DATE: 14 March 2021

Author: Justin Loi

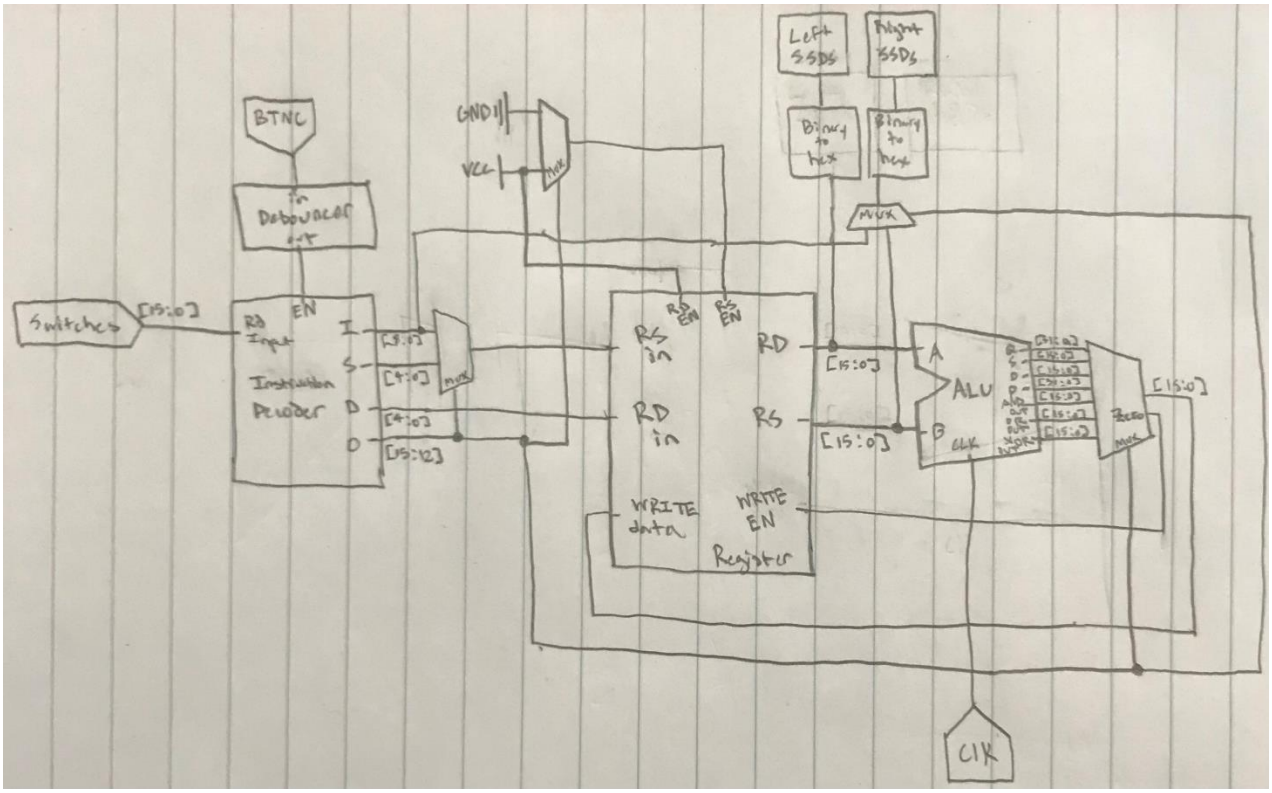
SUBJECT: FPGA CPU w/ custom ISA using Xilinx IP Report

Project Description:

This project involves the building of a simple, extremely streamlined CPU. It was designed in the interests of smooth implementation, rather than accuracy, so there are certain inherent design flaws in it, mostly related to high-speed timing.

Initial CPU Design:

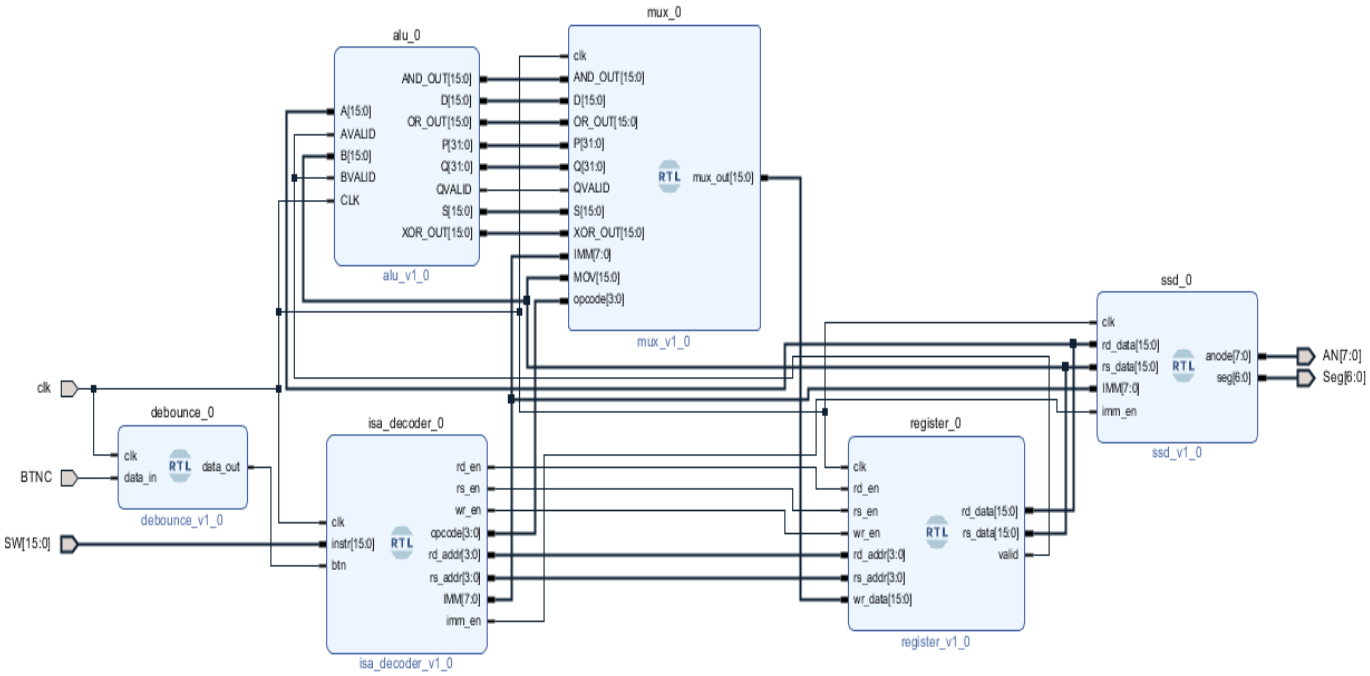
Starting from the left, I knew the switches would need to be decoded first, then read by the register module to retrieve the proper registers, then input the register values into the ALU and output the requested opcode using a multiplexer. Extra logic was needed to adapt to the LDI operation, as multiplexer were added for the register enable and right SSDs. For the select, the opcode from the decoder is used to change the values of the enables and which input to convert to hex for the right SSDs. Finally, a button and a de-bouncer is connected to the decoder so that the inputs from the switch are not entered until the 20-cycle delay is finished, needed for the ALU's division.



Initial CPU Block Diagram Design

Final CPU Design:

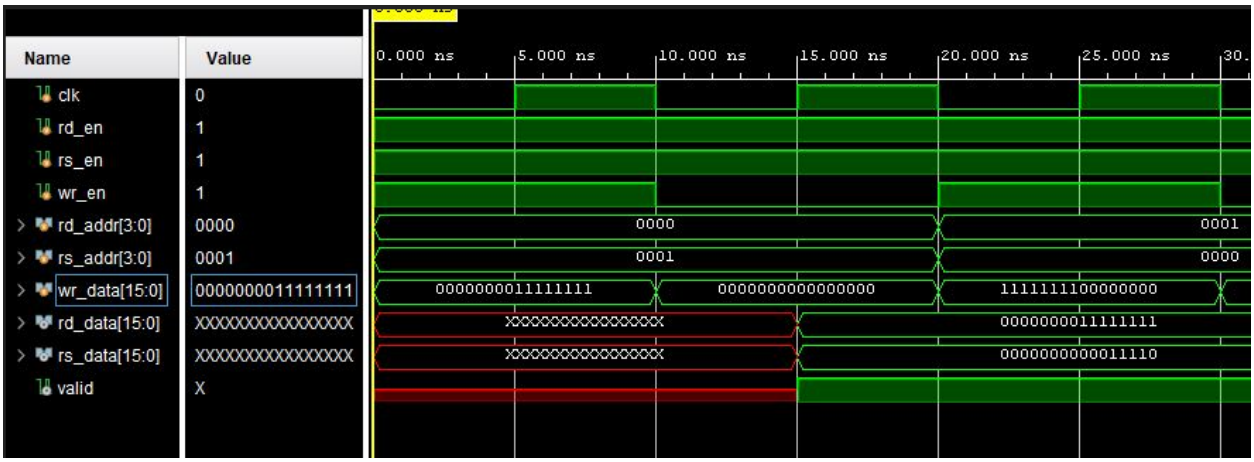
After spending time reviewing the initial design, I saw that multiple simplifications could be made. The first change was that the enables of the register could all be done within the decoder module itself instead of using external MUX modules. The second simplification was making an SSD module that would have the select input, convert the data into binary, and output the data into the SSDs on the FPGA all in one. The third and final change made was attaching the clock input into the SSD module so that it can cycle through all the different SSDs. Modules that needed to be implemented for phase 2 of the project include: SSD, MUX, Register, and ISA Decoder modules. The debounce module helps prevent timing issues in CPU. Other than those three changes, the rest of the diagram follows the draft from phase 1.



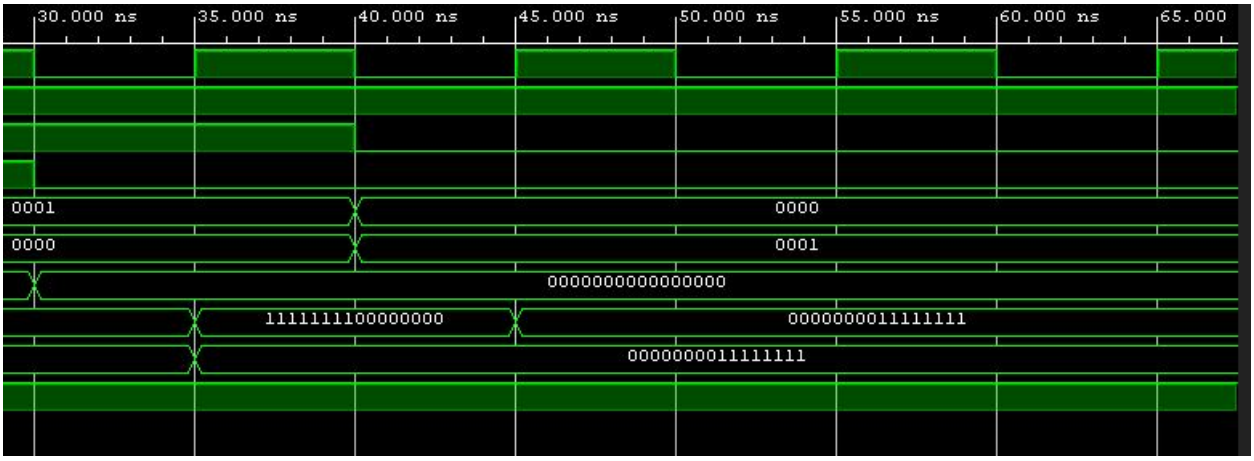
Final CPU Block Diagram Design

Register:

The register block is dual port memory design. For the testbench I tested to see if the data at certain addresses were changed properly when wr_en=1 and the data did not change when wr_en=0. I also tested to make sure that the data saved properly at the addresses written to.



Register Testbench Waveform (0-30ns)



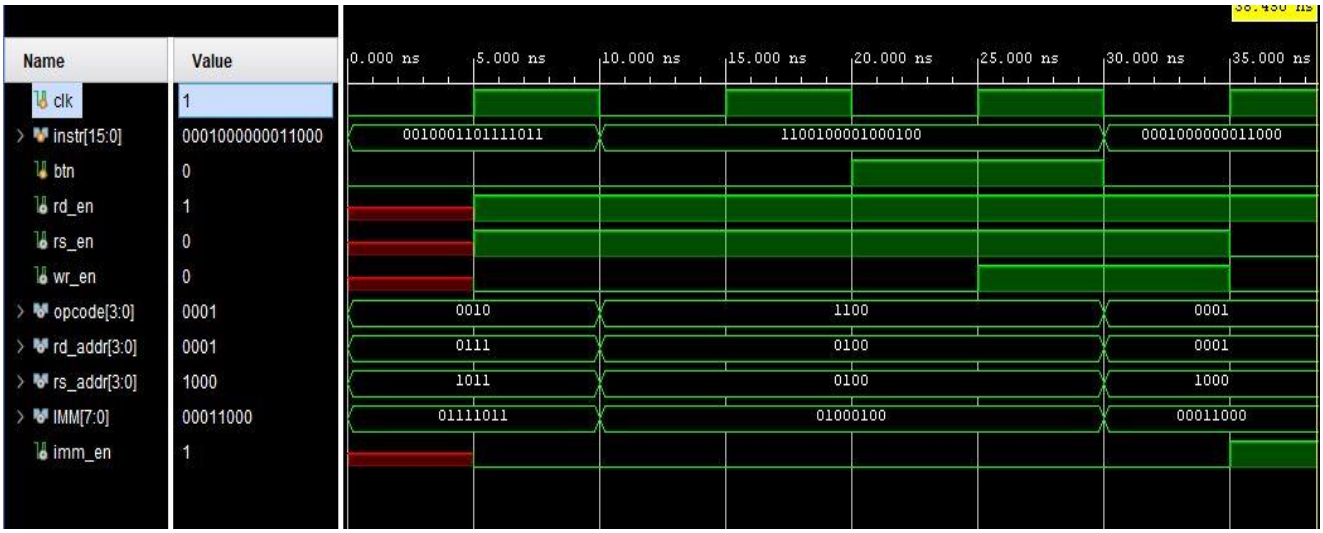
Register Testbench Waveform (30-65ns)

ISA Decoder:

Instruction	Format	Opcode(binary)	Description
MOV	OOOO----DDDDSSSS	0000	Moves Contents of RS into RD
LDI	OOOODDDDIIIIIIII	0001	Loads 8 bit # into RD
ADD	OOOO----DDDDSSSS	0010	RD = RD + RS
SUB	OOOO----DDDDSSSS	0011	RD = RD - RS
MUL	OOOO----DDDDSSSS	0100	RD = RD * RS
DIV	OOOO----DDDDSSSS	0101	RD = RD / RS
MOD	OOOO----DDDDSSSS	0110	RD = RD % RS
AND	OOOO----DDDDSSSS	0111	RD = RD & RS
OR	OOOO----DDDDSSSS	1000	RD = RD | RS
XOR	OOOO----DDDDSSSS	1001	RD = RD ^ RS

For the decoder I knew which bits of the instruction belonged to what output, be it the RS, IMM, and opcode. As a result, I used an assign statement for the rs_address, opcode, and IMM outputs since there location in the instruction set never changes. The only exception was RD as the location of the RD bits depended on the opcode whether it was a load opcode or something else. To solve this I used an ‘if’ block to check whether the ‘load’ opcode was true or not to decide which instruction bits to assign to the ‘RD’ address. After the ‘assign’ statements, an ‘always’ block was used to constantly loop to check the opcode and check if the button on the FPGA has been pushed, if pushed, wr_en=1 and vice versa.

For the testbench, I tested to see whether the instruction set was parsed correctly among the different address outputs. I then checked to see if the write and IMM enables worked correctly depending on whether the button was pushed or when the opcode=0001 (the load opcode).



ISA Decoder Testbench Waveform (0-40ns)

MUX:

The inputs of the multiplexer include all the outputs from the ALU, including the IMM and MOV data. Then I used the opcode from the decoder as the ‘select’ input, deciding which of the inputs to write to the register. For the implementation I used a ‘case’ block inside a ‘always’ block, where each case is a different opcode that determines what 16-bits the MUX will output.

SSD:

This module I had the hardest time implementing as I did not have any idea where to start. After consulting the TA in office hours and using online resources, I was able to understand how the seven segment displays worked and the process needed to light all eight displays. I first made a counter that would increment in binary and using a ‘case’ statement, select which display to light up depending on counter value. This count occurs every clock cycle so each of the displays are lighting up quickly enough to not be seen by the human eye. As the displays are being selected, each case assigns 4-bits from either the RD, RS, or IMM data to a BCD variable. The left 4 displays show the RD data in hex and the right 4 display the RS or IMM data that is selected by the decoder based on the instruction opcode (IMM data displayed for load instruction case). Then another ‘always’ block with a ‘case’ block is used to determine which segments to light up depending on the BCD value for the respective display.

ALU:

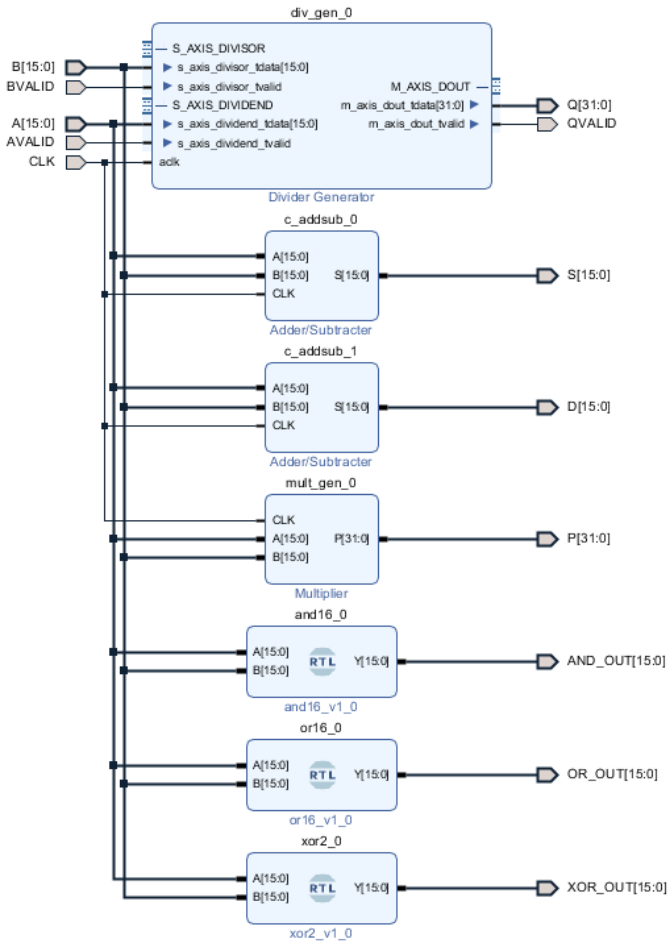
The ALU IP block is constructed exactly as the tutorial instructed without any problems. The purpose of the ALU is calculating different outputs for different arithmetic and logic functions, given two 16-bit inputs. The functions include division, adding, subtraction, multiplying, AND, OR, and XOR.

For the division block, the block will do "A/B", where the most significant 16-bits of the 'Q' output is the quotient, and the least significant 16-bits of the 'Q' output is the remainder.

For the addition and subtraction blocks, the sum and difference is calculated into a 16-bit output with equation A+/-B=output, plus for addition block and minus for subtraction block.

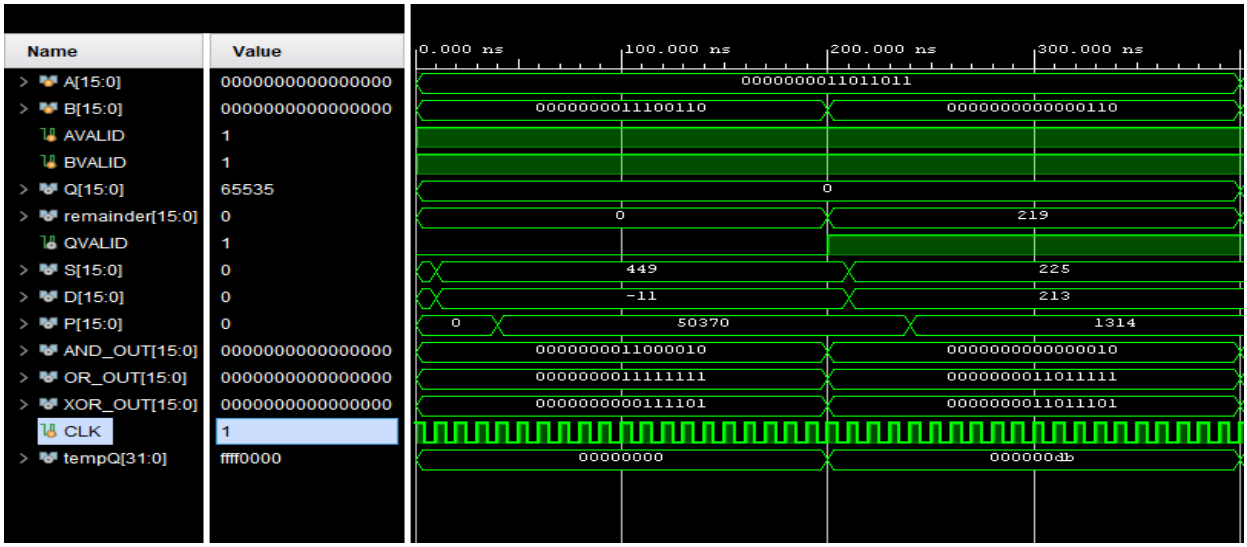
The multiplication block multiplies 'A' and 'B', so 32-bits are given for the output.

The AND, OR, and XOR performs their respective logic operations on both 'A' and 'B'.

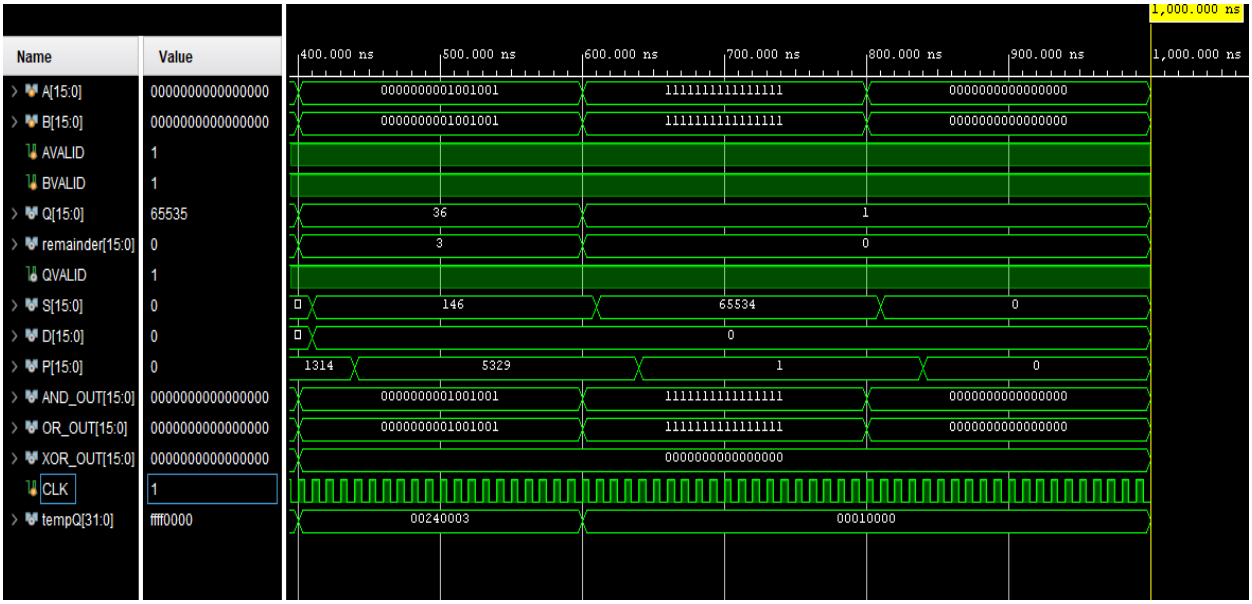


ALU Block Diagram

For the testbench, five different input combinations were used: A<B, A>B, A=B, A=B=0, and A=B=1. The clock used is a 10ns clock cycle with 200ns delay for each input change to account for calculation time in division block of ALU. The purpose of the testbench was to test all the arithmetic and logic functions worked normally in the constructed ALU. The arithmetic outputs are displayed in decimal and the logic outputs are displayed as binary for verification convenience. The output of the ALU testbench is displayed in the waveforms below.



ALU Testbench Waveform (0-400ns)



ALU Testbench Waveform (400-1000ns)