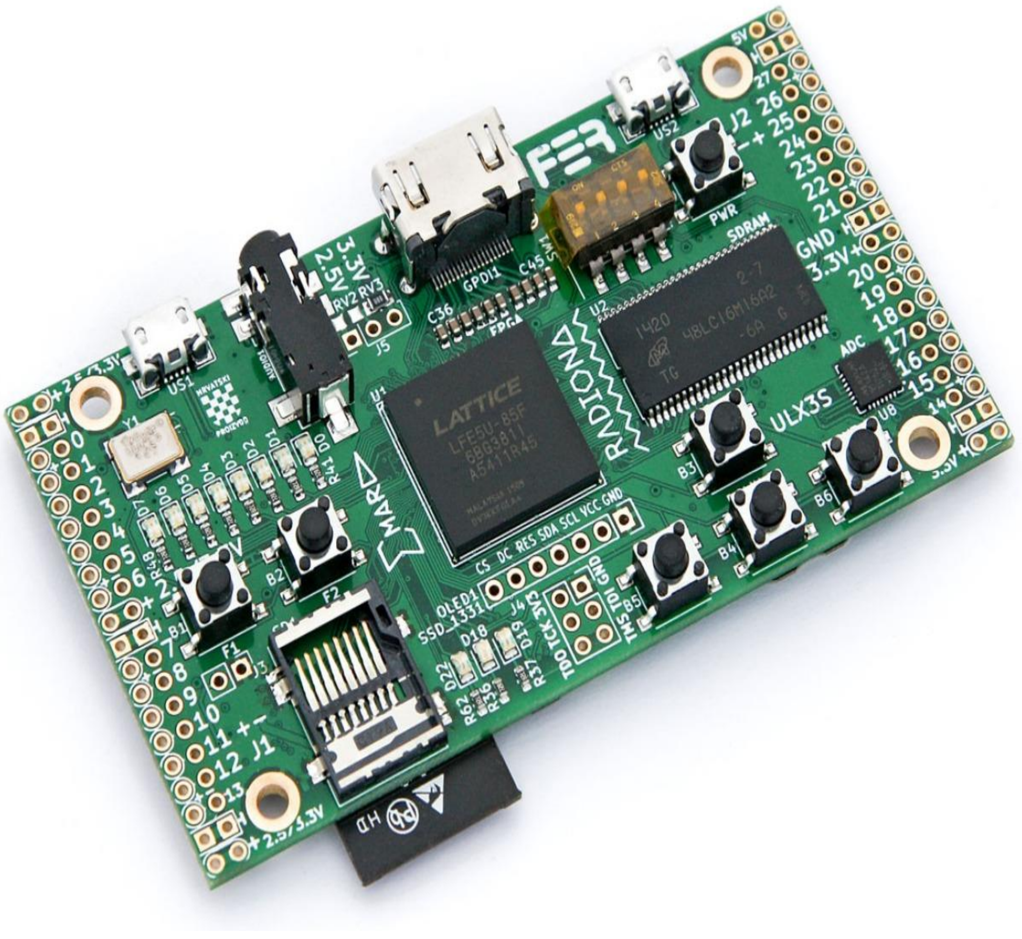
Advanced Hardware Design



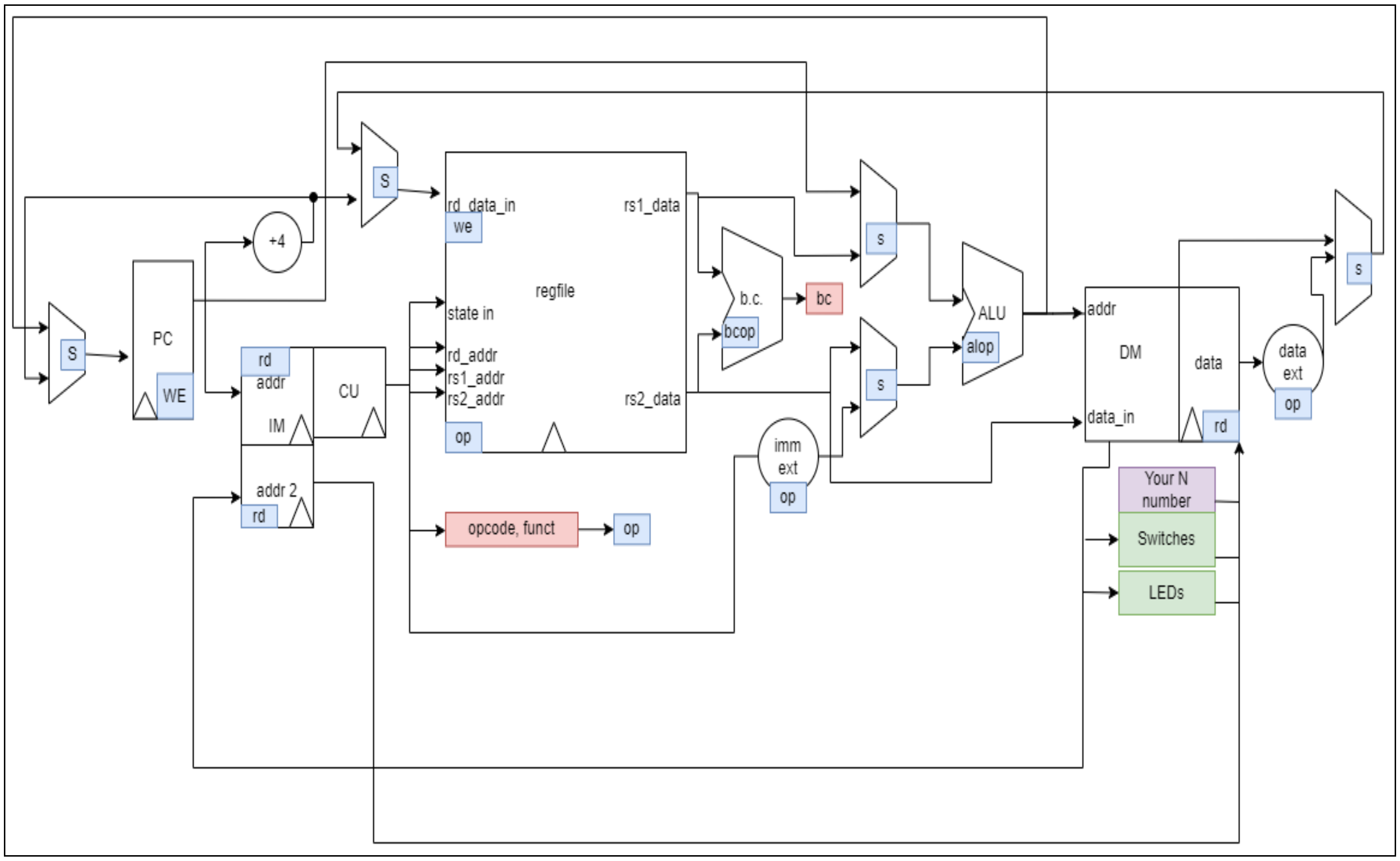
Names: Dajr Alfred, Artem Shlepchenko, Shubham Shandilya

NetID: dva240, as14836, ss15590

Group #: 5

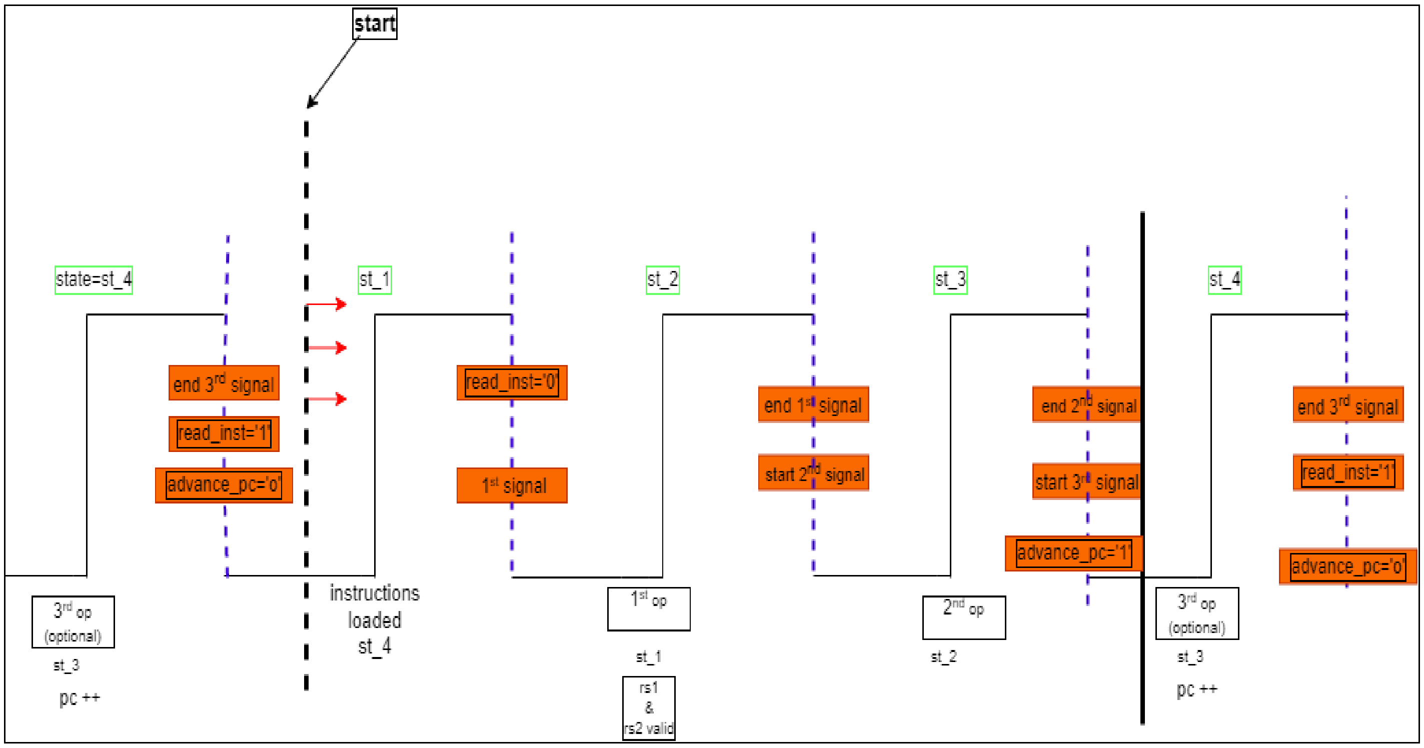
Class Project 2021

Design Datapath



The image above illustrates the datapath of our CPU. It is quite similar to the original datapath provided by Dr. Pearce. Minor differences can be seen in the addition of the Control Unit labelled CU and the addition of datapath that allows the Instruction memory to be accessed by the Data Memory. In implementation, the multiplexer (mux) serving the Program counter (PC) has been included as part of the PC. Similarly, the "+4" increment of the PC was also included in the implementation of the PC module. The data memory (RAM), N-number Memory, Switches Memory and LED memory were all added as submodules of one main Memory file. A noticeable difference to the "regfile" component is the addition of the FSM input "state_in" as well as the opcode input. These two inputs are necessary for correct implementation of JALR and Load instructions where rd and rs1 registers are the same.

FSM Diagram

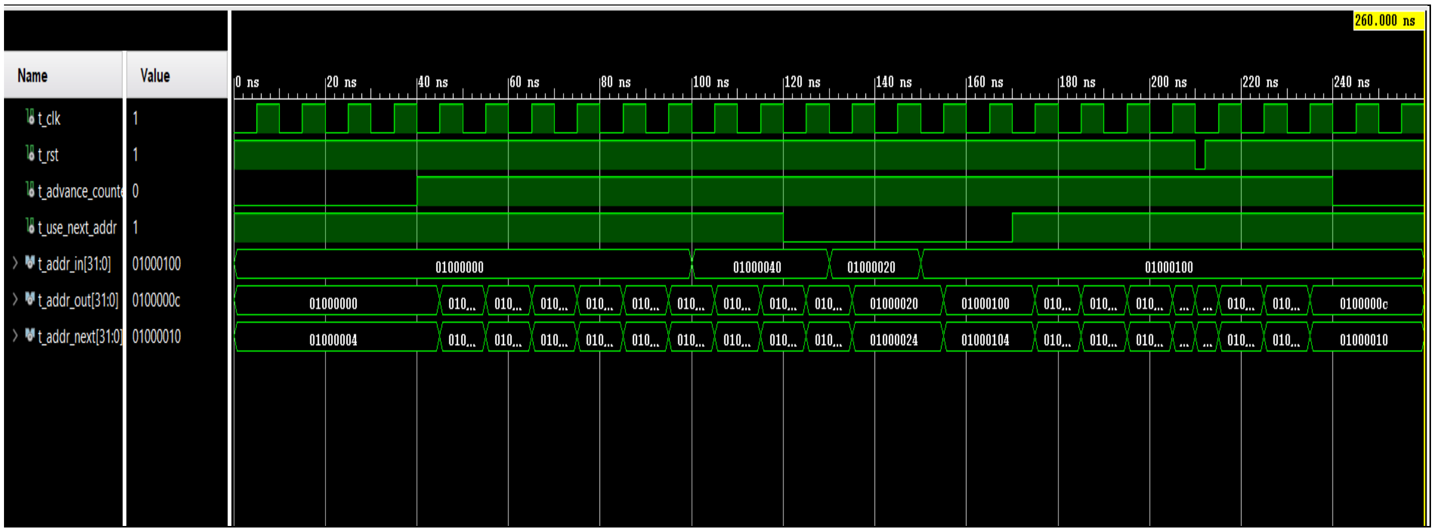


The image above illustrates the multicycle FSM operation of the CPU. There are four cycles (in green boxes) indicated above with the fourth cycle/ state being repeated to elucidate the periodic nature of the CPU operation. Relevant control signals from the control unit (in orange) are set on the falling edge while the operations they control (in black squares/bottom of waveform) occur on the rising edge. Most of the other signals from the control unit (CU) are set in a combinatorial manner and are not shown here. 1st signal, 2nd signal and 3rd signal may be read/ write enable signals for the Memory and regfile modules. There exists 1 final, fifth state which is the stop-state which is used to implement the EBREAK and ECALL instructions. **It should be noted that all instructions take four clock cycles!**

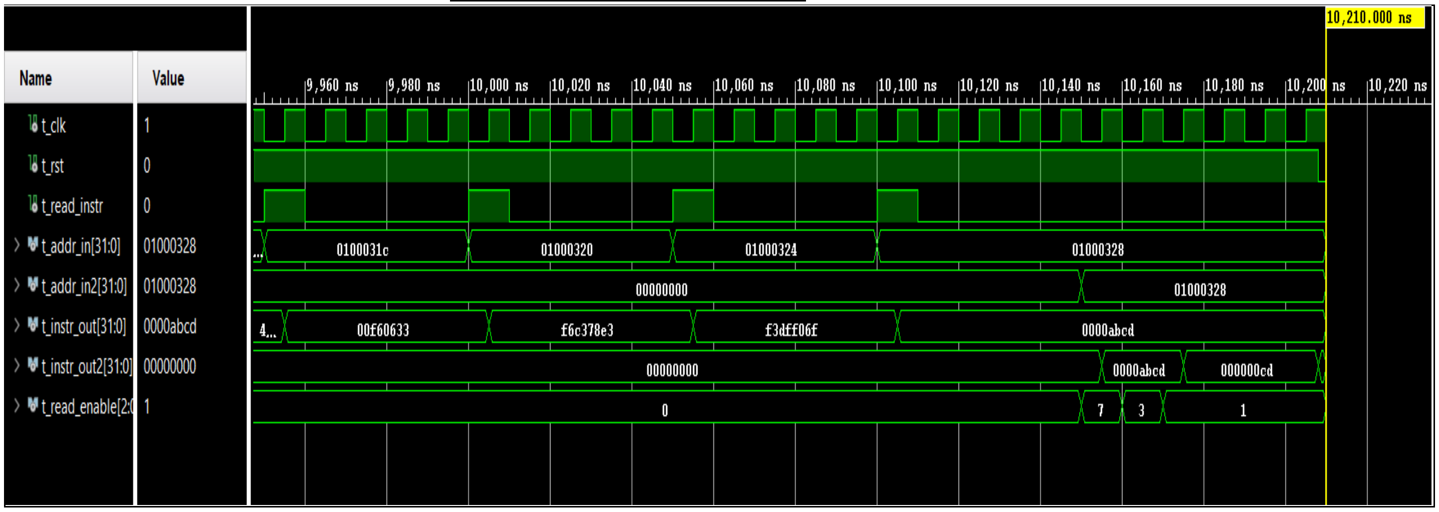
Design Validation

The following are simulation results for the testbenches of the various modules of the CPU. The time requirements for each of these tests is shown in the top-right corner of each image (in yellow). Timing requirements vary depending on the number of test cases and the time requirements per test case.

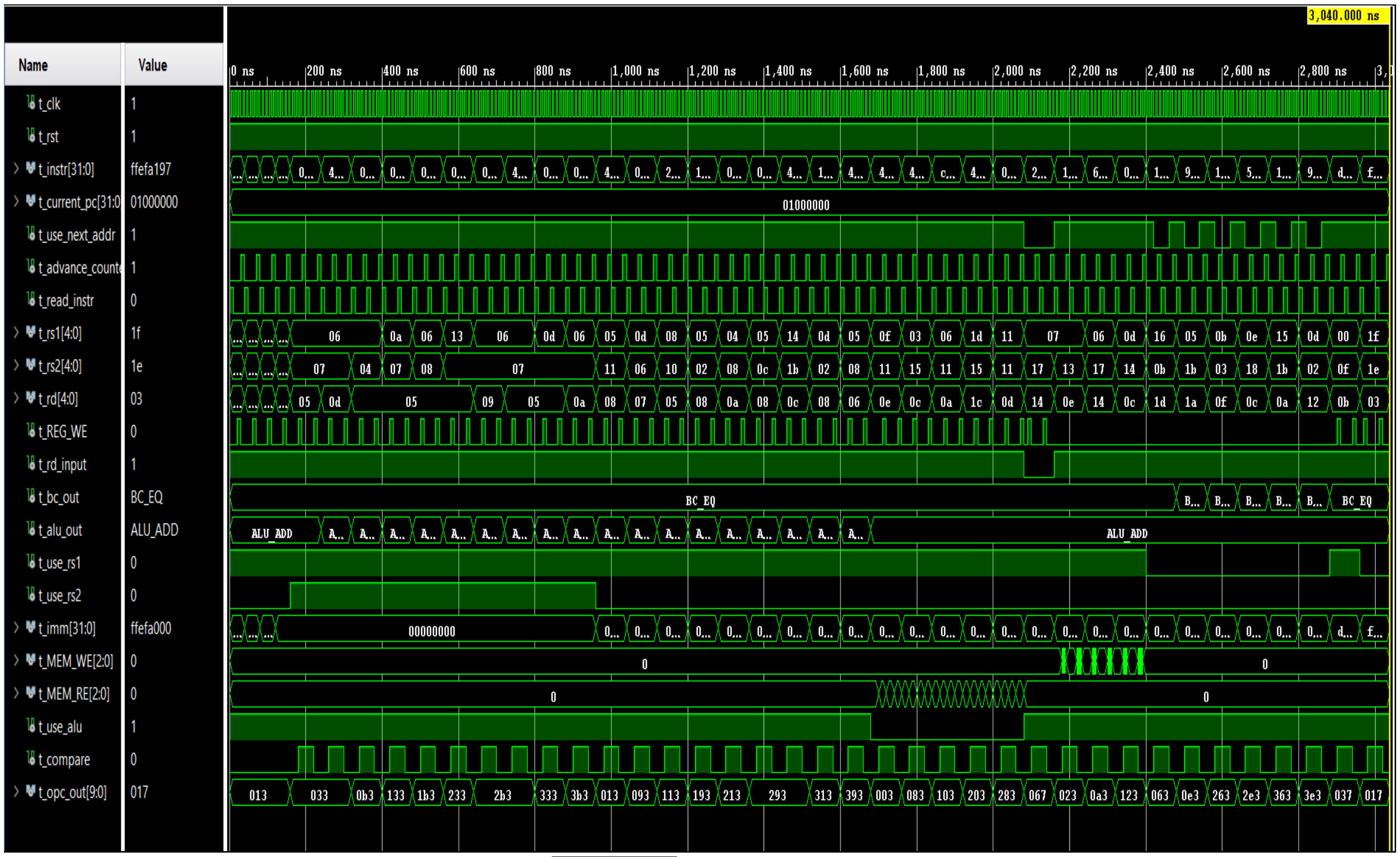
Program Counter



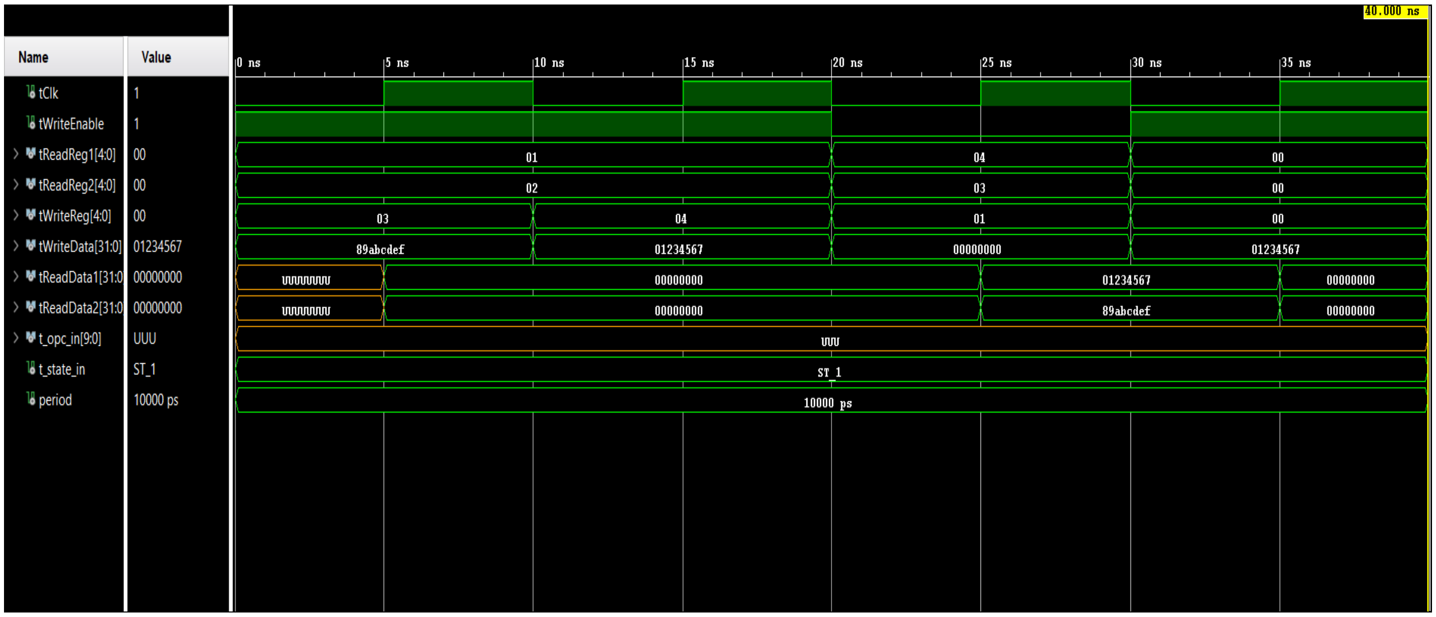
Instruction Memory



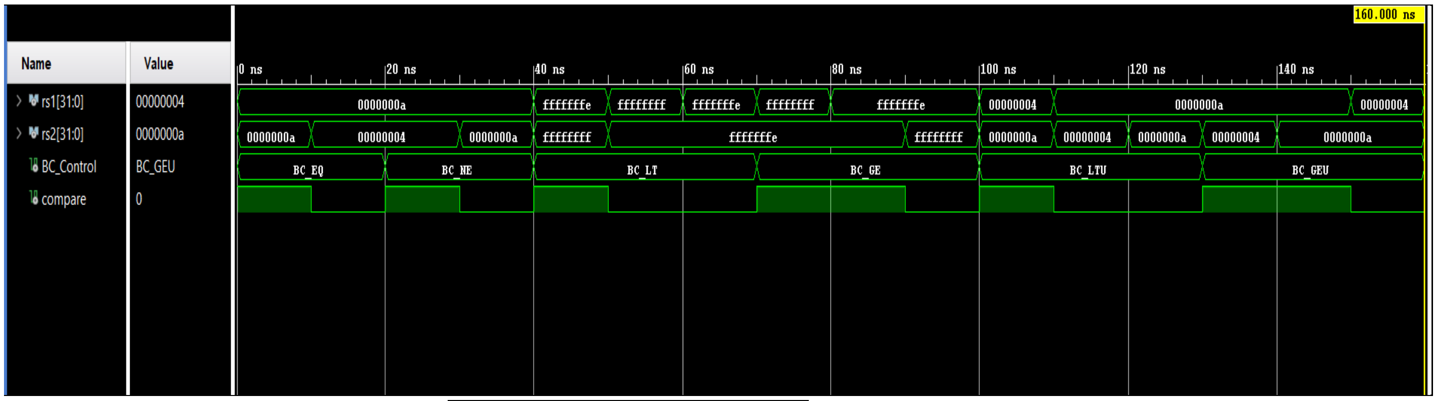
Control Unit



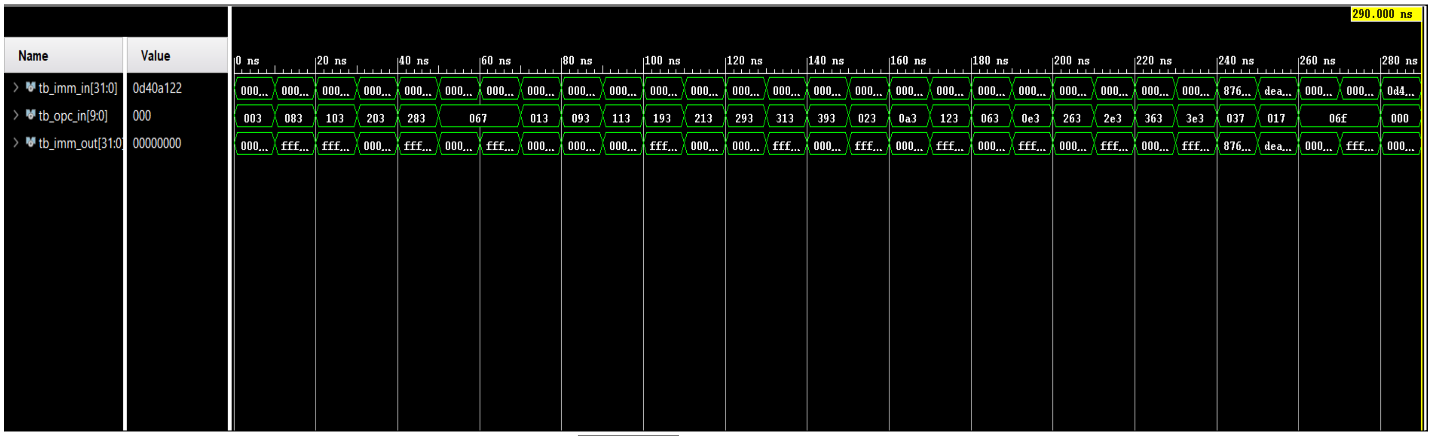
Regfile



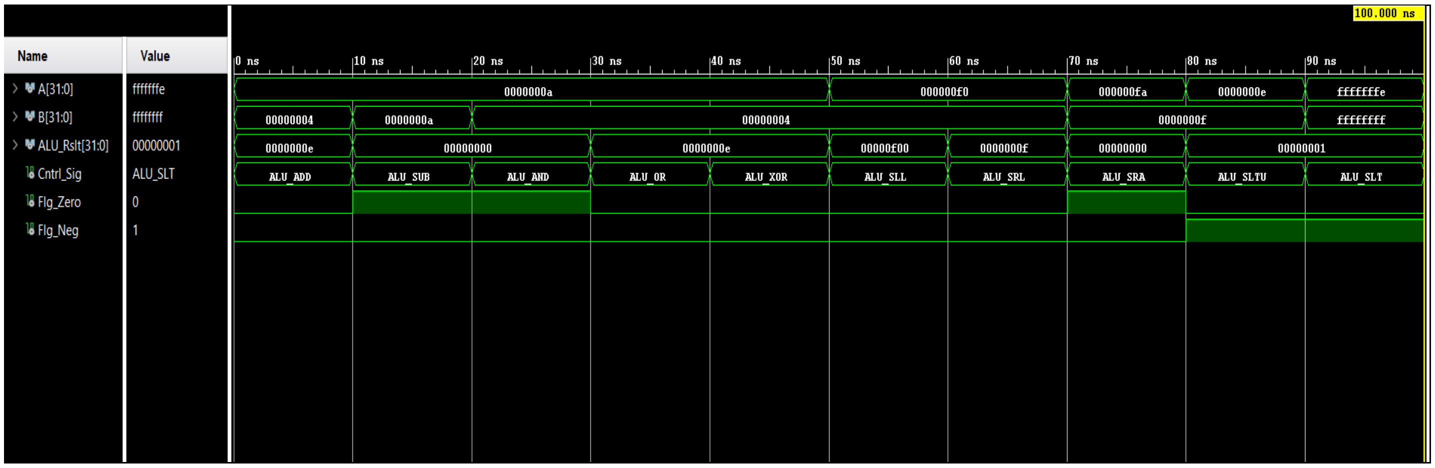
Branch Compare



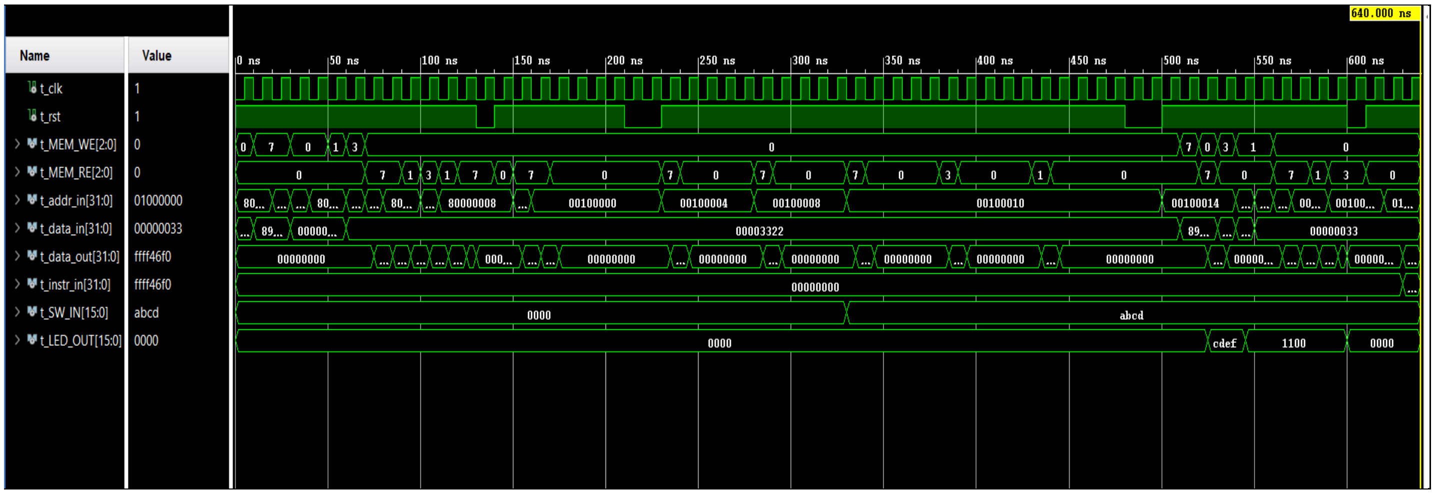
Immediate Extender



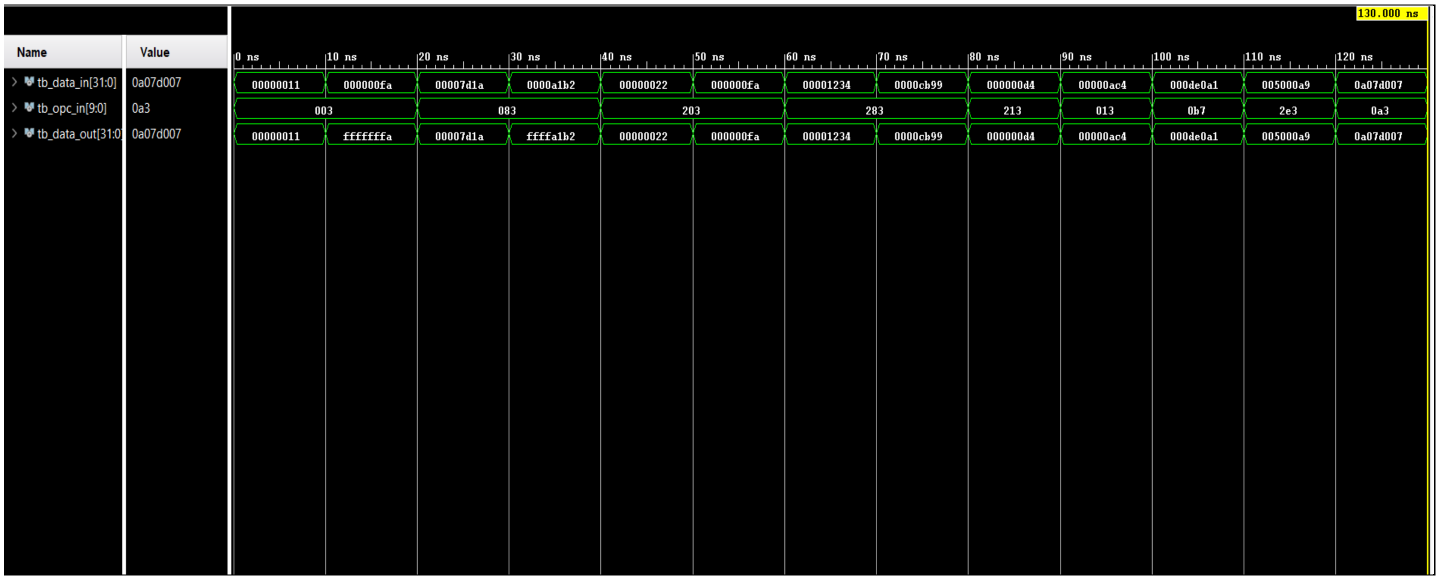
ALU



Memory

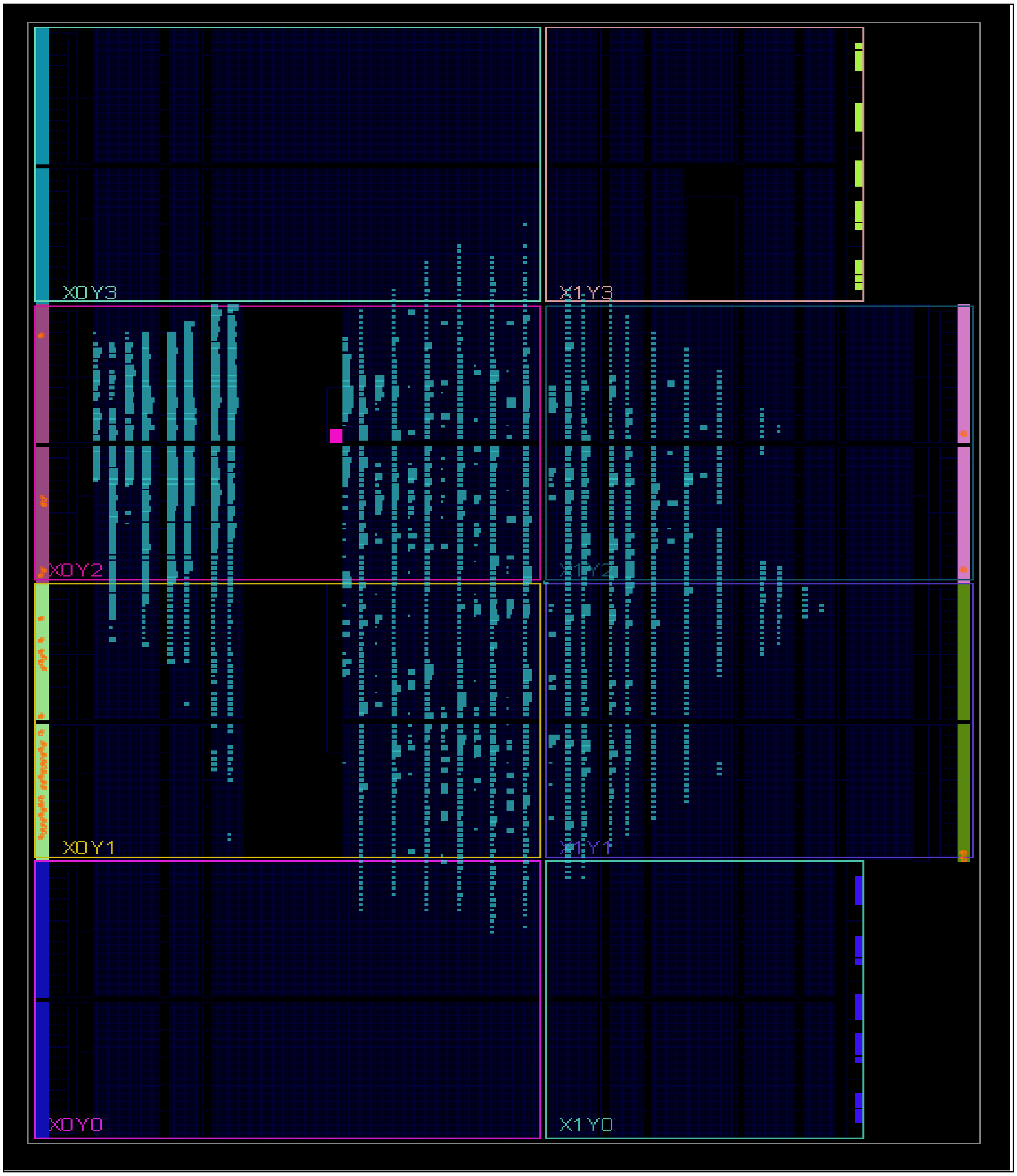


Data Extender

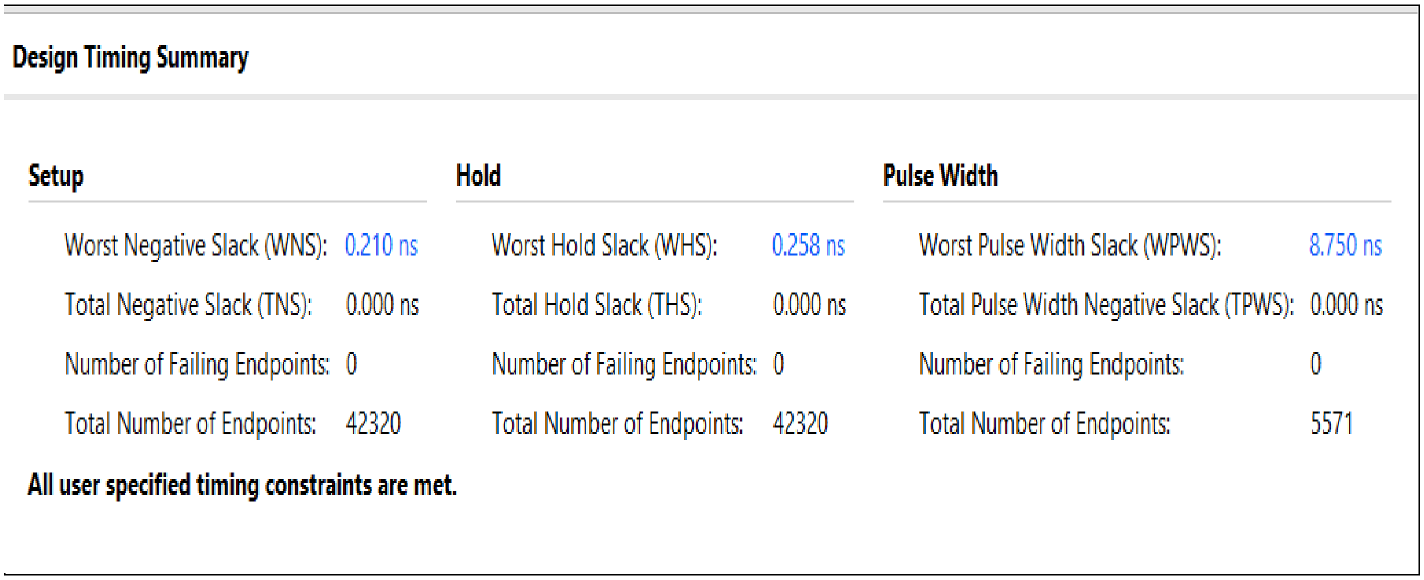


In addition to individual testbenches, the components were tested together once they were assembled into one main CPU using various assembly scripts. Individual .mem files for each of the 40 instructions were run on the CPU to ensure correct output. Short programs such as the 4-instruction program included in “Supplementary Material 2.pdf” were also run on the combined CPU to ensure proper output. **To run a program on the CPU, the assembly instructions should be copied into "main.mem"**. Any unexpected output was investigated, and the correct alterations made. Alterations were then made to the Instruction Memory module to allow for accessing of instruction memory through load instructions sent to the Memory module. Finally, since a cyclical loop can form in JALR instructions when rd and rs1 registers are equal, slight modifications were made to the regfile module. The size of the IM and DM were also adjusted from 2kB and 4kB, respectively to 4kB and 16kB, respectively. This is done by making the necessary changes to the "RV321.pkg" file and resaving each of the individual components. **It should be noted that whenever changes are made to "main.mem", the Instruction Memory module must be resaved before these changes are taken into account. This is accomplished by simply creating a blank space in the IM file and saving the file.**

Design Implementation



Timing Summary

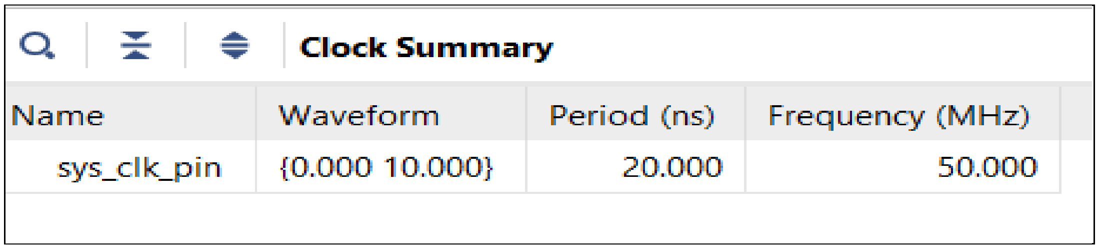


Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.210 ns	Worst Hold Slack (WHS):	0.258 ns	Worst Pulse Width Slack (WPWS):	8.750 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:	42320	Total Number of Endpoints:	42320	Total Number of Endpoints:	5571

All user specified timing constraints are met.

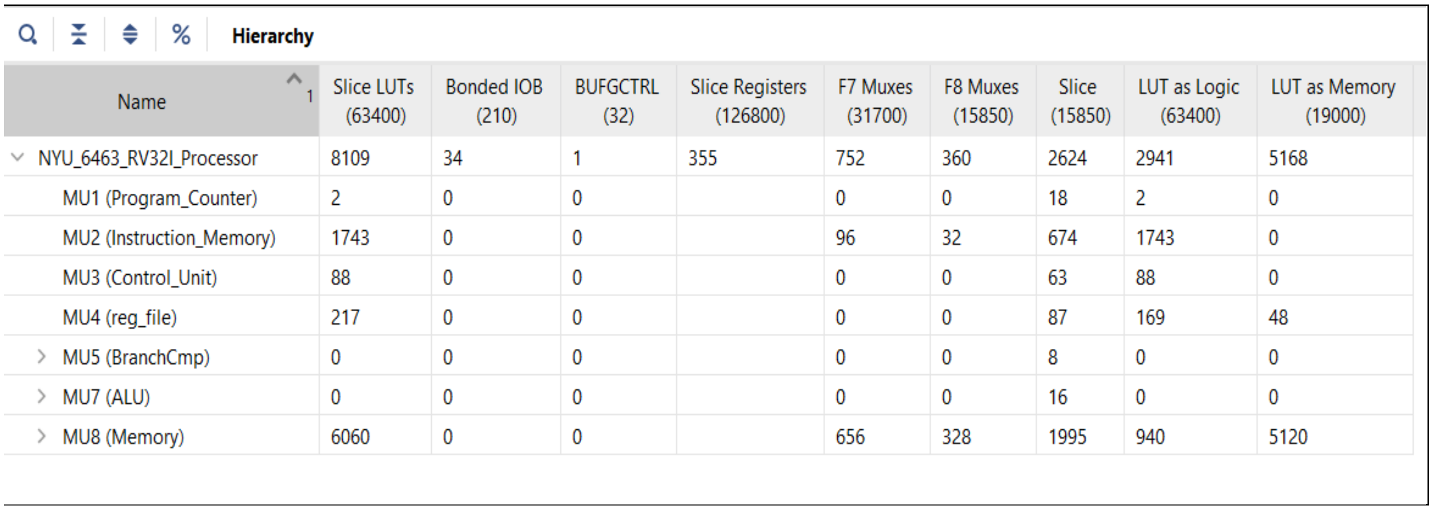
Clock Summary



Clock Summary

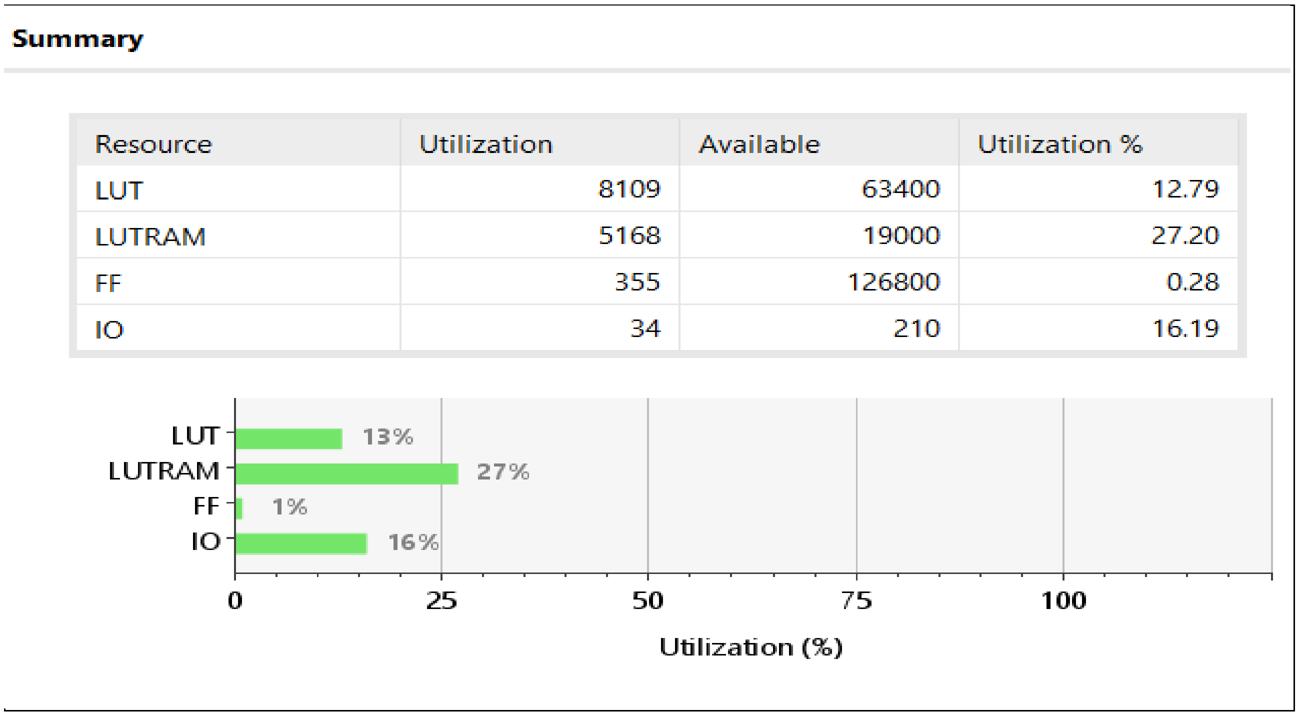
Name	Waveform	Period (ns)	Frequency (MHz)
sys_clk_pin	{0.000 10.000}	20.000	50.000

Utilization Hierarchy


Hierarchy

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)
NYU_6463_RV32I_Processor	8109	34	1	355	752	360	2624	2941	5168
MU1 (Program_Counter)	2	0	0		0	0	18	2	0
MU2 (Instruction_Memory)	1743	0	0		96	32	674	1743	0
MU3 (Control_Unit)	88	0	0		0	0	63	88	0
MU4 (reg_file)	217	0	0		0	0	87	169	48
MU5 (BranchCmp)	0	0	0		0	0	8	0	0
MU7 (ALU)	0	0	0		0	0	16	0	0
MU8 (Memory)	6060	0	0		656	328	1995	940	5120

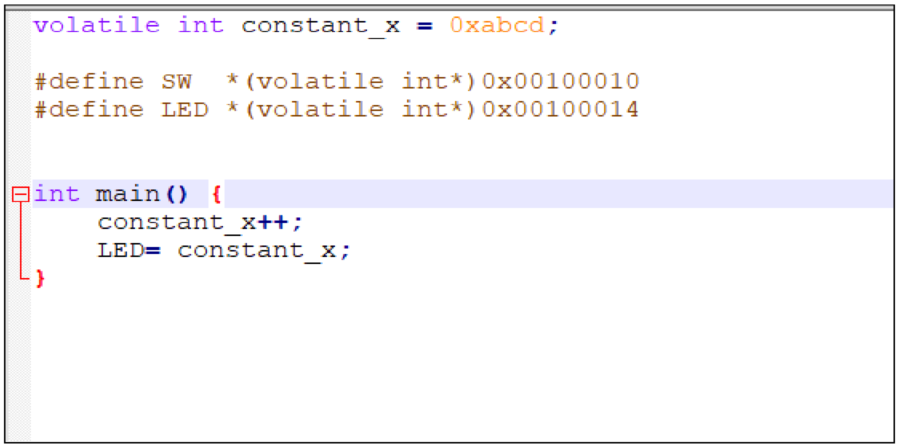
Utilization Summary


Summary

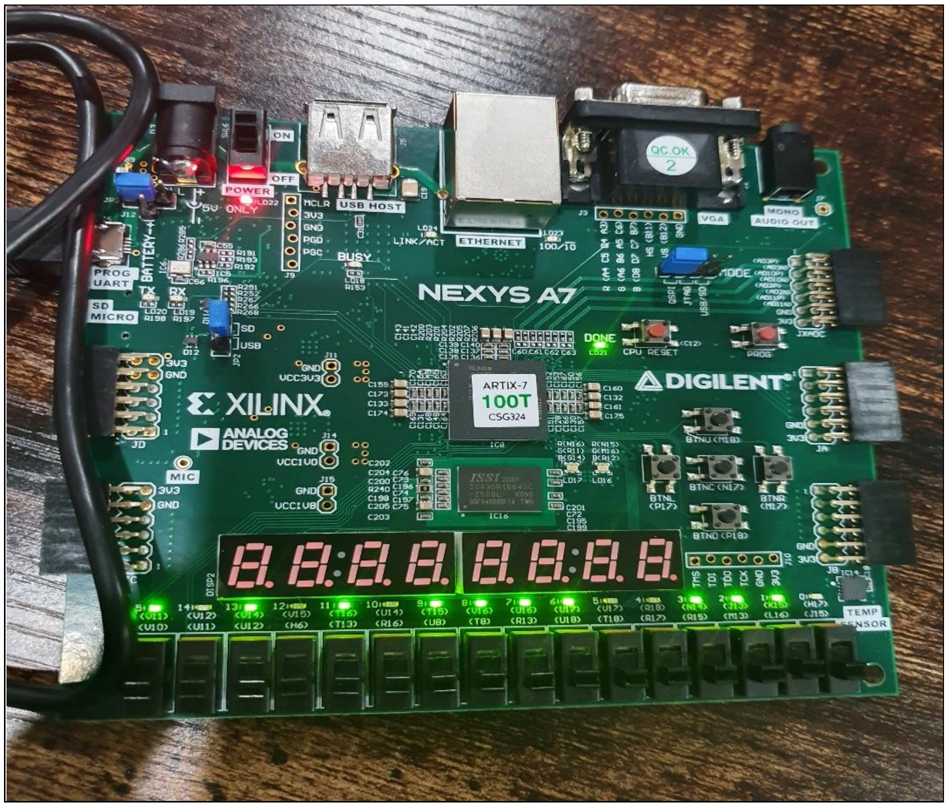
Resource	Utilization	Available	Utilization %
LUT	8109	63400	12.79
LUTRAM	5168	19000	27.20
FF	355	126800	0.28
IO	34	210	16.19

High Level C Code

1. Write Value to LED

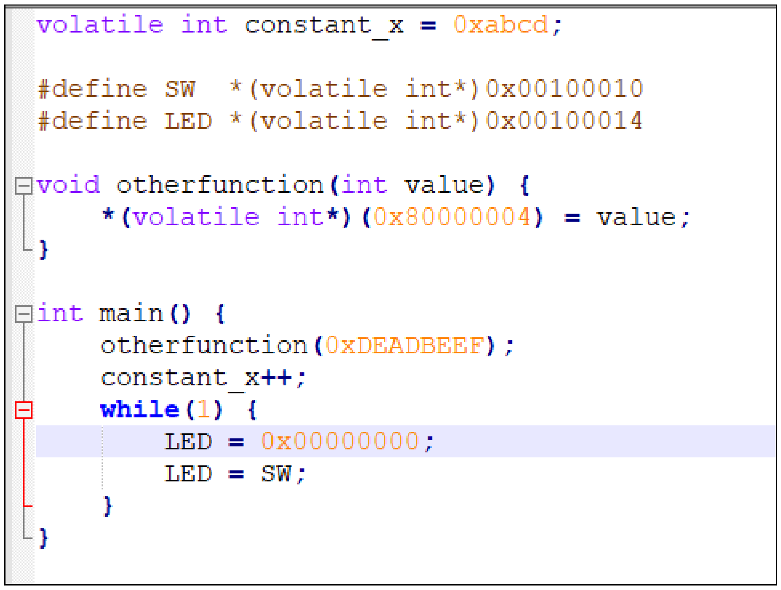


```
volatile int constant_x = 0xabcd;

#define SW  *(volatile int*)0x00100010
#define LED *(volatile int*)0x00100014


int main() {
    constant_x++;
    LED= constant_x;
}
```

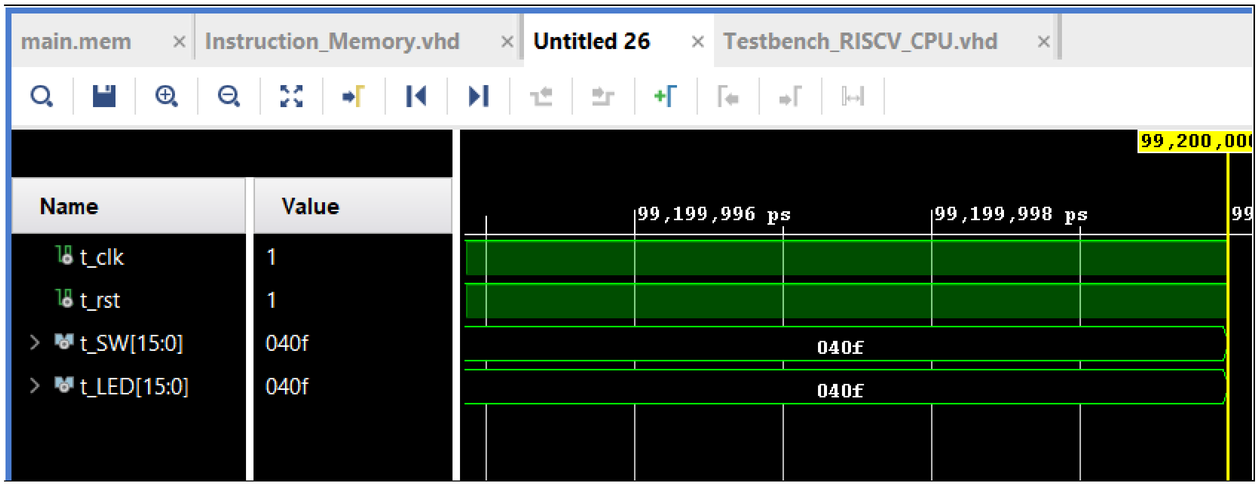


The code adds one to 0xABCD to get 0xABCE and outputs that value to the LED output. The second image clearly indicates that the LED output is binary for 0xABCE. This example takes 3.72 microseconds to run because it runs 186 lines of instructions.

2. Simulate LED Changing with Switch Input



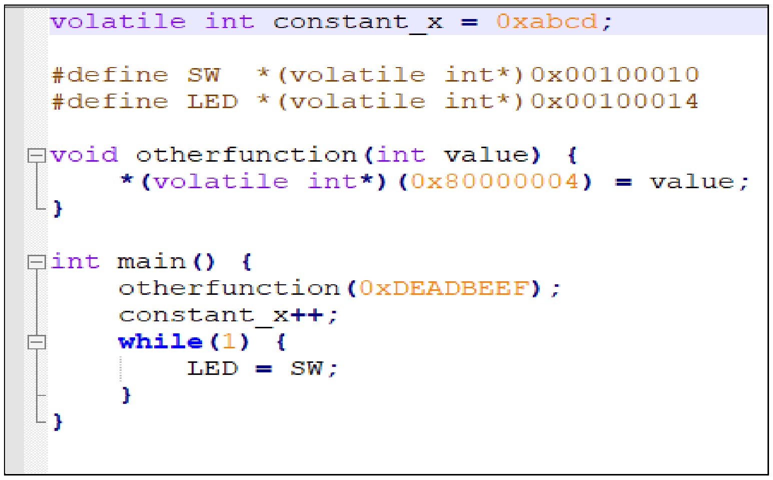
```
volatile int constant_x = 0xabcd;

#define SW  *(volatile int*)0x00100010
#define LED *(volatile int*)0x00100014

void otherfunction(int value) {
    *(volatile int*)(0x80000004) = value;
}

int main() {
    otherfunction(0xDEADBEEF);
    constant_x++;
    while(1) {
        LED = 0x00000000;
        LED = SW;
    }
}
```



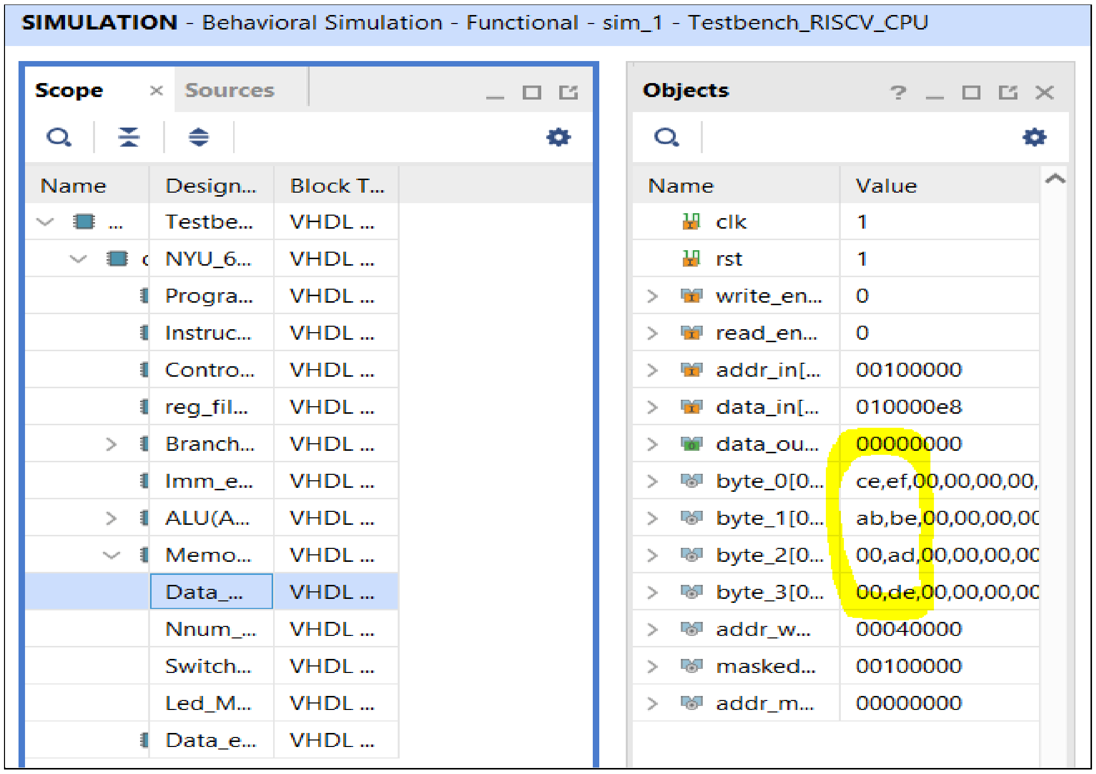
Name	Value
t_clk	1
t_rst	1
t_SW[15:0]	040f
t_LED[15:0]	040f

This example increments “constant_x” to 0xABCE and stores the value in RAM at address 0x80000000. 0xDEADBEEF is stored at address 0x80000004 in RAM. (See the output image of Program 3). Finally, the LED output is first cleared then set to match the Switches input in an infinite while loop. There is no definitive run-time for this example as it has an infinite while-loop which will continue to run indefinitely. The LED output (t_LED) matches the Switches input (t_SW).

3. Dr. Pearce’s Original Code



```c
volatile int constant_x = 0xabcd;

#define SW  *(volatile int*)0x00100010
#define LED *(volatile int*)0x00100014

void otherfunction(int value) {
     *(volatile int*)(0x80000004) = value;
}

int main() {
     otherfunction(0xDEADBEEF);
     constant_x++;
     while(1) {
          LED = SW;
     }
}
```

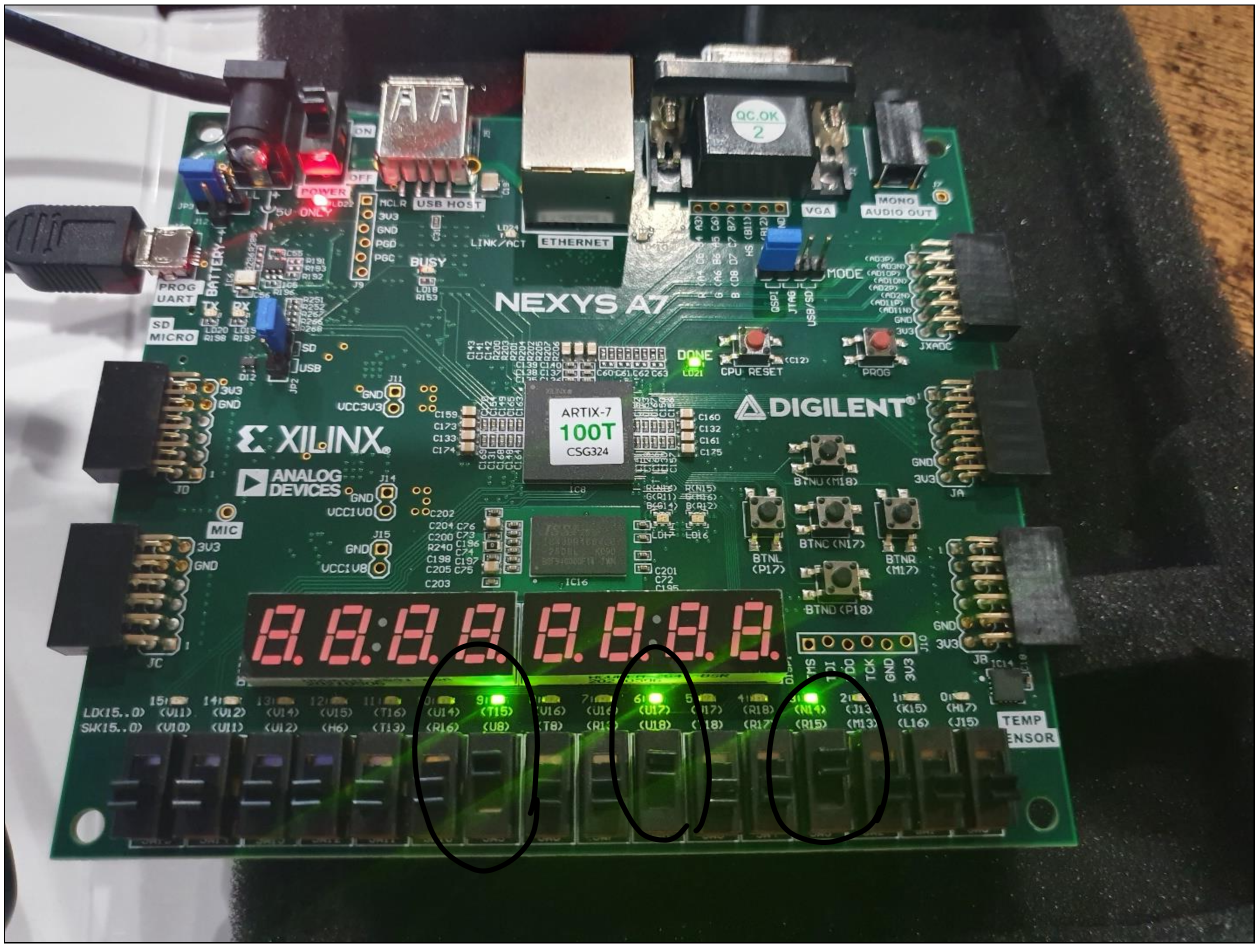


SIMULATION - Behavioral Simulation - Functional - sim_1 - Testbench_RISCV_CPU

Name	Design...	Block T...
...	Testbe...	VHDL ...
c	NYU_6...	VHDL ...
	Progra...	VHDL ...
	Instruc...	VHDL ...
	Contro...	VHDL ...
	reg_fil...	VHDL ...
	Branch...	VHDL ...
	Imm_e...	VHDL ...
	ALU(A...	VHDL ...
	Memo...	VHDL ...
	Data_...	VHDL ...
	Nnum_...	VHDL ...
	Switch...	VHDL ...
	Led_M...	VHDL ...
	Data_e...	VHDL ...

Name	Value
clk	1
rst	1
write_en...	0
read_en...	0
addr_in[...	00100000
data_in[...	010000e8
data_ou...	00000000
byte_0[0...	ce,ef,00,00,00,00,
byte_1[0...	ab,be,00,00,00,00
byte_2[0...	00,ad,00,00,00,00
byte_3[0...	00,de,00,00,00,00
addr_w...	00040000
masked...	00100000
addr_m...	00000000

This example increments “constant_x” to 0xABCE and stores the value in RAM at address 0x80000000. 0xDEADBEEF is stored at address 0x80000004 in RAM. (See the output image directly above). Finally, the LED output is set to match the Switches input in an infinite while loop. (See image below). There is no definitive run-time for this example as it has an infinite while-loop which will continue to run indefinitely.



<u>Additional examples in Assembly written by Shubham Shandilya can be found in our GitHub repository. The link is provided on the last page of this report.</u>

Possible Improvements

Clock-cycle Reduction

One possible area where improvements can be made is in the reduction of the number of clock cycles required for certain instructions, currently all instructions require 4 clock cycles. However, certain instructions such as JALR and LUI can be completed in 2 cycles quite easily. Certain operations such as loading rs1 and rs2 may occur on the falling edge of the clock which would allow for even further increases in speed. One major drawback to this would of course be added design complexity and the possibility of creating hard to find/ rectify bugs.

Memory-to-Memory Data Transfer

Another area where improvements could be made is in memory-to-memory data transfer instructions. For example, in Dr. Pearce's original code (Example 3), the assignment of the data at the Switch address to the data at the LED address requires data at one memory address to be placed at another memory address. Although placing a constant value in memory is very straight-forward, memory-to-memory data transfer still needs improvement. This may also be due to the FPGA I/O itself and should be investigated further.

Additional Instructions/ Improved C compatibilty

Additional RISC-V instructions may be added to this project and further improvement to C/ C++ compatibility undertaken.

Github Repository Link

https://github.com/dajralfred/NYU-6463-RV321

YouTube Video Link

https://youtu.be/2dY4A4t1lcg