types of load and store were tested (byte, halfword, word). The results were checked in both the Data Memory and Register File modules.

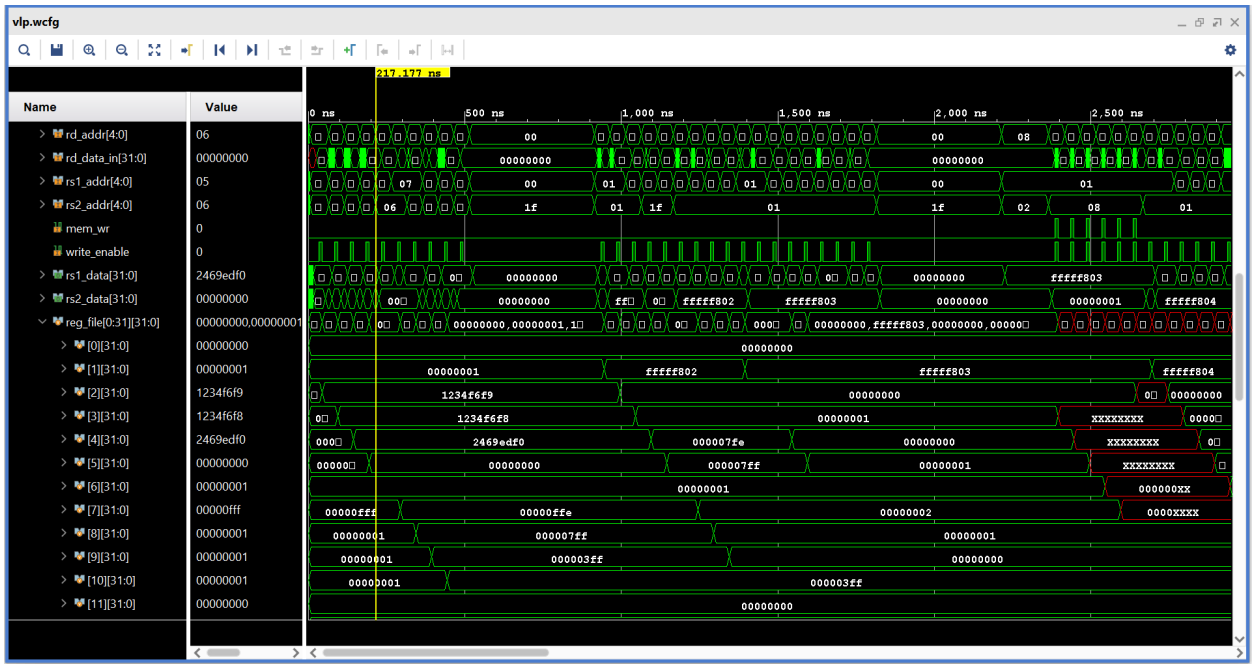
sb x2, 8(x1)	00208423
sh x2, 8(x1)	00209423
sw x2, 8(x1)	0020a423
lb x3, 8(x1)	00808183
lh x4, 8(x1)	00809203
lw x5, 8(x1)	0080a283
lbu x6, 8(x1)	0080c303
lhu x7, 8(x1)	0080d383





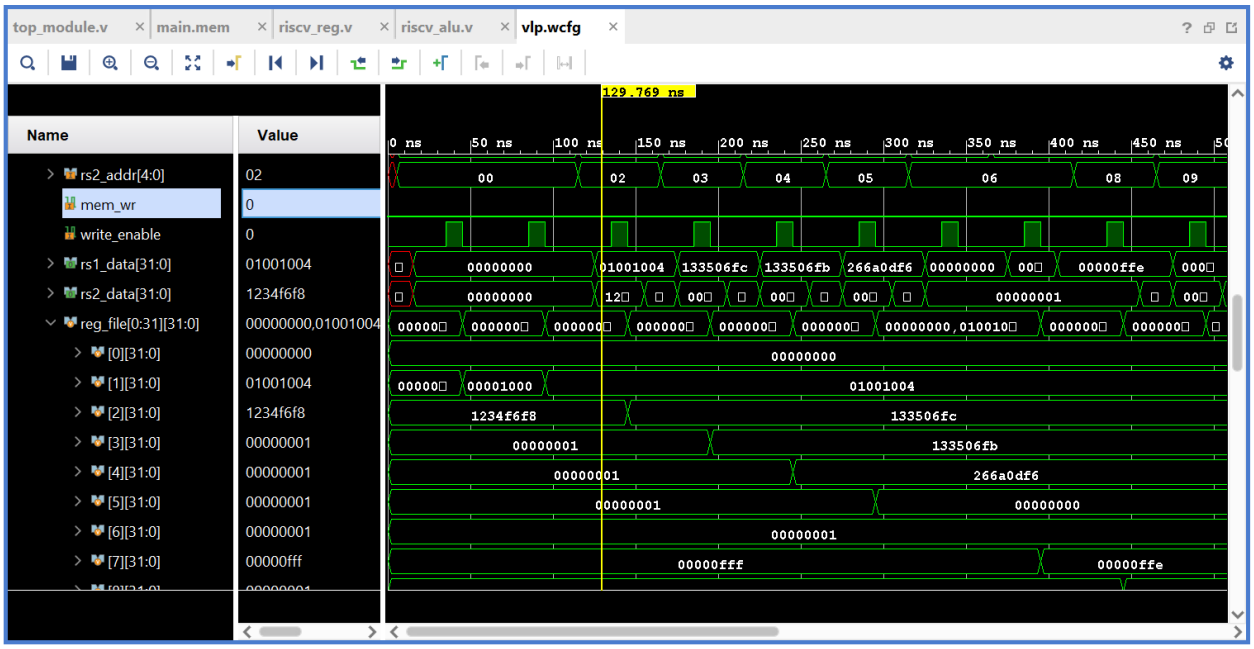
R-type: Various instances for each of the R type instructions were used for testing.

add x2, x1, x2	00208133
sub x3, x2, x3	403101b3
sll x4, x3, x4	00419233
slt x5, x4, x5	005222b3
sltu x6, x5, x6	0062b333
xor x7, x7, x6	0063c3b3
srl x8, x7, x8	0083d433
sra x9, x8, x9	409454b3
or x10, x9, x10	00a4e533
and x11, x10, x11	00b575b3



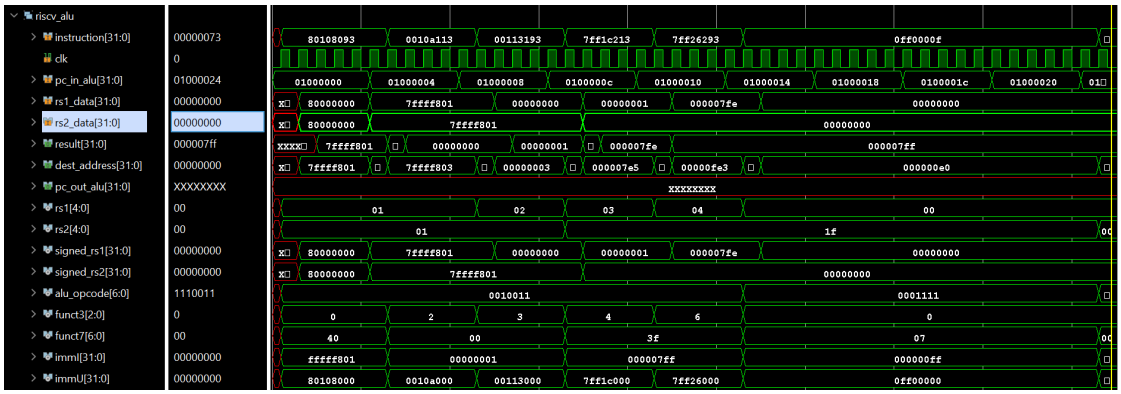
U-type: The two U-type instructions (LUI and AUIPC) were tested using different immediate values.

lui x1, 0x00001	000010b7
auipc x1,0x00001	00001097

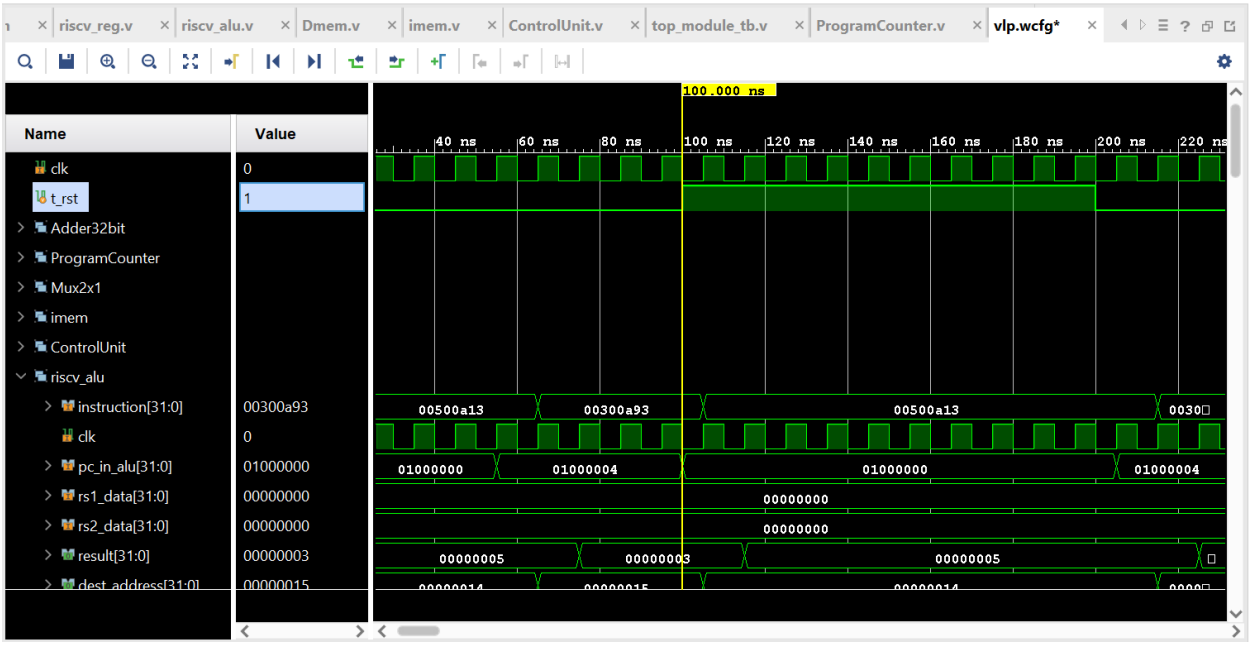


Other Instructions: A few instances for each of the rest of the instructions (FENCE, ECALL, EBREAK) were tested.

fence	0FF0000F
fence	0FF0000F
fence	0FF0000F
fence	0FF0000F
ecall	00000073
ebreak	00100073



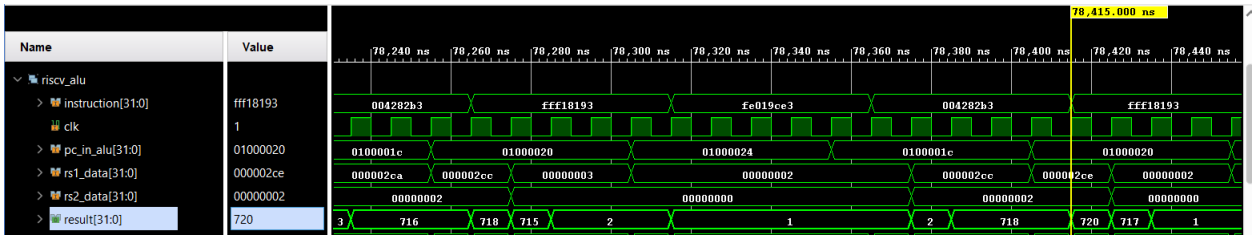
Reset Signal Test: A few test instances had been run in order to check the correctness of the reset signal and loading of PC default value and the stalling of FSM.



Higher Level Testing

For further testing, two assembly codes were used to check the functionality of the processor. One code is to find the factorial of an integer and the other is to calculate the sum of cubes of odd numbers between two integers.

- Factorial: The factorial was tested for numbers from 1 to 12, after which the value of factorial goes beyond 4294967296 = 2^{32} which is the maximum value a register can hold.

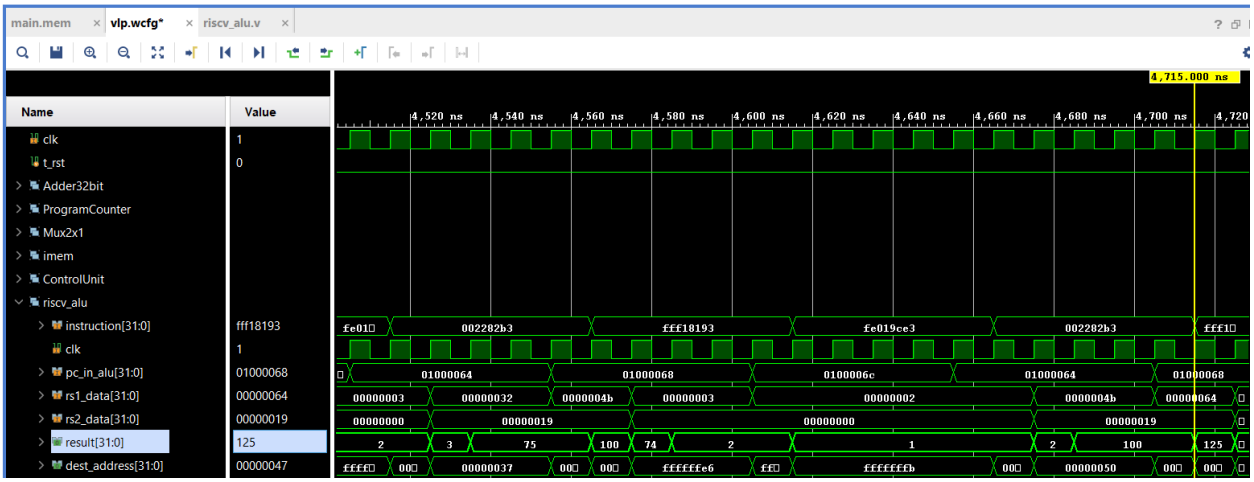


This is the simulation output for n = 6.

N	Result(X3)	Clock cycles
0	0	20
1	1	40
2	2	40
3	6	100
4	24	175
5	120	265
6	720	370
7	5040	490
8	40320	625
9	362880	775
10	3620800	940

- Sum of Cubes of Odd Numbers Between Two Integers +M and -N: This code was tasted for about 25 integer pairs.

  Shown below is the simulation output for inputs 5 and -3.

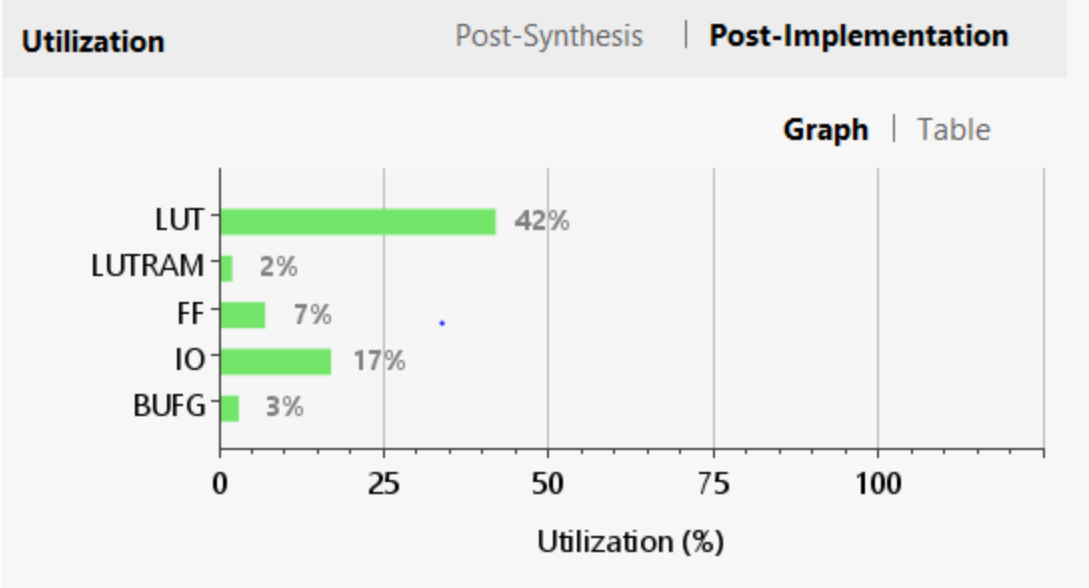


Describe high-level test cases in assembly (and how many clock cycles each test case takes and why) and a demonstration of their output (this may be a simulation

Sr No	+M	-N	Result(X12)	Clock cycles
1	0	0	0	105
2	1	0	1	135
3	0	1	-1	135
4	1	2	0	165
5	2	3	-27	365
6	3	4	0	565
7	5	3	125	825

Resource Utilization

The synthesis and implementation was run for FPGA part **xc7a100tcsg324-1** or Nexys4DDR board. The resource utilization reported as follows.

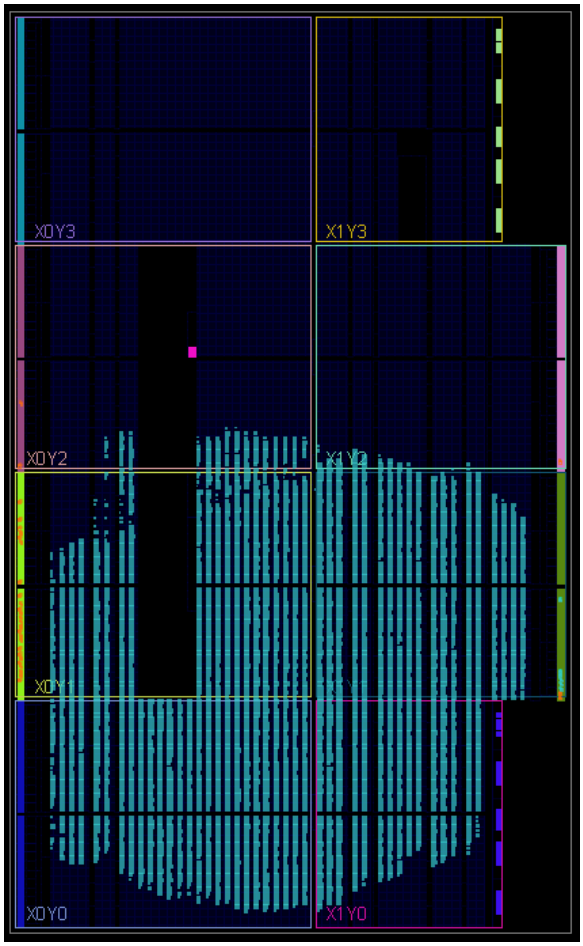


The utilization after synthesis

Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Bonded IOB (210)	BUFGCTRL (32)
top_module	26716	8509	1235	600	36	1
ALU (riscv_alu)	696	296	0	0	0	0
CU (ControlUnit)	11	8	0	0	0	0
DMEM (Dmem)	2674	8056	1232	600	0	0
IMEM (imem)	22479	53	0	0	0	0
PC (ProgramCounter)	216	32	3	0	0	0
REG (riscv_reg)	650	64	0	0	0	0

The utilization after implementation

Name	Slice LUTs (63400)	Bonded IOB (210)	BUFGCTRL (32)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Slice (15850)	LUT as Logic (63400)	LUT as Memory (19000)
top_module	26751	36	1	8525	1235	600	7161	26295	456
ALU (riscv_alu)	721	0	0		0	0	807	721	0
CU (ControlUnit)	11	0	0		0	0	7	11	0
DMEM (Dmem)	2674	0	0		1232	600	6440	2266	408
IMEM (imem)	22489	0	0		0	0	6358	22489	0
PC (ProgramCounter)	216	0	0		3	0	95	216	0
REG (riscv_reg)	649	0	0		0	0	255	601	48



Timing Simulation

The timing constraints were specified for the clock and input/output mapped to the read only switches and LEDs in the constraint file.

create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clk}];

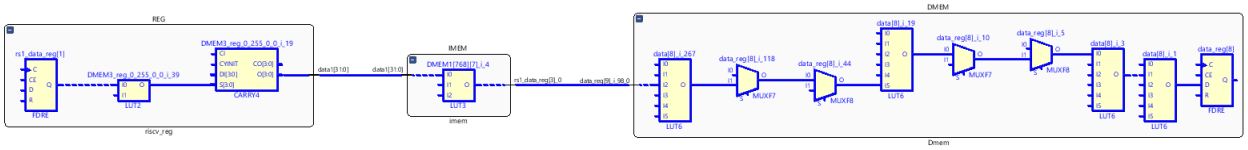
set_max_delay -datapath_only -from [all_inputs] -to [all_outputs] 10.000

The design passes implementation and all the user specified design constraints were met. The timing simulation reports are attached along with the project. Following is the timing summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.150 ns	Worst Hold Slack (WHS):	0.097 ns	Worst Pulse Width Slack (WPWS):	3.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	21757	Total Number of Endpoints:	21757	Total Number of Endpoints:	9025

All user specified timing constraints are met.

Eg. worst path for worst negative setup slack is from register file to data memory used in case of the store instruction i.e. REG/rs1_data_reg[1]/C to DMEM/data_reg[8]/D



Schematic of worst negative setup slack

Tcl Console | Messages | Log | Reports | Design Runs | Power | Methodology | DRC | Timing | Utilization

Intra-Clock Paths - sys_clk_pin - Setup

- General Information
- Timer Settings
- Design Timing Summary
- Clock Summary (1)
- Check Timing (60)
- Intra-Clock Paths
  - sys_clk_pin
    - Setup 0.150 ns (10)
    - Hold 0.097 ns (10)
    - Pulse Width 3.750 ns (30)
- Inter-Clock Paths
- Other Path Groups
- User Ignored Paths
- Unconstrained Paths

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock
Path 1	0.150	11	6	123	REG/rs1_data_reg[1]/C	DMEM/data_reg[8]/D	9.867	3.123	6.744	10.0	sys_clk_pin
Path 2	0.266	12	7	123	REG/rs1_data_reg[1]/C	DMEM/data_reg[10]/D	9.688	3.251	6.437	10.0	sys_clk_pin
Path 3	0.267	9	8	118	REG/rs1_data_reg[1]/C	DMEM/DMEM...980][0]/D	9.772	2.909	6.863	10.0	sys_clk_pin
Path 4	0.287	9	8	118	REG/rs1_data_reg[0]/C	DMEM/DMEM...889][4]/D	9.891	2.792	7.099	10.0	sys_clk_pin
Path 5	0.288	7	6	83	REG/rs1_data_reg[1]/C	DMEM/DMEM...734][3]/D	9.718	2.413	7.305	10.0	sys_clk_pin
Path 6	0.346	11	8	118	REG/rs1_data_reg[0]/C	DMEM/data_reg[12]/D	9.652	3.437	6.215	10.0	sys_clk_pin
Path 7	0.357	11	7	120	REG/rs1_data_reg[0]/C	DMEM/data_reg[11]/D	9.611	3.287	6.324	10.0	sys_clk_pin
Path 8	0.361	8	7	118	REG/rs1_data_reg[0]/C	DMEM/DMEM...281][5]/D	9.667	2.674	6.993	10.0	sys_clk_pin
Path 9	0.363	9	8	118	REG/rs1_data_reg[1]/C	DMEM/DMEM...979][0]/D	9.739	2.679	7.060	10.0	sys_clk_pin
Path 10	0.367	8	7	118	REG/rs1_data_reg[0]/C	DMEM/DMEM...798][6]/D	9.724	2.668	7.056	10.0	sys_clk_pin

Future Optimizations

The design proposed here satisfies all the requirements as part of the specs for NYU-6463-RV32I processor. Certain optimizations could be made to the current design which could be useful to minimize the resource utilization and improve performance.

Some improvements for the current design are:

- A separate immediate generation unit rather than integrating that logic into ALU which would result in lesser LUT usage for ALU
- Additional control logic block which could decode the ALU instructions in order to make the design modular
- The current design does not infer any BRAM resources and implements the IMEM and DMEM using LUTs. The mapping of IMEM and DMEM onto the BRAM would result in minimizing the LUT usage and improve the timing performance which would make the design faster.
- Current design is just a 5 stage multi cycle design which is not very efficient from a practical point of view. Improvement could be made by implementing a pipelined processor.

Youtube Link: https://youtu.be/fOEYcsd_2b0