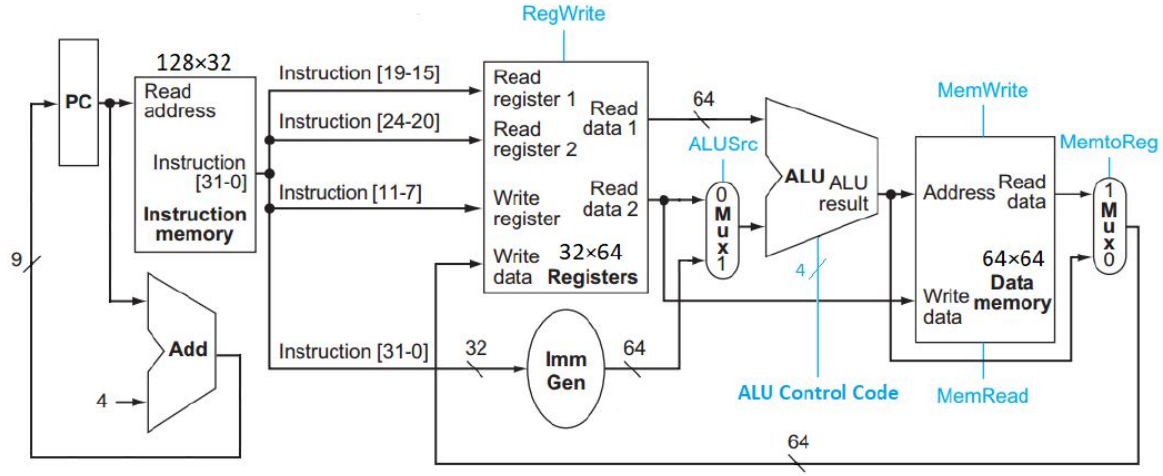
Final Report: RISC-V Design
EECS/CSE 31L
Prof. Pooria M. Yaghini
Due Date: 10/30/17
Group ID: Team Rocket

Jerry Lee
SID: 95436717
Keiser Ruiz
SID: 25807123
Jerry He
SID: 49794121

Objective

•Complete a design of the single cycle RISC-V processor using SystemVerilog
•Design and build modules for individual parts of Data Path
•Compile modules of Data Path, Controller, and ALU from Lab 7 and Lab 8 into one design
•Simulate and test using the Basys-2 FPGA Board and test benches

Data-Path of RISC-V Processor



Program Counter
The Program Counter (PC) is a 9-bit counter that outputs the addressing of the registers in Instruction Memory (IM). Each register has a length of 32-bits and there are 128 registers using the 9-bit counter. Because the Instruction Memory is byte addressable, the PC must be incremented by 4 each clock cycle.
In our implementation, the PC is implemented as a Flip-Flop with the output being the input of Instruction memory. The output is also in input to an adder which increments the address and outputs back to the input of the Flip-Flip.
Immediate Generator
Immediate Generator(ImmGen) receives the 32-bit instructions. Depending on the opcode, it will decide which bits are to be interpreted as the 12-bit immediate, then sign extending it to 64-bits.

Instruction Class	Opcode
R-type	0110011
I-type	0000011 0010011 1100111
S-type	0100011

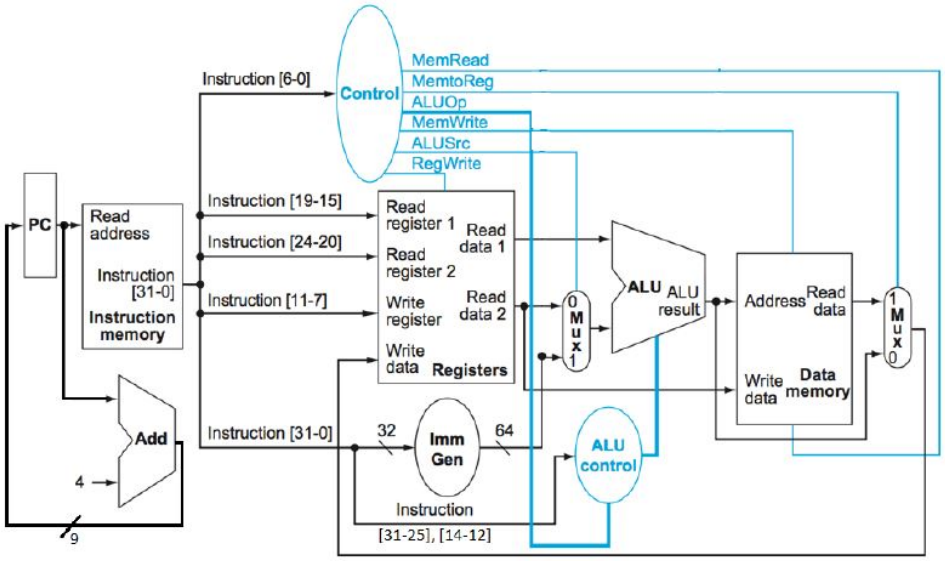
Instruction Memory
Instruction Memory holds the instructions and is byte addressable. In our implementation with 9-bits, it could cover up to $2^9/4 = 128$ registers. Because each instruction is length of 32-bits, the memory is 128x32.
Register File
The Register File reads two data address inputs and two data outputs. An additional 5-bit address input is the writing data address, as well as another input of length 64-bits for writing data.
ALU
The ALU is where arithmetic is done. It receives two 62-bit inputs and outputs one 64-bit output. The first Operand (Op1) is always taken by the ALU. Operand 2 (Op2) can come from either the Register File or the Immediate Generator. The ALU is controlled using a 4-bit Control Code received from the control unit that decides what operations are to be executed.

ALU Control Lines	Function
0000	AND
0001	OR
0010	ADD
0110	SUB

Data Memory
The Data Memory(DM) stores all data. In our implementation, Data Memory is byte addressable and is 512 bytes in this project. 9-bits are required to address the entire Data Memory, which are the 9 Least Significant Bits of the output from the ALU. The dimensions of the Data Memory are 64x64 due to being 512 bytes with 8 bytes per each register.
Mux
The first Mux at the output Operand 2 of the Register File determines if the second input of the ALU comes from the Immediate Generator or Operand 2 of Register File. The second Mux at the output of Data Memory decides if the data to be written back to the Register File should be from the ALU (operations like add, sub, etc.) or from the Data Memory (Load Operation).

Design and Implementation of Controller Unit

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



Controller

The Controller Unit drives the control signals throughout the architecture, determining the path of the operation. The 7-bit opcode from the instruction is the input of the Control Unit. The output of the unit consists of:

•Two separate 1-bit signals to the ALU Mux (ALUSrc) and Data Memory MUX (MemtoReg)

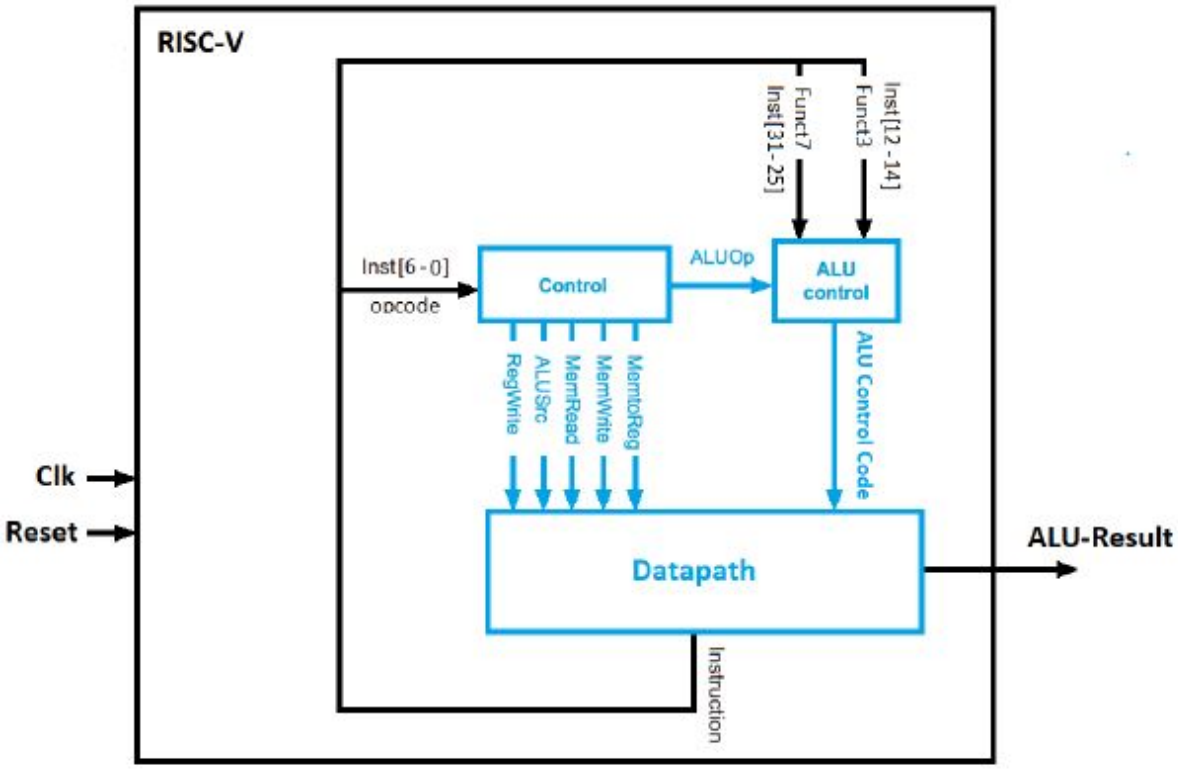
•Three signals to control the reads and writes in Register File (RegWrite) and Data Memory (MemRead & MemWrite)

•One 2-bit control signal for ALU (ALUOp)

Instruction	ALUSrc	Memto-Reg	RegWrite	Mem-Read	Mem-Write	ALUOp1	ALUOp0
R-format	0	0	1	0	0	1	0
ld	1	1	1	1	0	0	0
sd	1	X	0	0	1	0	0

RISCV Design

Once all Modules with the functionalities above are implemented in SystemVerilog, they are put together to form a complete single-cycle RISCV processor. Below is the diagram showing how the Data-Path, ALY control modules and the Controller are related to each other.



Error Analysis & Design Challenges

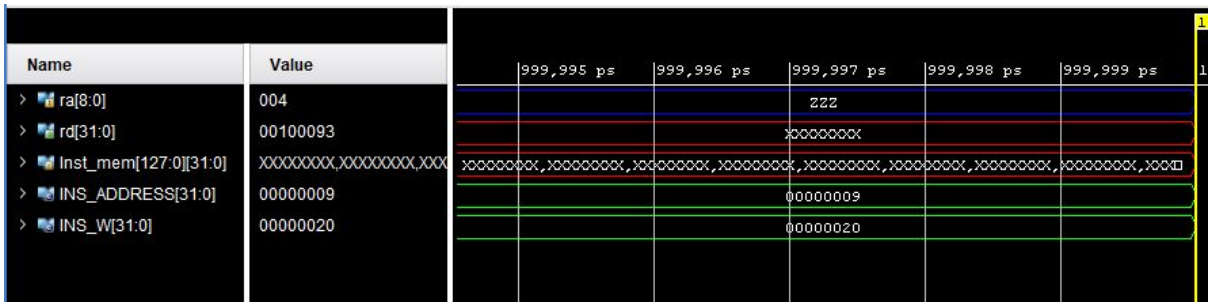
Initial errors were due to using "always begin" in our module implementations. Using "always" without a sensitivity list would render the code unusable due it running at all times. To fix this error, we rewrote the modules using "assign" instead of if statements inside "always". Our other solution was to include a sensitivity list. We found that both solutions were valid for our implementation.

We ran into many problems when putting all the modules into the datapath. Before implementing the datapath, we wrote test benches and tested each module individually. That eliminated problems within the modules themselves. From there, we would test the datapath in the RISCV module with the Controller and ALU control. Those modules were individually tested as well to eliminate errors within themselves. We could then run simulations, add different scopes to the waveform and trace see all input/output values throughout the processor. From there we traced the inputs/outputs to the sources of our errors. Many of our errors consisted of logic errors or mistyping inputs/outputs leading to many runtime errors.

One error we came across was in our load and store instructions. The store would have problems with storing the next instructions values at the end of the instruction cycle. To fix this, we added a 1ns delay before the Memory Write portion. That way, it would not see the next values

and write over itself. To do this, we also had to separate our read and write code so the delay would not also affect the read functionality.

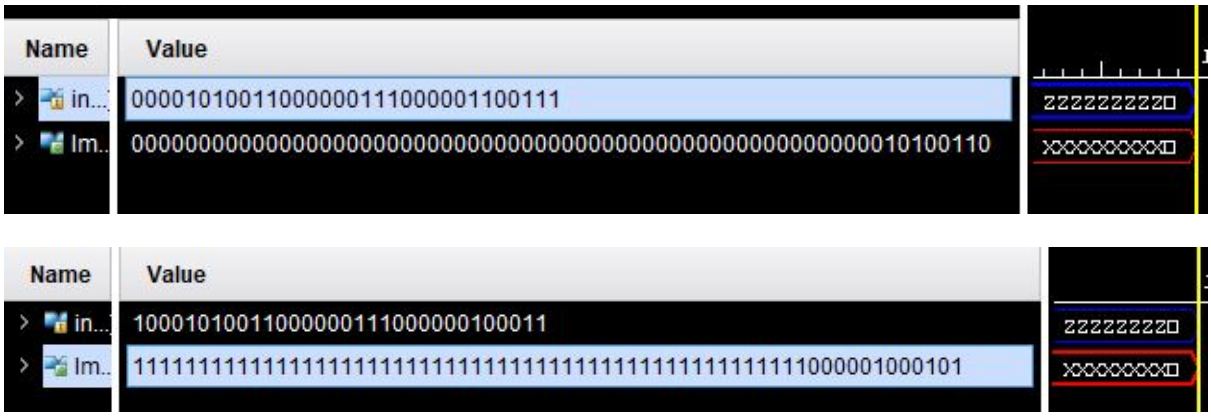
Testing



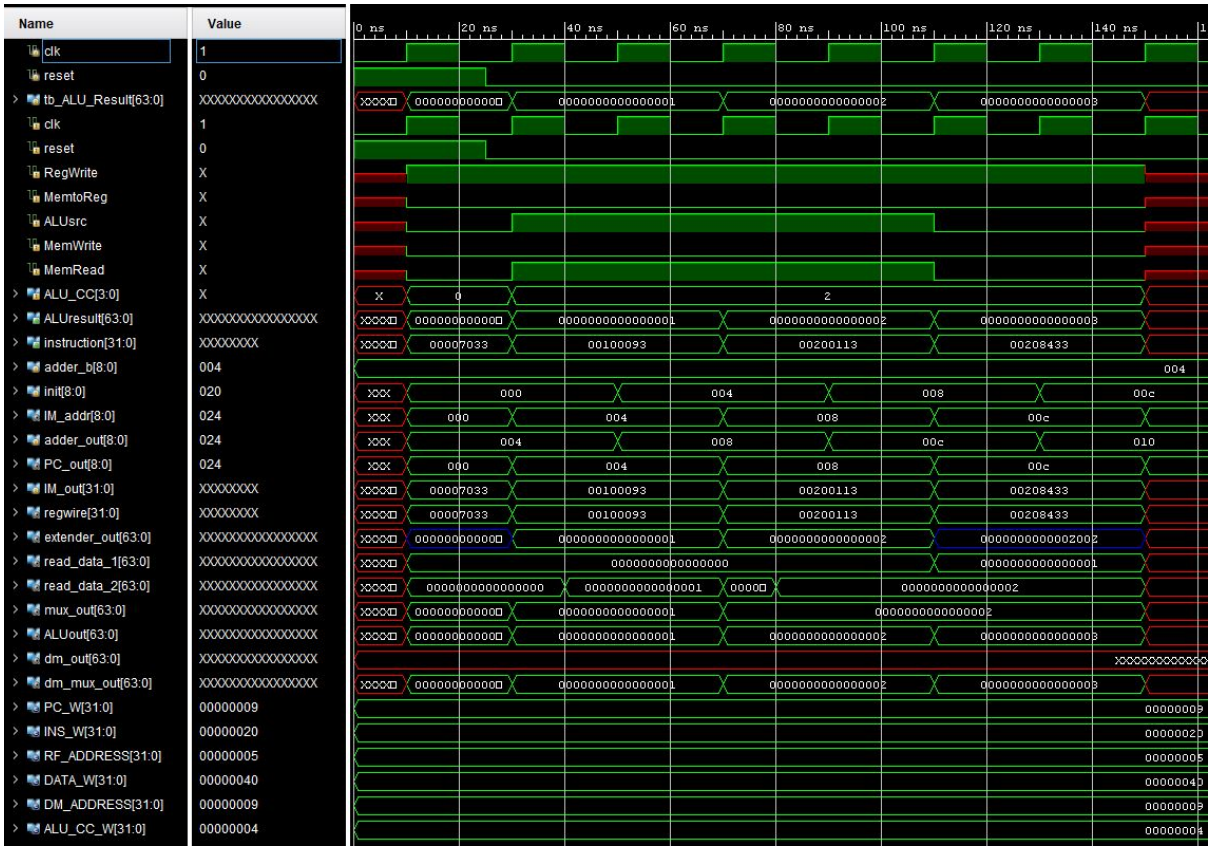
Run with an Instruction Memory Testbench:

To test instruction memory, we pre-loaded the instruction memory with instructions in several addresses and simulated inputs to see if the outputs match what we were looking to extract from the memory. The module worked as expected. We tested other modules similarly, running simulation and testing if the outputs are as expected with our chosen inputs. Below are two examples of test bench results for the Immediate Generator to test if its sign extension was operating correctly.

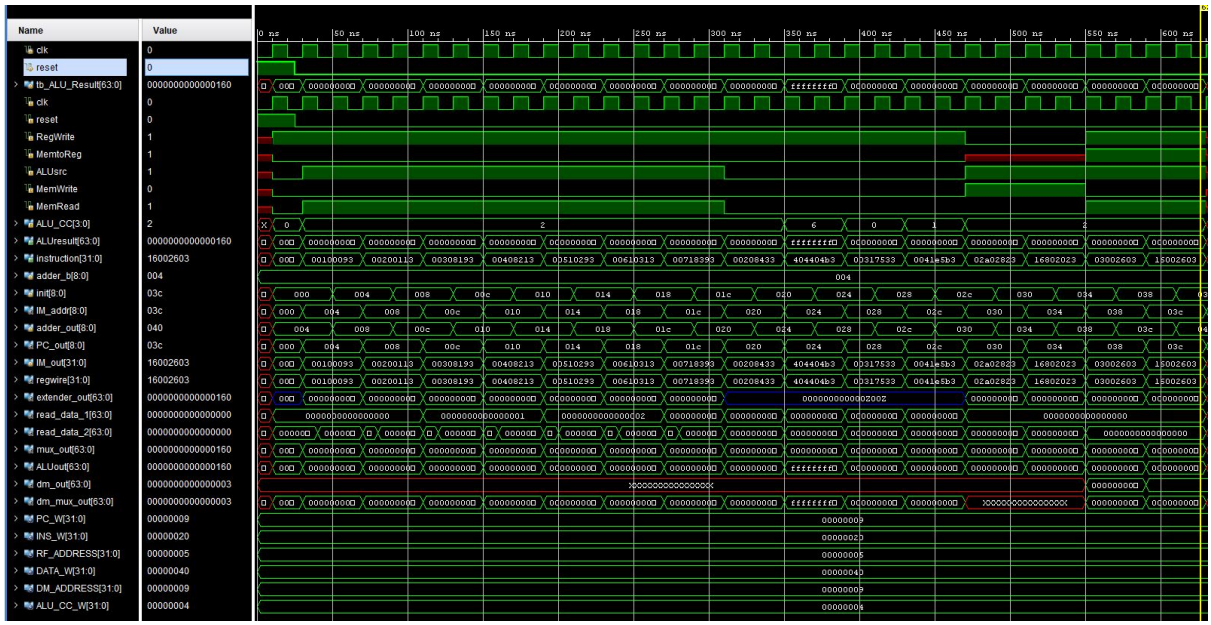
Immediate Generator (Sign Extension):



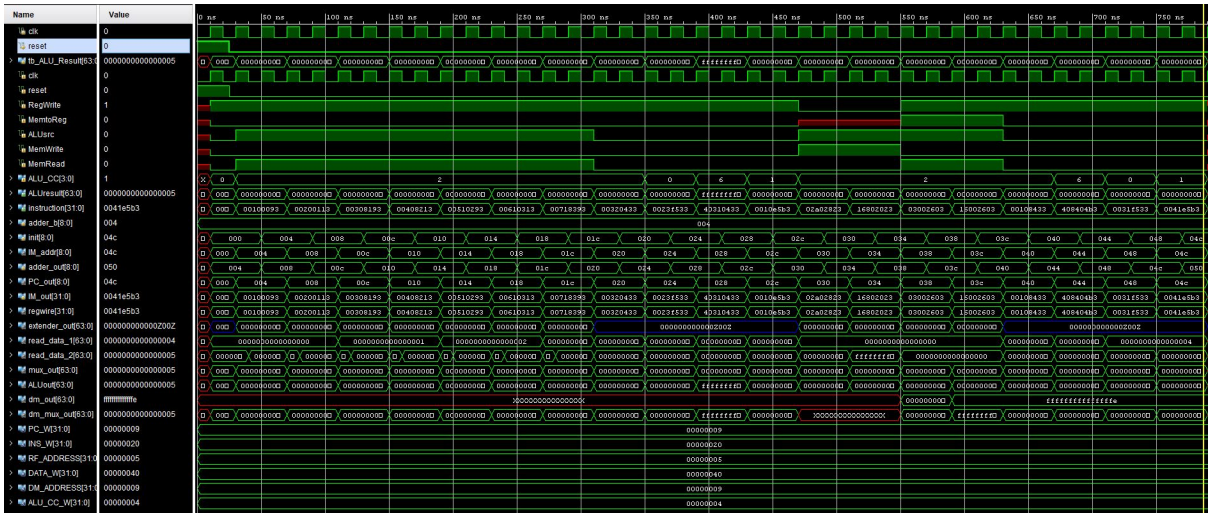
Waveform of the Lab 9 given instruction program run in test bench:



Waveform of the Github Instructions program run in test bench:



Waveform of the Self-Designed Instructions program run in test bench:



Time Analysis

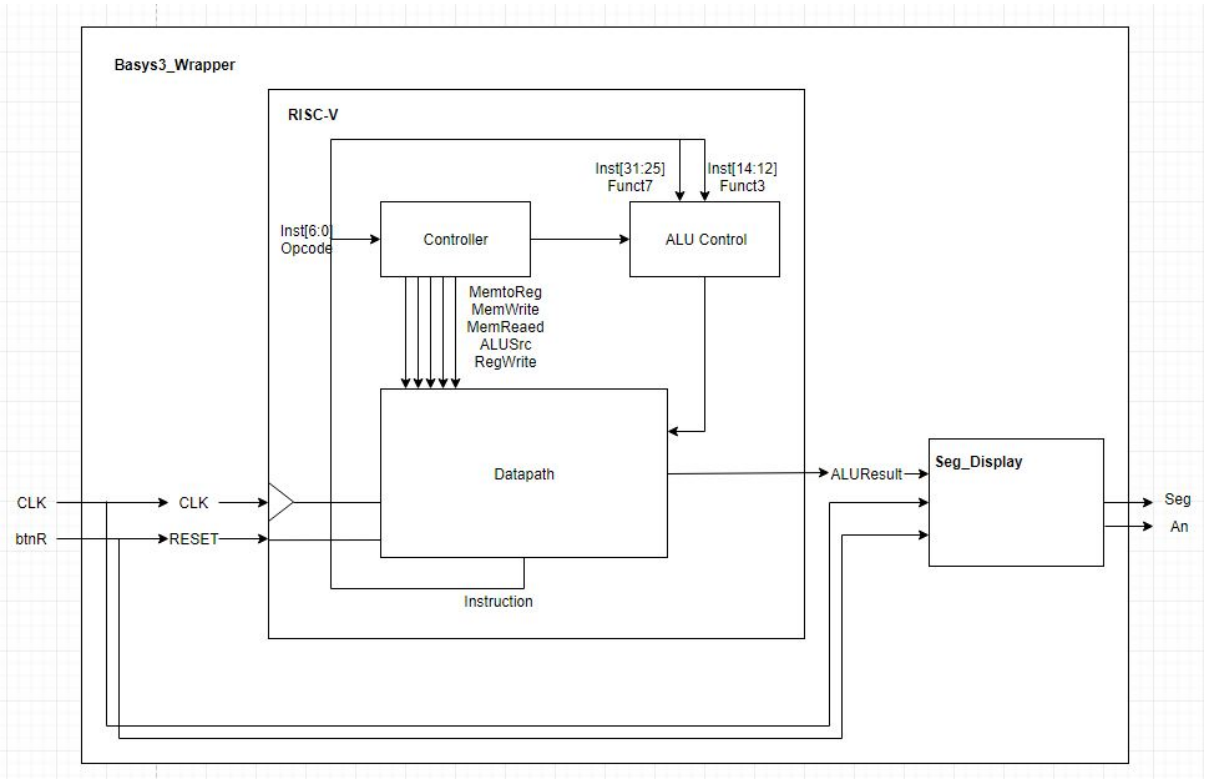
By default, our processor is Default to a clock speed 100.00 MHz when using a minimum of a 10ns period. Using a timing report, we were able to see that the Slack was 6.986ns, which is the required time subtracting the arrival time.
Therefore, the clock period could have by 6.986ns shorter and it should still run properly. The period requirement could be 10 - 6.986 = 3.014ns, which would come to a clock speed of about 331Mhz. Our maximal clock would be 331Mhz.
Timing reports:

```
------------------------------------------------------------------------------------------------
| Clock Summary
| -------------
------------------------------------------------------------------------------------------------

Clock        Waveform(ns)       Period(ns)      Frequency(MHz)
-----        ------------       ----------      --------------
sys_clk_pin  {0.000 5.000}      10.000          100.000


------------------------------------------------------------------------------------------------
| Intra Clock Table
| -----------------
------------------------------------------------------------------------------------------------

Clock             WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints      WHS(ns)
-----             -------      -------  ---------------------  -------------------      -------
sys_clk_pin         6.986        0.000                      0                   27        0.113
```

```
Max Delay Paths
--------------------------------------------------------------------------------
Slack (MET) :             6.986ns  (required time - arrival time)
  Source:                 clk_divide_reg[1]/C
                            (rising edge-triggered cell FDRE clocke
  Destination:            clk_divide_reg[25]/D
                            (rising edge-triggered cell FDRE clocke
  Path Group:             sys_clk_pin
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            10.000ns  (sys_clk_pin rise@10.000ns - sy
  Data Path Delay:        2.896ns  (logic 2.048ns (70.718%)  route
```

Overall Block Diagram

Overall Block Diagram:



Final Evaluation

In this final project, we implemented individual modules for a datapath throughout the second half of the quarter. We combined them all to implement a single cycle RISC-V processor and output our results on a FPGA board seven segment display. In this process, we learned about the system architecture, how a processor works on a low level, how instructions are executed in the architecture, and how to design and implement the system in SystemVerilog. We also

reinforced our knowledge in SystemVerilog programming. We were successful in getting the results expected of our processor.