ECE-GY 6463 ADVANCED HARDWARE DESIGN
RV32I ISA Processor Implementation: NYU-6463-RV32I Processor

Md Raz	Siddharth Kandpal	Vivek Khithani
N17762874	N10799721	N16661513
mr4425@nyu.edu	sk8944@nyu.edu	vk2279@nyu.edu

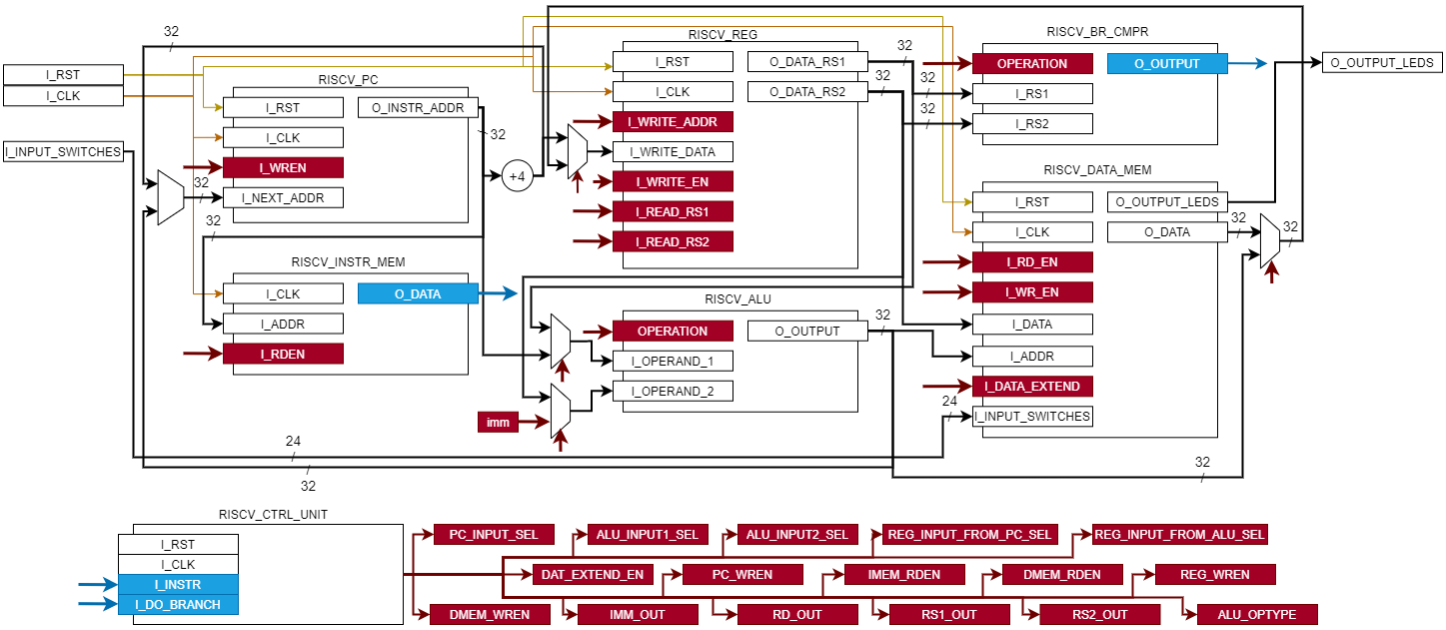
Contents:

1. Introduction

The NYU-6463-RV32I Processor is a 32-bit implementation of the open-source RISC-V 32I base Integer instruction set architecture (ISA). RISC-V, which stands for "Reduced Instruction Set Computer – Five" was created at the University of California, Berkeley, and was introduced in 2010 as an open-source alternative to existing ISAs, with the belief that providing an architecturally neutral and fully hardware implementable ISA will allow for deep levels of customization and will support a wide range of uses, from embedded systems to super computers, at the discretion of the designer.

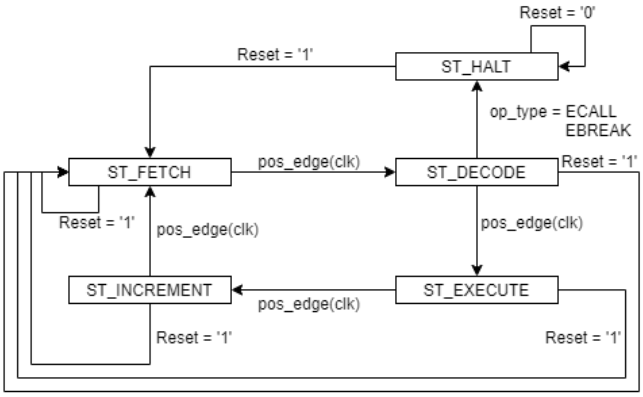
2. Datapath and FSM Overview

2.1. Datapath



The datapath diagram depicted above shows the inputs, outputs, and interconnects between the modules within the top core module of the processor. The standard inputs, clock and reset, are available and connected to several modules which are bound to the clock. The modules which are completely combination, such as the ALU, and the branch comparator, do not need the clock and reset signals. The third implemented input is the input for interfacing the switches on the FPGA board. This is a 24-bit input, as there are 24 switches present, and it connects directly to the identity matrix within the data memory module. Finally, the output is a 32-bit signal to interface the 32 LEDs present on the FPGA board, and this signal is also connected directly to the data memory module. Inside the core module exists several multiplexers which route the signals between modules in accordance with control signals from the control unit. The immediate extender is housed within the control unit, and the data extender is housed within the data memory module, and therefore they are not visible in this datapath diagram. The logic which increments the program counter output by four is also visible in this datapath. The signals depicted in blue are control signals which are inputs into the control unit, and the signals depicted in red are outputs from the control unit.

2.2. Finite State Machine

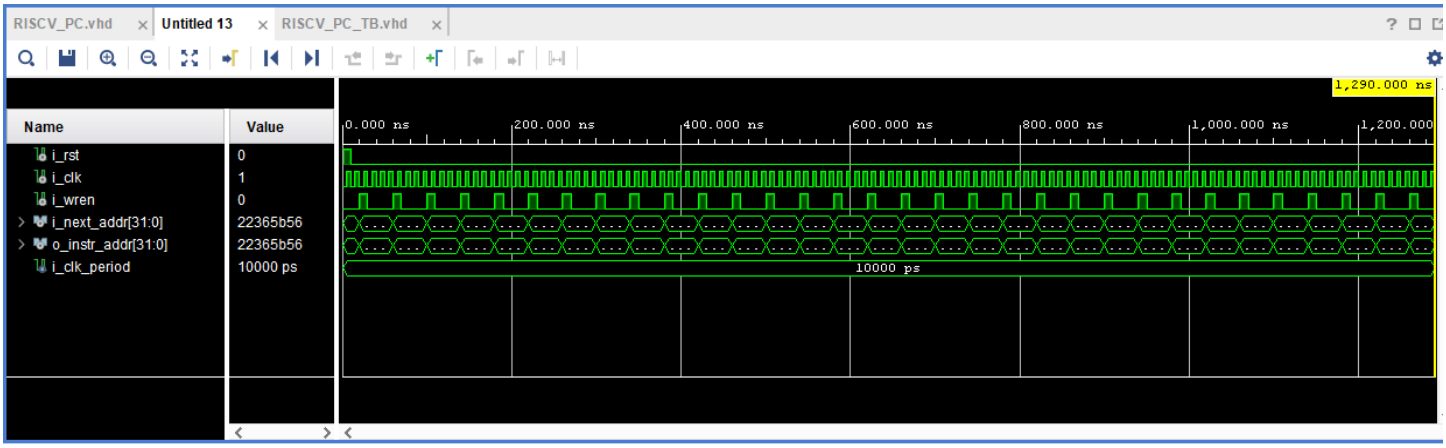


The finite state machine was built into the control unit module and was designed to execute each and every instruction within four clock cycles. The finite state machine has a total of 5 states, four of which are used during regular instruction execution, and the last of which, the HALT state, is used to suspend operations in the processor until a reset is asserted, setting the program counter back to the zeroth address. The four states used for execution are Fetch, Decode, Execute, and Increment. The decode Fetch state waits one clock cycle for the output of the instruction memory to be available to FSM. The Decode state takes this 32-bit instruction and breaks it down to an instruction type, instruction arguments, and the operation that needs to be completed. The Execution state then sets the control signals for which memories need to be accessed for current instruction read and write operations. Finally, the Increment state sets the write enable signal to the program counter in order to increment the input address or jump it to another address in the instruction memory.

3. Design Methodology and Tests

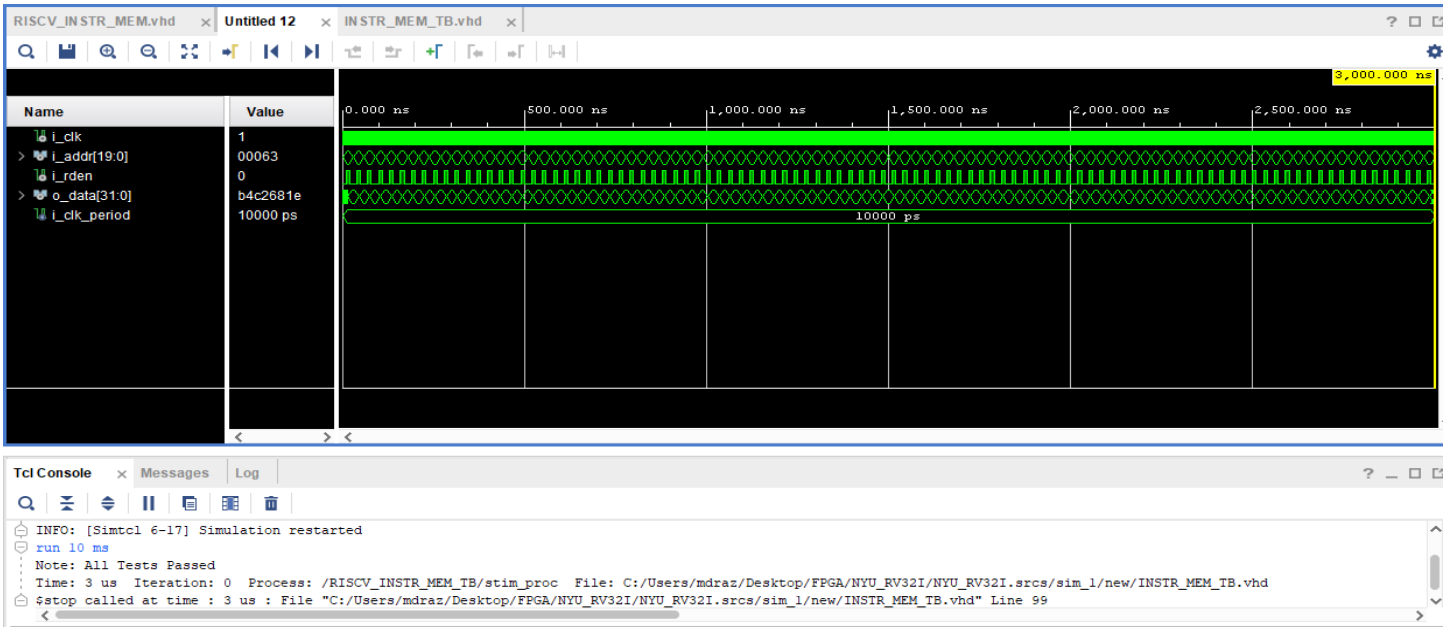
3.1. Module-Level Design Methodologies and Tests

- Program Counter



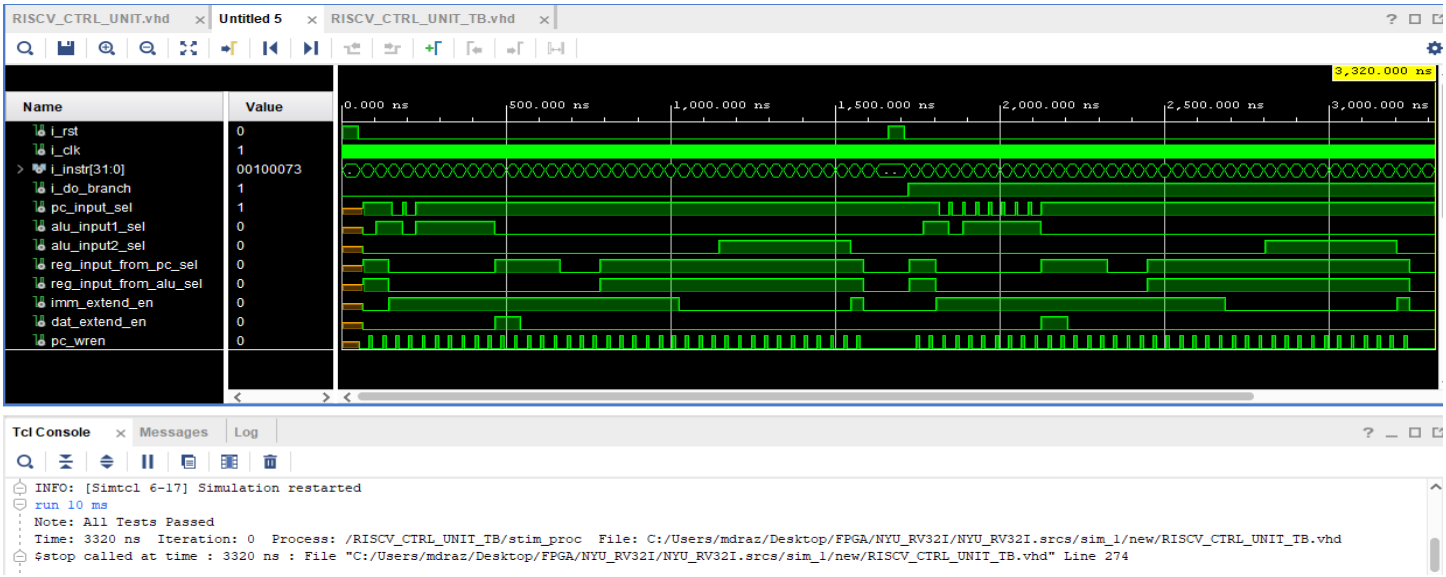
The program counter was designed as a single 32 bit register with a write enable signal input and a set of 32-bit input and output signals. The program counter also has a clock input so that the update process can be bound to a clock, making each new output signal available at the end of each clock cycle. Finally, the program counter has a reset input so that the internal register value can be reset to an initial value of the first address. The test bench for the program counter tested 32 input vectors, which were randomized 32-bit numbers to ensure that the inputs and outputs to and from the module were correct.

- Instruction Memory



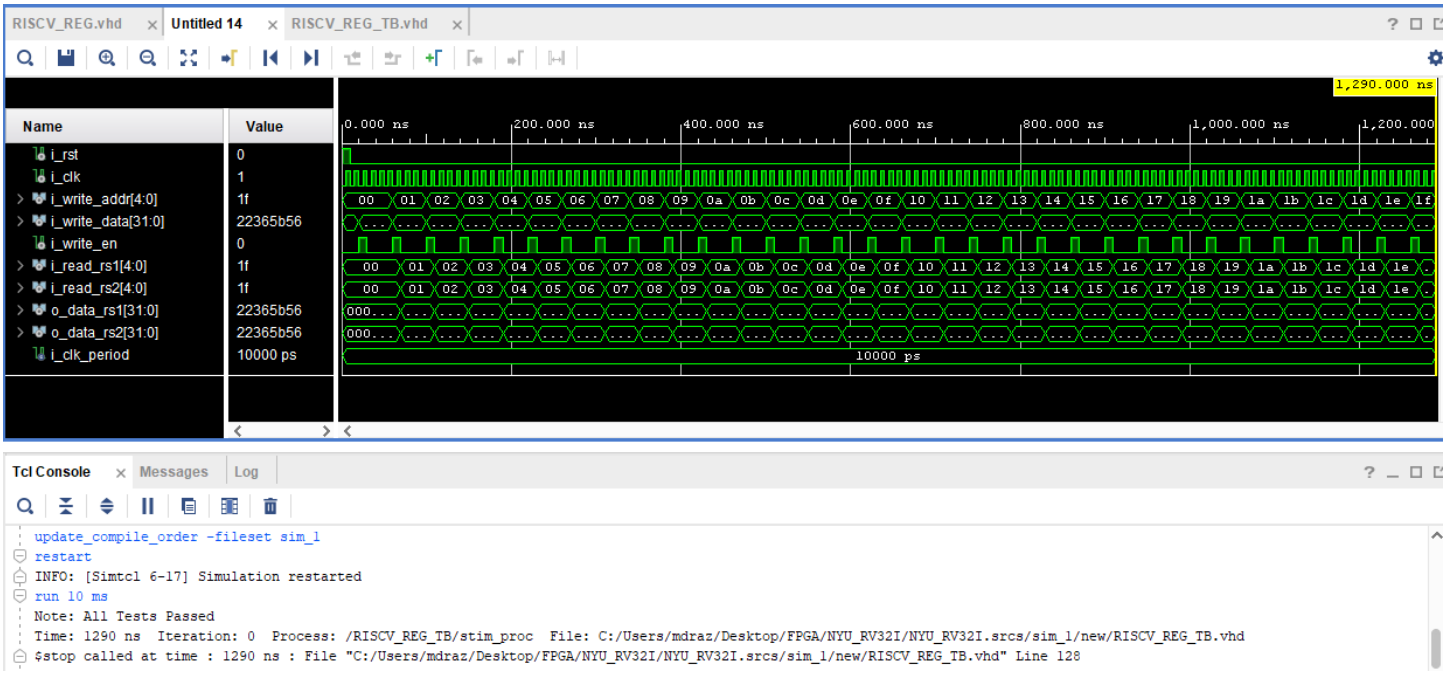
The instruction memory was designed to provide a combinational output based on the input address to be read. Due to the combinational nature of this memory, the output data is available to the control unit without using extra clock cycles. This also allows the processor to truly initialize at address zero, and not need an extra four cycles for the control unit to decode and execute the first instruction. To test the functionality of the instruction memory, the memory was loaded with 100 random 32-bit vectors and then the outputs were tested against the same output vectors for each instruction address.

- Control Unit



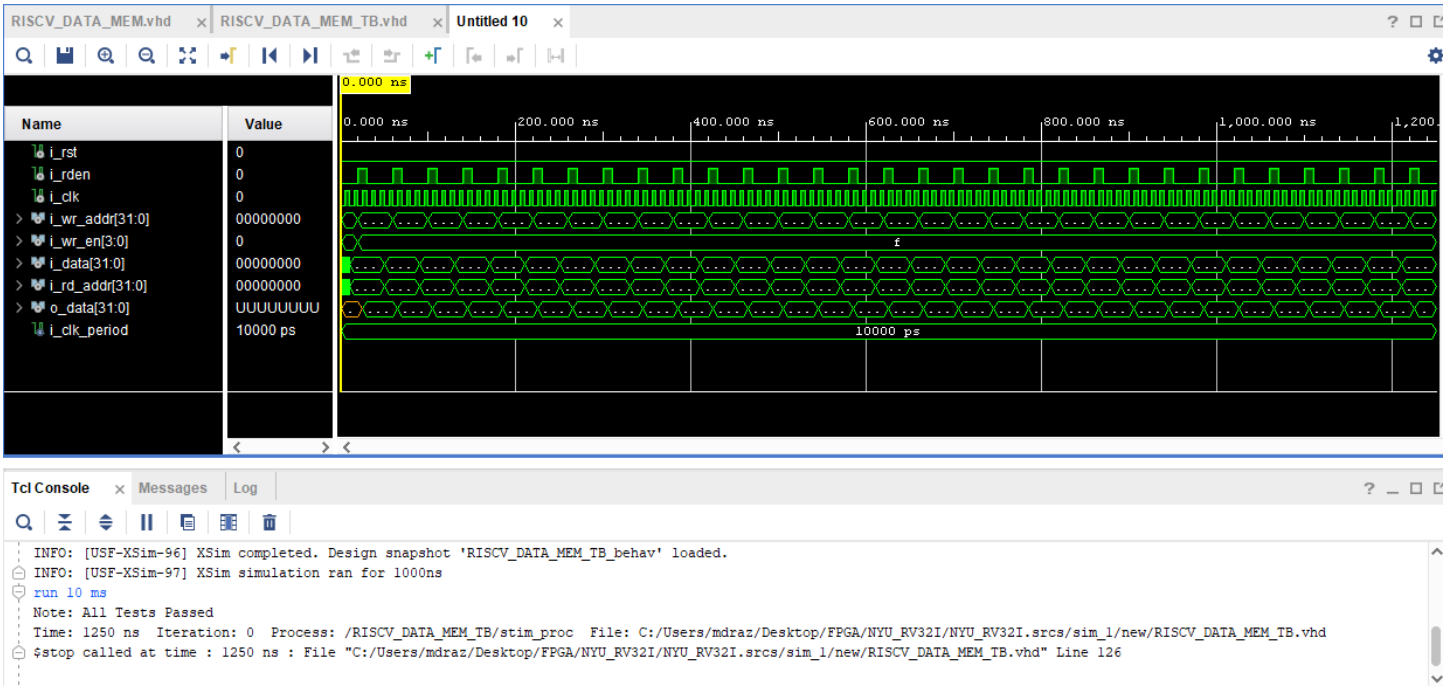
The control unit was designed to incorporate both the FSM and the immediate extender, as the immediate value is always zero or sign extended, and this can be done immediately as the instruction is decoded. The control unit provides two inputs, one for the 32-bit instruction, and a 1-bit signal from the branch comparator so that the control unit can jump to a different instruction address using the program counter if needed. The control unit also provides 16 outputs, most of which are 1-bit control signals, such as MUX selection outputs and read/write enables, while others are the inputted arguments from the instruction such as destination register addresses. The control unit also outputs a special enumerated "op-type", which signals the ALU and Branch Comparator so that those modules know which operation to execute or check for. The control unit was tested for 80 input vectors across two test vectors files. Each of the test vector files contained a test vector of every instruction, while one test vector file tested jumps when the other one did not.

- Register File



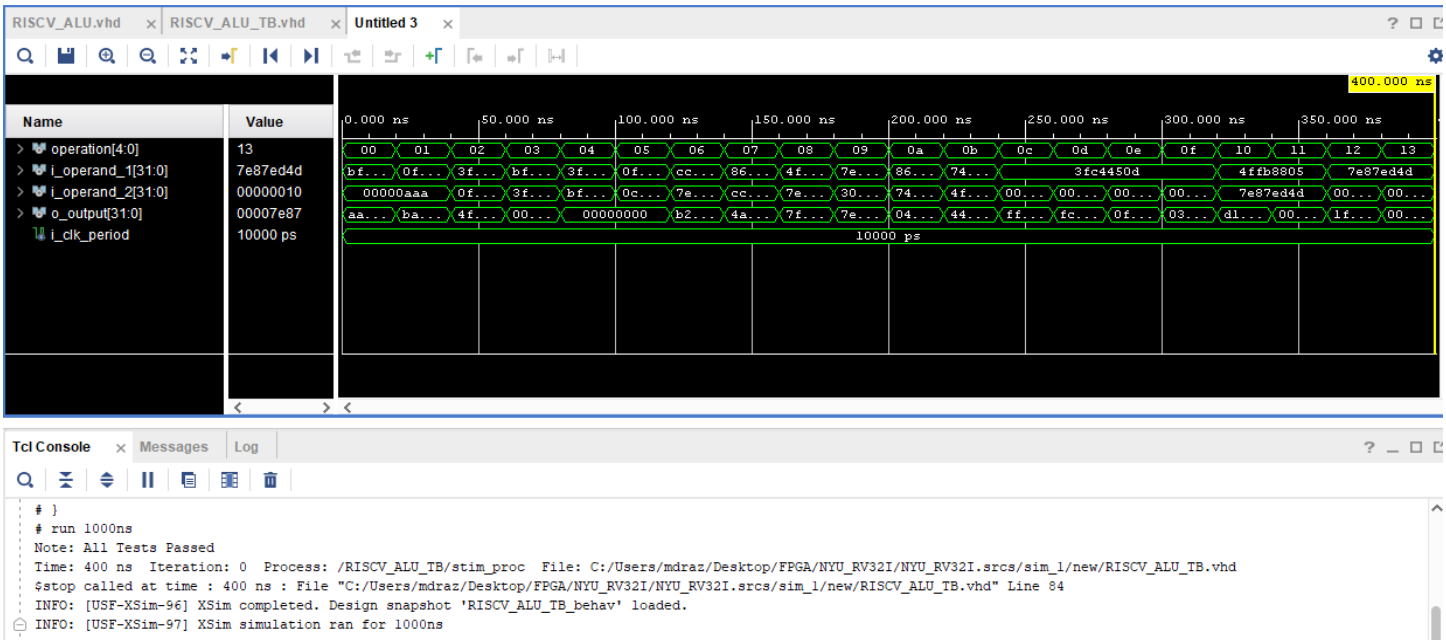
The register file was implemented as 32 row memory with 32-bits in each row. The Register file can read data values from the memory using combinational logic, while writing values is bound to the clock and consumes one clock cycle. The register file was tested by writing each of the 32 rows of the memory with a random 32-bit test vector and then reading form that location and ensuring that the data was written and read correctly.

- Data Memory



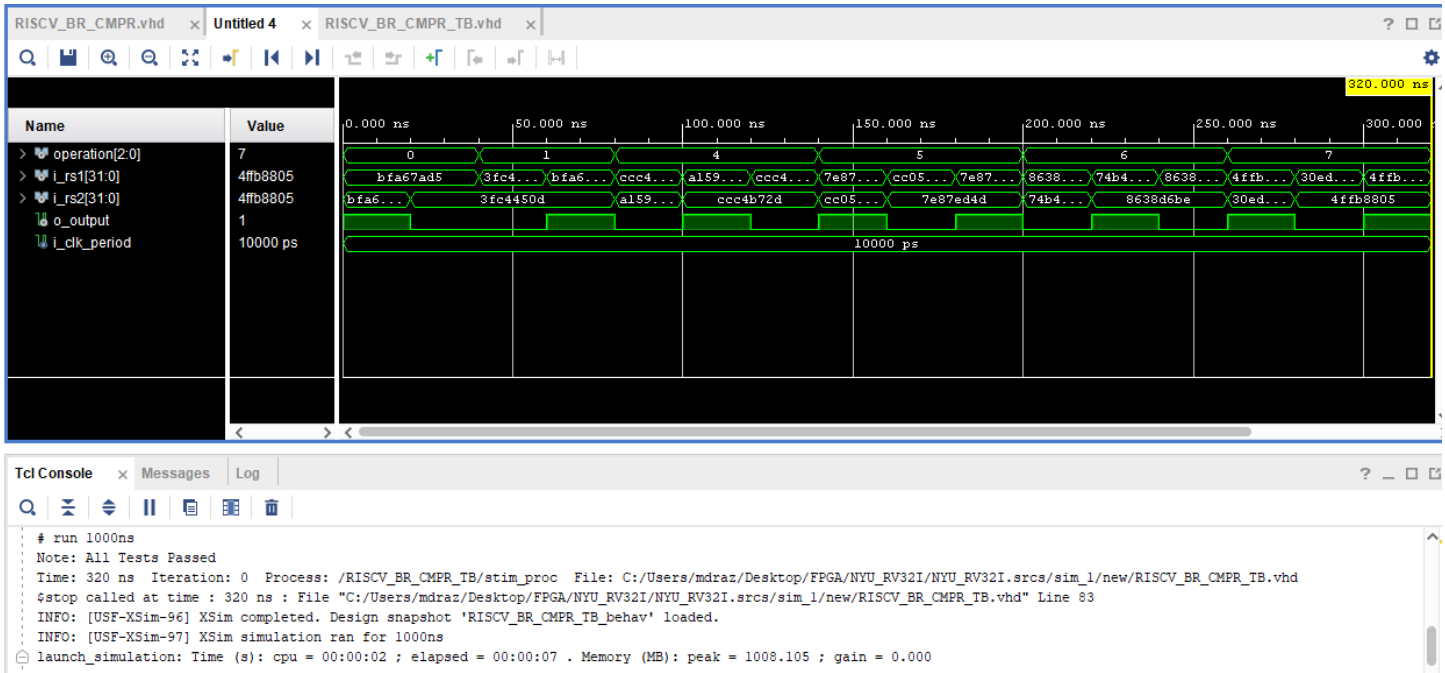
The data memory module was designed to have the standard inputs and outputs for the memory, which included write enable, read enable, address, and 32-bit input/output data signals. The module also included inputs and outputs that directly interface the pins that connect to the LEDs and switches on the FPGA board, and a single 1-bit input to signify if the data is zero-extended or sign-extended. The write enable uses four bits to signify if the write is a byte, halfword, or word, and the read enable signal uses two bits to signify the same. Both processes for reading from and writing to the data memory are bound to the input clock, and both processes take one clock cycle to execute. The data matrix in the data memory has a size of 1024 rows, each row holding a 32-bit vector. This equates to a storage size of 4 kilobytes. The identity matrix inside the module has 6 rows, which include the N numbers, the rows for LEDs / switches, and then an empty row. The data memory module was tested using random writes and reads to different locations, and then tested for the proper N number output for 32 vectors. The module also includes the data extender to directly zero-extend or sign-extend the output data before it leaves the module, depending on the control extension enable signal from the control unit.

- Arithmetic Logic Unit



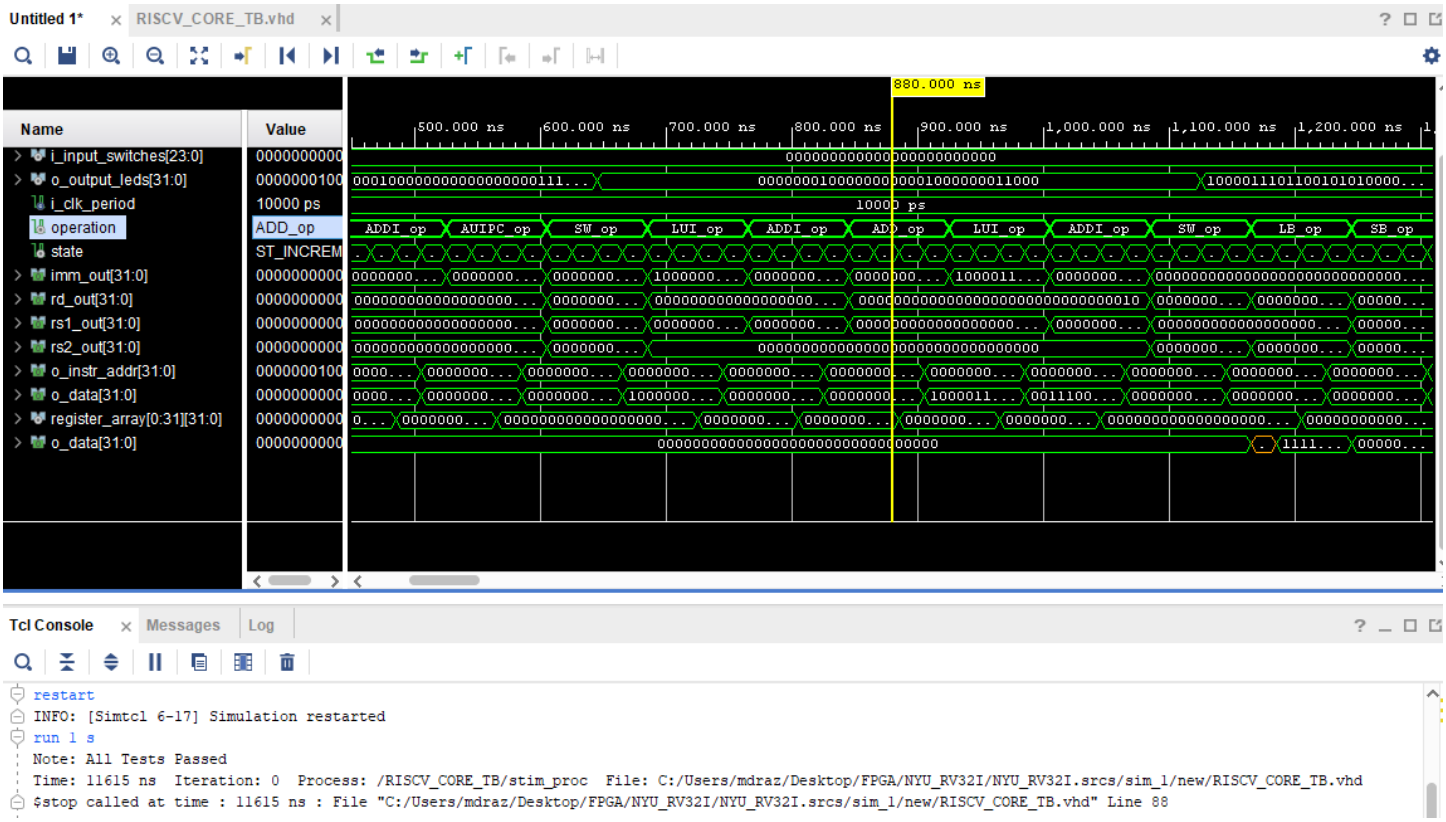
The ALU has two 32-bit inputs for operands, a single input for operation type, and then a 32-bit output signal. The operation type signal comes from the control unit and tells the ALU which operation to perform. Inside the ALU module, the input operation is taken and turned into one of 9 different operations. 20 of the 37 implemented instructions will do an addition operation in the ALU, while the other 17 instructions will subtract, “XOR”, “OR”, “AND”, left shift, right shift, or check for greater than against the two input operands. The test bench for the ALU used 20 test vectors to check that all of operations were working with different operands. With fully combination logic, the ALU does not consume any clock cycles to execute any operations, and will provide an output directly after data is available to the input signals.

- Branch Comparator



The branch comparator was implemented to have two 32-bit inputs, and one 1-bit output which goes to the control unit on order to tell it whether to branch to a different instruction address. The Branch Comparator will take one input and check whether it is equal to, not equal to, less than, or greater than the other input, and will also take into consideration whether the input signal is signed or not. After this, it decides the output based on the inputted operation, and will provide a branch signal if the operation matches and the test between the two inputs is a '1'. The test bench for the branch comparator used 20 test vectors to check that all the operation worked using both signed and unsigned input signals.

3.2. Core Design and Integration Tests



The top module was designed to have inputs for the switches and outputs for the LEDs on the FPGA board. It also has standard inputs such as clock and a high reset. The top core module contains all the other modules in a hierarchical fashion and also contains all signals to connect each of the modules with each other. Furthermore, the top module contains all the logic to route signals with multiplexers, which are controlled from signals coming from the control unit. The top module was tested using compiles and linked assembly, which runs every implemented instruction and checks that the output is correct by writing to the data memory LED array location using the LED output pins. The test bench was tested for 60 outputs using about 240 lines of assembly.

4. Area and Performance Analysis

The FPGA board used is the Alchitry Au FPGA Development Board (Xilinx Artix-7 XC7A35T-1C).

4.1. Area Analysis

Once the processor was implemented and placed, the component usage was found to be as follows:

Site Type	Used	Fixed	Available	Util%
Slice	844	0	8150	10.36
LUT as Logic	1512	0	20800	7.27
LUT as Memory	512	0	9600	5.33
LUT as Distributed RAM	512	0		
Slice Registers	1279	0	41600	3.07
Unique Control Sets	50		8150	0.61

The synthesis and place-and-route process successfully inferred RAM for the data memory, and found that the RAM would be fastest if implemented as a distributed RAM rather than a block RAM. Other than this, the processor uses 10.36% of all slices available on the Artix-7 XC7A35T-1C, which was the FPGA integrated on the board we were using. The total number of LUTs used for logic was 1,512, and then 512 more were used for implementing the RAM. The total number of registers implement in this design is 1,279. With a small utilization footprint such as this, this processor implementation would be a good candidate for multicore implementations, as on this FPGA alone a total of 9 processors can be fit with room left for extra logic.

4.2. Performance Analysis

Using an initial constraint for input timing of 100 MHz, the processor was found to be able to run at a maximum speed of about 75 MHz, since the critical path, which was a clock signal coming from the control unit going into the data memory, had a delay of 13.2 ns. In order to properly implement this on the FPGA board, the input clock speed had to be reduced so that timing constraints would be met during operation. A clock divider in the top module would be needed as the physical clock on the FPGA board is fixed to 100 MHz. The following clock divider was implemented in the top core module of the processor, which divided the input clock speed by 2, making available a 50 MHz clock to all the modules.

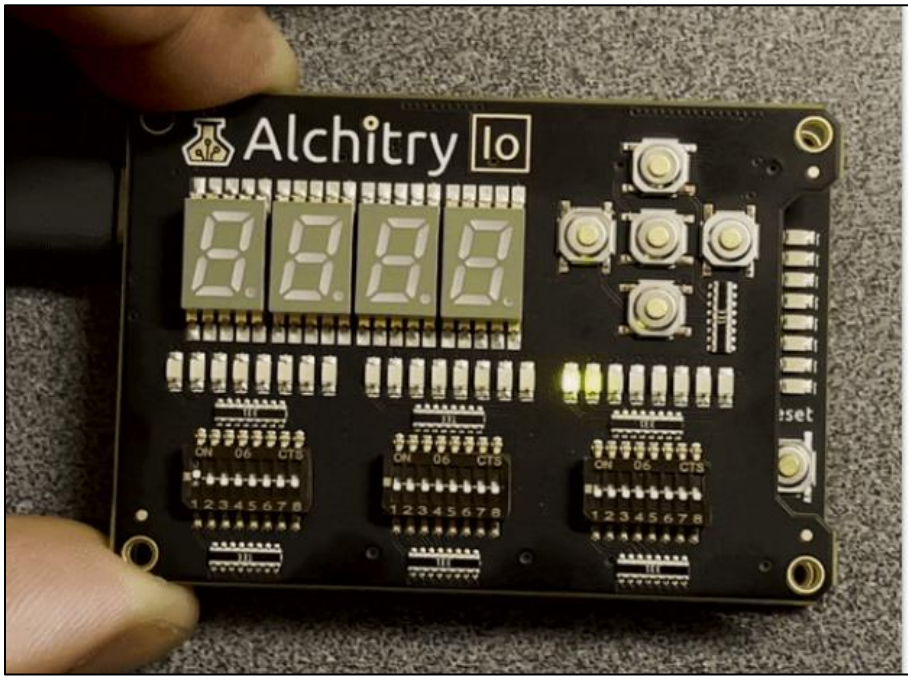
```
clk <= not clk when (RISING_EDGE(i_clk));
```

Once the clock divider was created, the timing reports were once again generated post place-and-route. The timing report, as shown below, shows that the maximum path delay is 11.257 nanoseconds, as the clock period is 20 ns and the slack is 8.743 ns. All the timing constrains were met, as the slack of 8.743 is positive and allows the processor to run faster than the 10 ns requirement before the clock divider.

$$Max\ Path\ Delay = Clock\ Period - Slack = 20\ ns - 8.743\ ns = 11.257\ ns$$

```
Max Delay Paths
--------------------------------------------------------------------------------------
Slack (MET) :             8.743ns  (required time - arrival time)
  Path Group:             i_clk
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            10.000ns  (i_clk rise@10.000ns - i_clk rise@0.000ns)
  Data Path Delay:        1.253ns  (logic 0.580ns (46.298%)  route 0.673ns (53.702%))

  ----------------------------------------------------------------------
                          required time                         15.145
                          arrival time                          -6.402
  ----------------------------------------------------------------------
                          slack                                  8.743
```

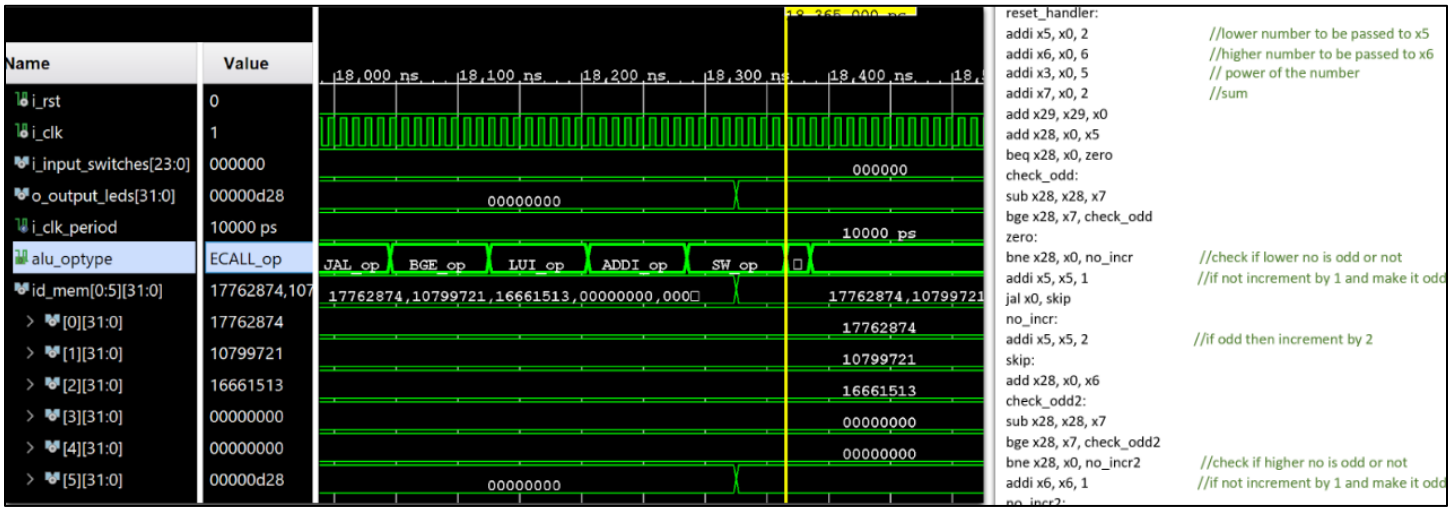
5. High Level Test Cases
   5.1. LED Sweeping Assembly Program on FPGA




```
reset_handler:
        LUI   x1, 0x00100
        ADDI  x2, x1, 0x10  // Switch Input register
        ADDI  x1, x1, 0x14  // LED Output Register
        ADDI  x3, x0, 1     // Increment amount register, =1
        LUI   x9, 0x00020   // Final Value register = 0x20000
        ADDI  x8, x0, 0     // Count register
RESET:  LUI   x10, 0x80000  // init Shift register, right
        ADDI  x7, x0, 1     // init Shift register, left
        // Find switch state
        LW    x5, 0(x2)     // if 0, SLL, if 1, SRL
        BEQ   x5, x0, LINIT // Branch based on switch
        BEQ   x5, x3, RINIT
LINIT:  SW    x7, 0(x1)     // Load LED w/ LShift val
        JAL   x4, INCR      // Start Counting
RINIT:  SW    x10, 0(x1)    // Load LED w/ RShift val
        JAL   x4, INCR      // Start Counting

SLFT:   SLL   x7, x7, x3    // LEFT SHIFT * 1
        ADDI  x8, x0, 0     // reset count
        BEQ   x7, x10, RESET// Reset if reached end of array
        JAL   x4, LINIT     // Otherwise start next count
SRHT:   SRL   x10, x10, x3  // RIGHT SHIFT * 1
        ADDI  x8, x0, 0     // reset count
        BEQ   x10, x7, RESET// Reset if reached end of array
        JAL   x4, RINIT     // Otherwise start next count

INCR:   ADD   x8, x8, x3    // Increment counter
        BNE   x8, x9, INCR  // Branch if not final value
        BEQ   x8, x9, SHIFT // Shift if final value is reached

SHIFT   BEQ   x5, x0, SLFT  // CHeck of left or right Shift
        BEQ   x5, x3, SRHT
```

The first high level test case included writing an assembly program to sweep the onboard LEDs. This assembly file is given in the “Complex Test Cases” folder as “led_sweep.s” within the project directory. The program first initializes several registers and then then uses a counter and a shift register to sweep the LEDs along the board. To further increase the complexity, and input signal was taken from one of the switches on the board, and this input was used to indicate the direction in which the LEDs should move. In assembly, this translated to different starting and ending values for the shift register, and then the shift direction is then dictated by the switch value. The counter is set such that when the counter value is 0x20000, the assembly will branch and then do a shift in the shift register. Once the shift is complete, the register value is then written to the data memory location which holds the LEDs. In this design, the FPGA input clock is 100 MHz, and a clock divider allows the processor to run at 50 MHz, equating to a 20 ns clock period. Each of the ADD operations in the processor take four clock cycles, and therefore one shift process (a single change in the LED light location), which adds up to 0x20000, takes 524,000 clock cycles, or about 10 milliseconds. With the added overhead for branching and shifting, a slight increase is added to the time it takes to change from one LED to another.

5.2. Sum of Power of Odd Numbers Assembly Program



The program determines all odd numbers between 0 and an input N and then sums the product of all the odd numbers by itself depending upon the input power given. First it checks if the lower and the higher numbers in the range are odd or not – this is done by diving the number i.e., performing multiple subtraction and checking if the remainder is zero or not. If the number is not odd, then it is made odd by incrementing it by one. Further, multiplication of the number by itself is performed by doing repeated addition, the number of times the odd number is multiplied by itself is based on the input power given. The lower number is to be passed to x5, higher no is to be passed to x6 and the power/exponent is passed to x3. The result is stored in register x29 and copied to memory location 0x00100014. Test case covers instructions JAL, ECALL, LUI, SW, ADDI, ADD, BGE, BNE, SUB and BEQ. The entire assembly program has 43 instructions, and therefore takes 172 clock cycles to execute. The assembly file is given in the "Complex Test Cases" folder as "sum_of_powers.s" within the project directory.

6. Further Optimizations and Tradeoffs

6.1. Reducing signals

Some signals may be reduced in the processor to reduce the number of registers used. These signals may include the "op-type" signal from the control unit to the ALU, which is currently and enumerated type. Turning this enumerated type into a single 6-bit vector will allow the signal to specify all 37 instructions, as around 20 of these operations use the add operation. This change will have no trade-offs, as the signal already exists. On the contrary, this change may also increase the clock speed if implemented properly, as the holistic optimization process during synthesis may remove unneeded signals and further increase the maximum speed of the processor.

6.2. Use existing memory

Using memory blocks existing on the FPGA will reduce the number of registers used, as currently the instruction memory uses registers to provide a combination data output. To do this change, the instruction memory must be inferred as a ROM, which will also require some other signals such as a read enable. Once implemented, less registers will be used at the cost of a probable lower speed, as the memory will now need to be clocked in order to provide the next

output. If implemented properly, it should still be able provide the instruction data within the allotted time, allowing for four-cycle single instruction execution.

6.3. Consolidation of the Control Unit

As it stands now, the control unit has branches that check for each and every instruction. Some of these branches can be removed as they provide the same outputs for multiple instructions. Removing redundant branches will reduce the number of registers and LUTs used, as it may remove extra logic which has not already been optimized out by the synthesis process. This optimization should have no tradeoffs, as the control unit should have the same outputs and speed if implemented properly.

7. Video Link: https://youtu.be/TvEK0f5t2RQ