ECE 3544: Digital Design I
Project 5 – Datapath and Finite State Machine Design

Student Name: Richard Ugo

Honor Code Pledge: I have neither given nor received unauthorized assistance on this assignment.

I have neither given nor received unauthorized assistance on this assignment

---

Grading: The design project will be graded on a 100 point basis, as shown below:

Manner of Presentation (30 points)

_____ / 5 Completed cover sheet included with report

_____ / 15 Organization: Clear, concise presentation of content; Use of appropriate, well-organized sections

_____ / 10 Mechanics: Spelling and grammar

Technical Merit (70 points)

_____ / 5 General discussion: *Did you describe the objectives in your own words? Did you discuss your other conclusions and the lessons you learned from the assignment?*

_____ / 15 Design discussion: *Did you discuss the approach you took to designing and implementing the modules that make up your system, and how you synthesized the system from its components?*

_____ / 5 State diagrams: *Does your state diagram model a system that performs the required tasks? Connect this discussion to your design discussion.*

_____ / 5 Testing discussion: *What was your approach to formulating your test benches? How did you verify the correctness of the modules you designed?*

_____ / 10 Supporting figures: *Waveforms showing correct operation of the top-level module.*

_____ / 30 Validation of the final design on the DE1-SoC board

_____ **Project Grade**

Project 5 – Datapath and Finite State Machine Design

Richard Ngo

Project Objectives:

The overall objective of this project is to implement an ALU with a multi cycle input to work around the DE1-SoC's limited input. The ALU must also be efficient with it's time, and output it's results as quick as possible. To accomplish this, the following will be done:

- Combine different modules to implement ALU
- Implement a state machine to keep track of inputs and functionalities of ALU

Design and Processes

The ALU should be able to obtain an opcode from the input in order to choose its operation, and then two input variables afterwards (A and B) in order to compute the result of the operation. These values are inputted with each press and release of the enter button (KEY 0), and the result is displayed immediately after the last necessary input variable is entered. This process is also repeatable, and the result of the last operation will be displayed until it is replaced by the result of the current operation. Before designing this ALU, multiple factors must be considered.

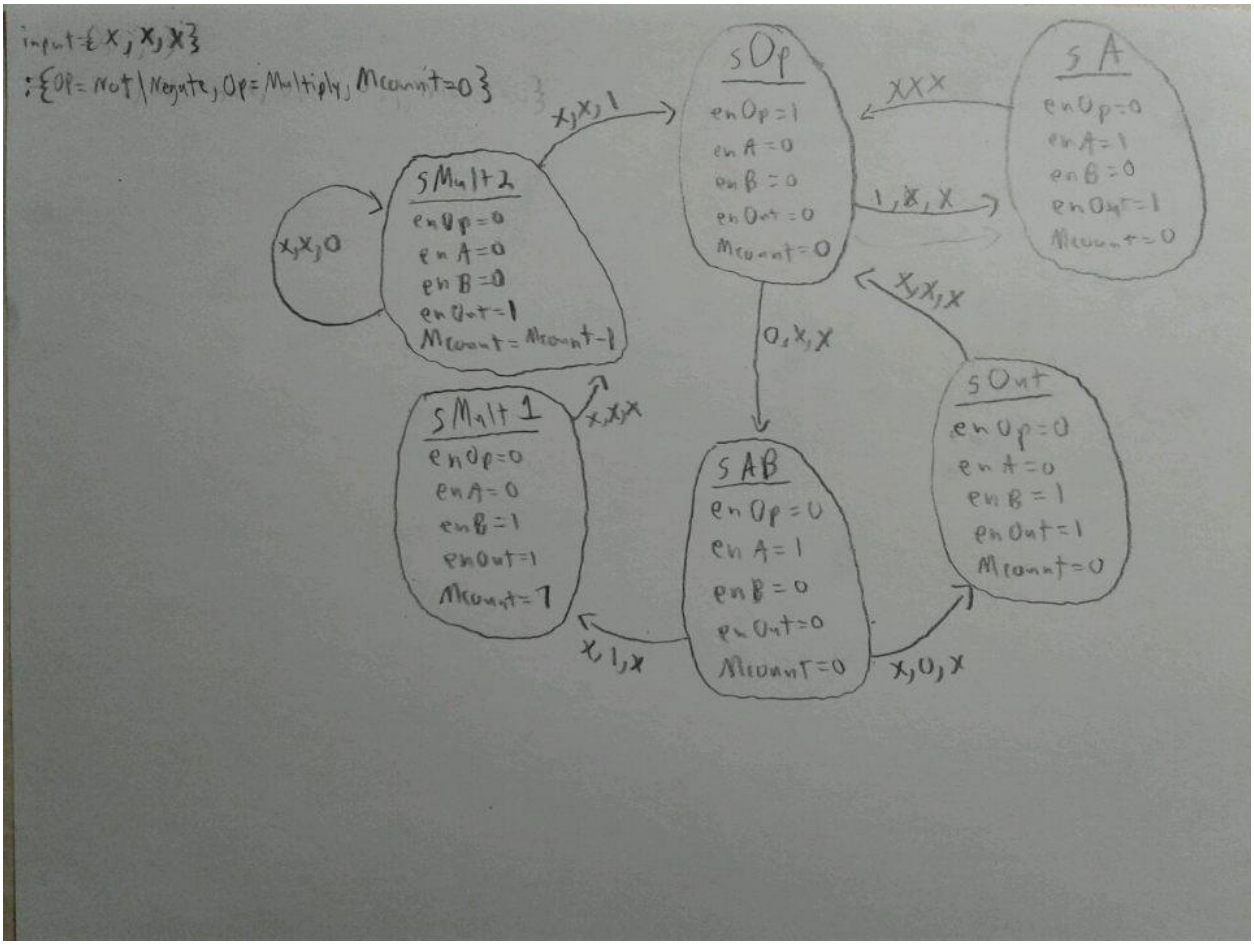
Registers

Due to the limited input of the board, the ALU must take its variables one at a time. This means that the ALU will not naturally have access to all the necessary variables for computation at all times, and will need to store its past inputs in order to do so. The result of the current operation must also be stored so that it can persist throughout the next operation. This is done by using registers, which saves the value at its input whenever activated. At least 3 variables must be stored, the opcode, input A, and the output of the last result. Because the opcode, A, and B are only relevant until the result is calculated, and B is generally the last input, B does not need to be saved since it is only needed at the moment it is entered. Thus, 3 registers are required for the implementation. To make sure that the inputs go to the right registers, an ALU controller also needs to be included.

ALU Controller

Generally, with each press and release of the enter button, the ALU will first read in an operation's opcode, and then the first operand A, and then the second operand B, though this process may shorten or extend depending on the current operation. The ALU controller should account for 3 different scenarios, when only input A is needed for operations like not and negate, when both inputs A and B are needed, and when a multiplication is taking place, which requires 8 extra presses until the result is complete.

State Diagram of ALU Controller



The en variables represent enable signals. Rather than directing the input signal to the correct register, the controller keeps track of the variables by disabling all irrelevant registers until it is a specific register's turn to save data.

From the state diagram, it is also noted that both the enables for the inputs and outputs are active at the same time in some of the states. This is to allow for the output to update at the instant the last input is updated, saving an extra button press.

ALU Core and Multiplier

The ALU should be able to carry out the following operations

Operation	**Result of the Operation**	**Operation code**
OR Your ID	Bitwise-OR the value {A, B} with the last four digits of your student number, expressed in BCD.	0000
NOT	Bitwise-complement the value of A. Zero-fill the 8-bit result to 16 bits.	0001
NAND	Bitwise-NAND the values of A and B. Zero-fill the 8-bit result to 16 bits. Zero-fill the 8-bit result to 16 bits.	0010
NOR	Bitwise-NOR the values of A and B. Zero-fill the 8-bit result to 16 bits.	0011
Add	Add A and B. Take A and B as 8-bit signed 2's complement operands, sign-extended to 16 bits. Your result should be a 16-bit signed 2's complement number.	0100
Subtract	Subtract B from A. Take A and B as 8-bit signed 2's complement operands, sign-extended to 16 bits. Your result should be a 16-bit signed 2's complement number.	0101
Negate	Take the 2's complement of A. Take A as an 8-bit signed 2's complement operand, sign-extended to 16 bits. Your result should be a 16-bit signed 2's complement number.	0110
Multiply	Multiply A and B. Take A as the multiplicand, and B as the multiplier. Take A and B as 8-bit unsigned numbers. Your result should be a 16-bit unsigned number. *Do not use the multiply dataflow operator to implement your multiplier*.	0111
Circular-Shift Left	Circular-shift A to the left by B positions. Take A as an 8-bit binary value, zero-filled to 16-bits. Take B as a 4-bit unsigned operand.	1000

Circular-Shift Right	Circular-shift A to the right by B positions. Take A as an 8-bit binary value, zero-filled to 16-bits. Take B as a 4-bit unsigned operand.	1001
Logical Shift Left	Logical-shift A to the left by B positions. Take A as an 8-bit binary value, zero filled to 16 bits. Take B as a 4-bit unsigned operand.	1010
Arithmetic Shift Right	Arithmetic-shift A to the right by B positions. Take A as an 8-bit signed 2's complement number, sign-extended to 16 bits. Take B as a 4-bit unsigned operand. *Do not use the Arithmetic Shift dataflow operators to implement your arithmetic shift operation.*	1011

To implement these instructions, a multiplier module is implemented and used to handle the multiply instruction, and an ALU Core is implemented and used to handle the rest.

Due to multiply being a multi-step operation, it is given its own dedicated module, and is progressed each time enter is pressed and released. A sequential multiplier was also used, since it is shift based, and can have its shift operations paced by the enter button.

The ALU Core was implemented using the dataflow operators available in Verilog.

added = opA+opB, subbed = opA-opB, twosC = ~opA+8'b00000001
product = output from multiplier

Operation	**Verilog Implementation**
OR Your ID (1296)	{opA,opB}|16'b0001001010010110
NOT	{8'b00000000,~opA}
NAND	{8'b00000000,~(opA&opB)}
NOR	{8'b00000000,~(opA|opB)}
Add	{{8{added[7]}},added}
Subtract	{{8{subbed[7]}},subbed}

Negate	{{8{twosC[7]}},twosC}
Multiply	product
Circular-Shift Left (combination of normal shift, and shift in reverse direction to replicate loop around)	({8'b00000000,opA} << opB[3:0])|({8'b00000000,opA} >> (16-opB[3:0]))
Circular-Shift Right	({8'b00000000,opA} >> opB[3:0])|({8'b00000000,opA} << (16-opB[3:0]))
Logical Shift Left	{8'b00000000,opA} << opB[3:0]
Arithmetic Shift Right	{{23{opA[7]}},opA} >> opB[3:0]

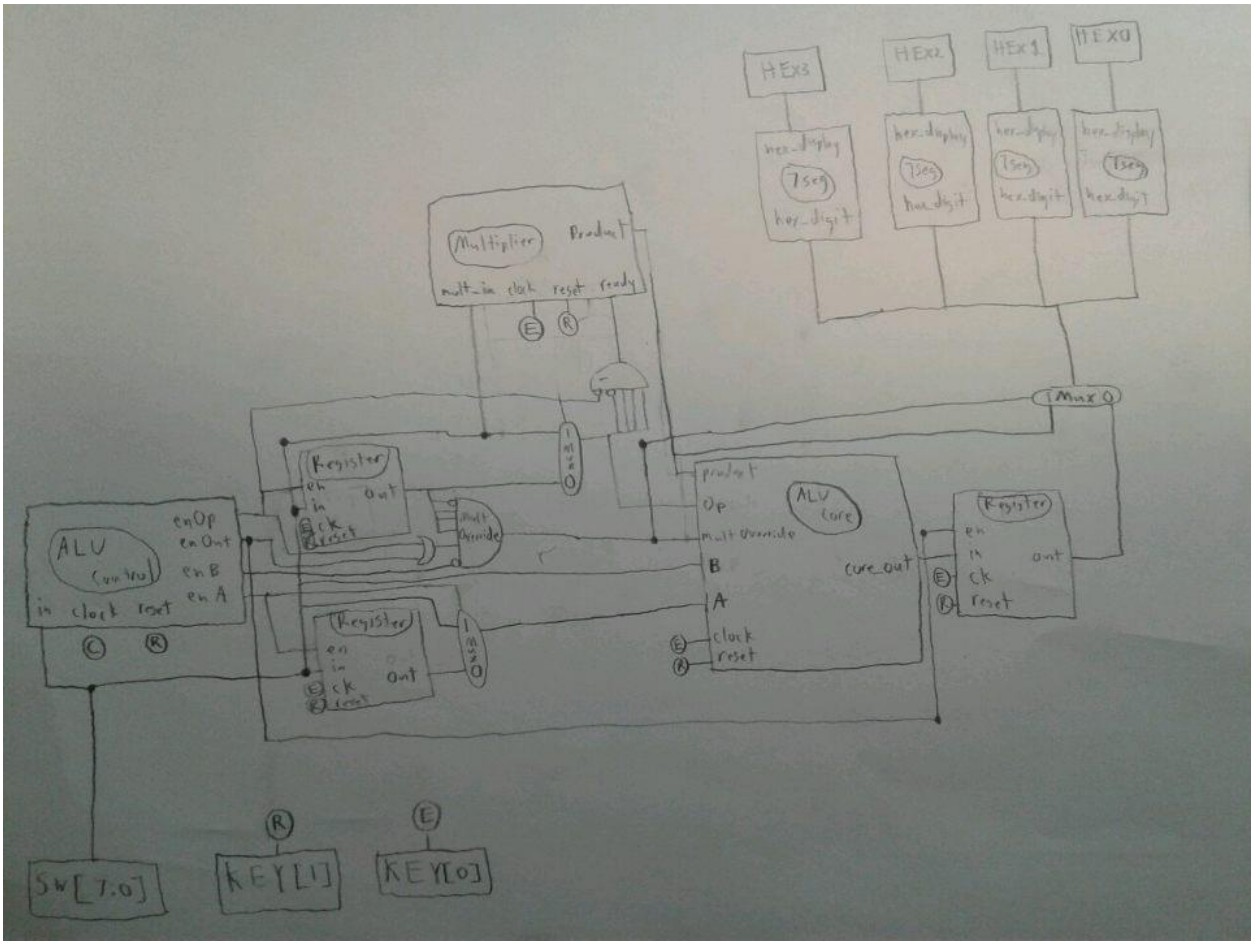
2x1 Multiplexers

Due to how registers work, a variable will not be saved until right after it is entered. Because the program should be capable of outputting the result at the moment the last variable is entered, the ALU won't always have enough time to wait for the registers to be ready and must take their values directly from the input switches. Also, because the ALU controller is triggered at the moment of an enter, and thus can't make decisions based on variables updated right after the button press, the controller won't know if a NOT or NEGATE operation is taking place the instant the operation is chosen, and can't tell the ALU that it needs to output at the very next trigger. To account for all these situations, multiplexers were added in order to give the ALU Core and Controller direct access to the variable entered in the input, until the variable is saved and ready to be used from the register.

Lastly, because the program needs to retain it's result until the next result is ready, the output must always be saved. Because unlike the ALU Core, the multiplier is triggered at the moment enter is pressed. This means the current output won't be ready until right after the button press, and the result saved to the register will always be one step behind. This can be dealt with by feeding the output of the multiplier directly to the output of the program during a multiplication using a multiplexer. However, multiple problems still come up at the very last cycle, where the opcode the next operation is inputted. Since the opcode will be overwritten, the ALU will no longer believe it is still doing a multiplication operation, and thus, it will stop outputting the multiplication before the output register can store the final product. Thus, the ALU must be delayed from changing its output for one extra state. This can be done by giving the ALU an input, that when high, will output the product regardless of what the current operation is. The

value will be set high if prior operation was a multiply operation, which can be known by checking the not yet updated opcode register, and if the multiplication operation is still ongoing, which can be obtained by seeing if the B variable has been inputted (~enB), and if the result is still being outputted (enOut).

A block diagram of the ALU with the above considerations is shown below



Simulation of ALU

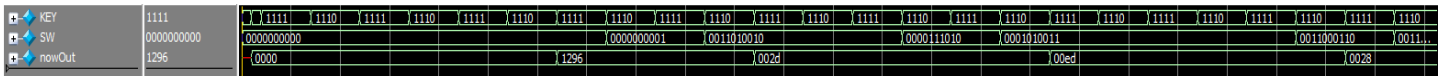
To test the ALU, each of the operations were simulated using ModelSim. The tests used in the validation sheet were used.

Opcode – SW[3:0]	**Operand A – SW[7:0]**	**Operand B – SW[7:0]**	**Result (HEX)**
0000	0000 0000	0000 0000	0001001010010110 1296
0001 (NOT)	1101 0010	NO VALUE	0000000000101101 002D
0010 (NAND)	0011 1010	0101 0011	0000000011101101 00ED
0011 (NOR)	0101 0011	1100 0110	0000000000101000 0028
0100 (Add)	1100 0101	0011 0101	1111111111111010 FFFA
0101 (Subtract)	0010 1010	0111 1110	1111111110101100 FFAC
0110 (Negate)	0000 1111	NO VALUE	1111111111110001 FFF1
1000 (CSL)	1000 0000	0000 1001	0000000000000001 0001
1001 (CSR)	0000 1111	0000 0100	1111000000000000 F000
1010 (LSL)	1000 1111	0000 0101	0001000111100000 11E0
1011 (ASR)	1100 1100	0000 0011	1111111111111001 FFF9
0011 (Multiply)	0011 1011	1010 0101	0000000010100101 0001110111010010 0000111011101001 0010010011110100 0001001001111010 0000100100111101 0010001000011110 0001000100001111 0010011000000111 00A5 1DD2

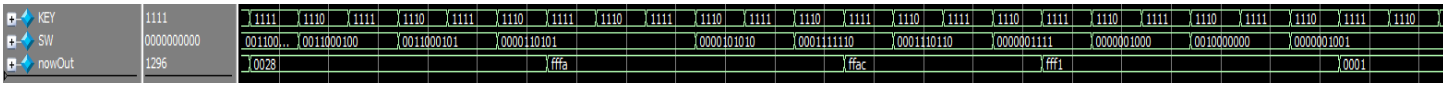
			0EE9 24F4 127A 093D 221E 110F 2607

Because the any wire can be accessed in a ModelSim simulation, debugging using the unused LEDs and switches were unnecessary. The wires leading into the seven segment decoders were used to examine the result value of the ALU, and when necessary, the en wires and multiplier output were used to observe the current state of the ALU and multiplier.

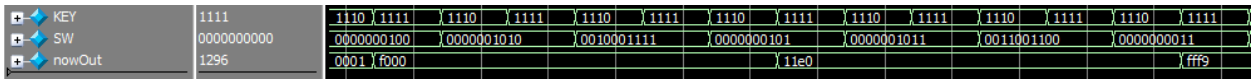
Testing: Reset, OR Your ID (1296), NOT, NAND, NOR



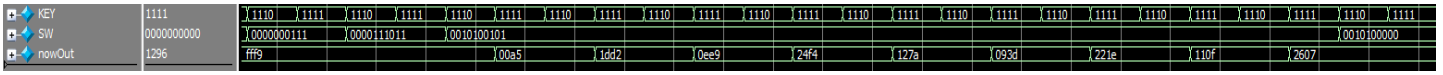
Testing: Add, Subtract, Negate, CSL



Testing: CSR, LSL, ASR

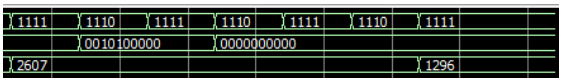


Testing: Multiply



To check that the Multiply operation also ended within the correct amount of cycles, an OR Your ID operation was also added to the end.

Testing: Multiply takes up correct number of cycles



Given that the expected results match the simulated results and that the waveform behaves as expected, it can be assumed that the ALU works correctly.

Physical FPGA Testing

The circuit was loaded onto the FPGA successfully. Because the HEX displays visually shows the output of the ALU in hexadecimal, the operations can be checked without needing to access any internal values, and the extra LEDs and switches were also unnecessary for debugging. Pressing KEY 1 restarted the ALU to the initial state and pressing and releasing KEY 0 correctly progressed the ALU. Reusing the inputs that were used in the simulations also yielded the same results. From these tests, I can assume that the circuit was implemented onto the board correctly.

Conclusion

This project was successfully completed with no problems. Through this project, I became more familiar with how ALUs work and how to work with module procedures that persist across multiple cycles. I also learned how to work around the limitations of registers when a response is needed within the same cycle and learned when it is appropriate to use the live input and when it is appropriate to use what is stored. Overall, this project has made me more comfortable all the digital design fundamentals learned so far in this course.