ECE 385

Fall 2023

Lab 2: 4 bit logic processor

Alex Yu, Corey Yu
GL/ 13:45-14:00
Gene Lee

Introduction

The circuit is a 4-bit processor that stores two values A, B in two registers and can perform bitwise operations (and, or, xor, etc.) and route the result to one of the two registers. The circuit consists of 4 sub-circuits, a register unit, computation unit, routing unit and control unit. The register unit consists of two shift registers. This is wired to the computational unit, which is capable of performing 4 bitwise functions on the two inputs: and, or, xor, set low, nand, nor, xnor, set high), The computational unit's output is fed to a routing unit, which consists of a dual 4-1 MUX that can discard the output, overwrite the output into A or B, and swap A and B. The entire circuit is controlled by the control unit; a two state Mealy Finite State Machine.

To load data into a register, set the switches D3 through D0 to the desired input, e.g. on / off / off / on for 1001, then toggle the switch for load A or load B respectively. To initiate a computation and routing operation, set the switches F2 through F0 to select the operation you wish to perform, and R1 through R0 to select which register to send the output to. Then toggle the execute switch.

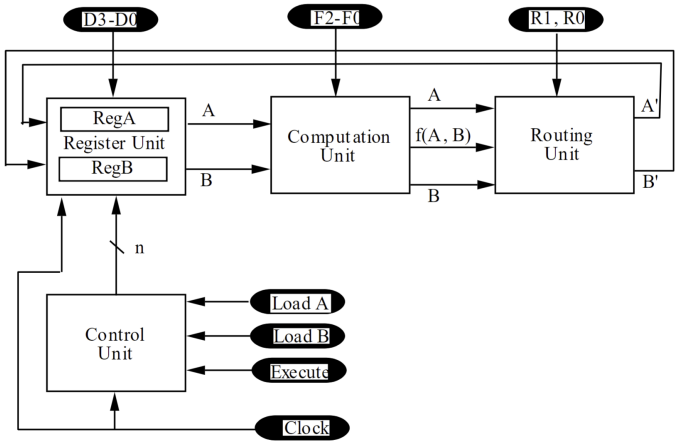

Figure 1: Block diagram of circuit

The control unit is an FSM and serves as the brain of the entire circuit. The control unit utilized a two state Mealy finite state machine, Q=0 for a reset state and Q=1 that combines the 4 shift states and a hold state. FSM and inputs consist of C1C0 (the bits of the counter), E (execute). S, the shift signal is the only output of the FSM. The bits on each transition arc are C1C0 / E / S.

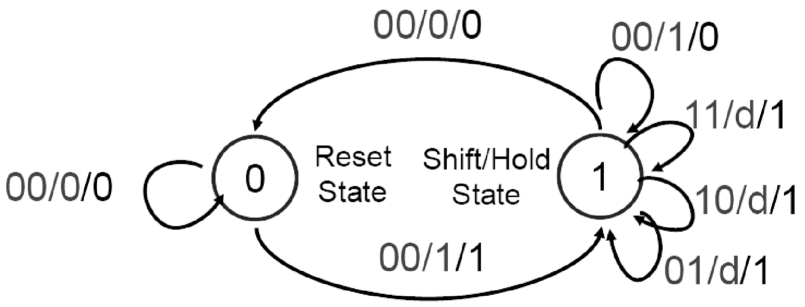

Figure 2: 2 state Mealy FSM final implementation

A Moore state machine, a FSM that only depends on the current state, was first designed to fit the constraints. To collapse the moore state machine, a 4 bit counter was used to reduce shift 0 to shift 3. The design was then transformed into a Mealy machine, where a new state bit Q was created, and S moved to an output. The shift states were collapsed with the halt state to Q=1.

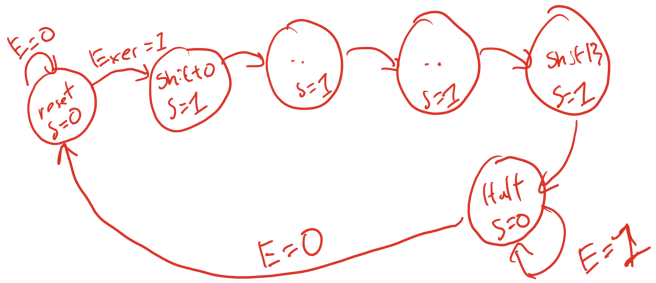

Figure 3: Initial Moore FSM design

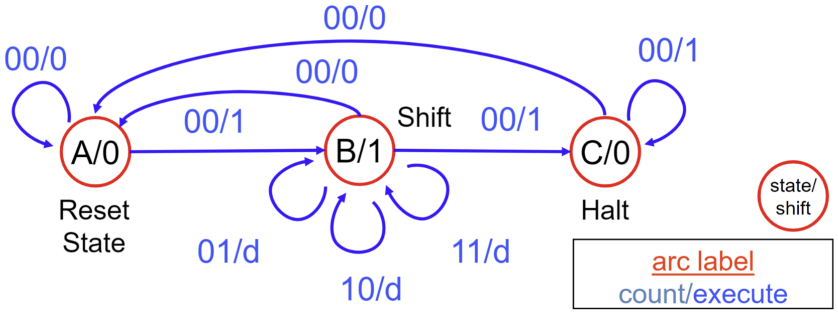

Figure 4: Collapsed (with counter) Moore FSM

To design the control unit: signals for the control unit were defined–load A, load B and execute. Load A/B is toggled on to load a value into the respective register, and execute is switched high for at least one clock cycle to perform a computation. The state machine is defined such that turning execute on for one cycle will execute one computation (or 4 logic operations/shifts) regardless of if it is turned off and on again while the operation is being

performed. Using the provided next-state table, K-maps were drawn for S and Q. C1 and C0 were not k-mapped as they would just be implemented on a counter chip.

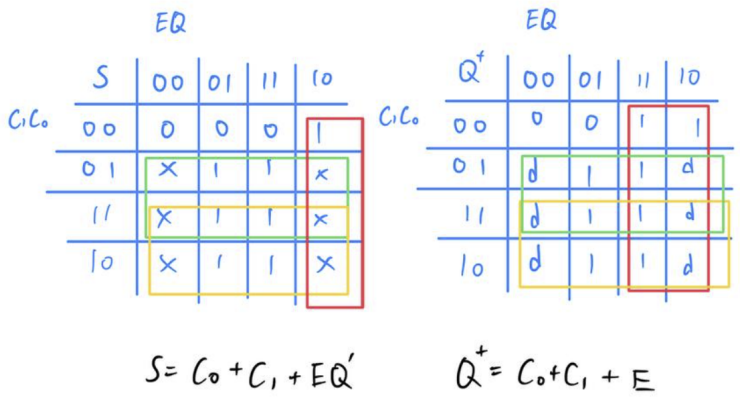

Figure 5: K-maps of control unit outputs

The computational unit has 8 possible outputs and only one is output at a time. The obvious choice for this is an 8-1 MUX with lines 0-7 the desired output for a given F2, F1, F0. Rather than using De Morgan's laws (A + B = !(AB)) to convert AND-OR logic into the desired outputs, a hex inverter was used. The lab kit has NAND, NOR, and XOR chips in addition to hex inverters. Thus when an OR gate was needed, the signals were fed into a NOR gate and inverted. By prioritizing logical simplicity over chip efficiency, debugging was made simpler.

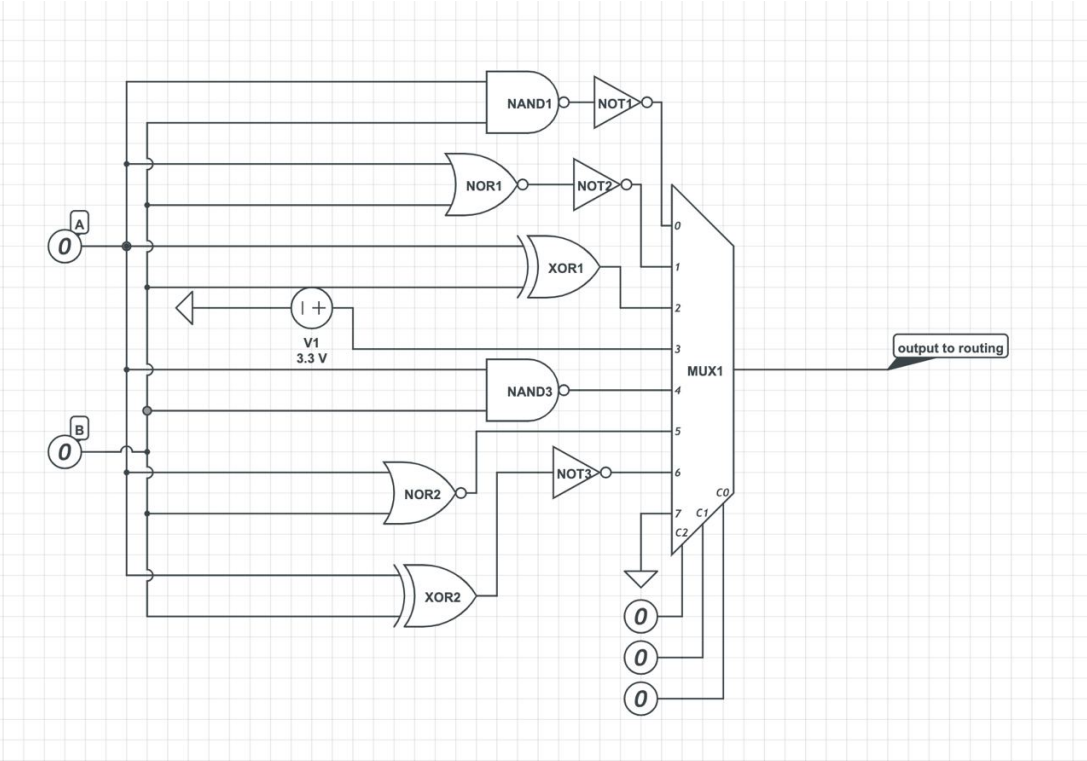

Figure 6: Logic diagram of computational unit

The routing unit has 4 possible outputs, and 2 outputs at a time. A 4-1 MUX satisfies this requirement. For the A-output MUX, wire A to line 0 and 1, F to line 2 and B to line 3. For the B-output MUX wire B to line 0, F to line 1, B to line 2 and A to line 4.

Function Selection Inputs			Computation Unit Output
F2	F1	F0	f(A, B)
0	0	0	A AND B
0	0	1	A OR B
0	1	0	A XOR B
0	1	1	1111
1	0	0	A NAND B
1	0	1	A NOR B
1	1	0	A XNOR B
1	1	1	0000

Routing Selection		Router Output	
R1	R0	A*	B*
0	0	A	B
0	1	A	F
1	0	F	B
1	1	B	A

Figure 7: Desired outputs of the computation and routing units

The register unit consists of two 4 bit registers to store A, B. D3-D0 are wired to D0-D3 (which order) as parallel load signals. Each register has two signals S1, S0 to control the operation of the register. As such, additional combinational logic is needed to generate S1 S0. By inspection, S0A = Load_A || S , S1A = Load_A and S0B = Load_B || S, S1B = Load_B. The output of the routing unit is wired to DSL, and Q0 is wired as an output of the register unit into the computational unit.

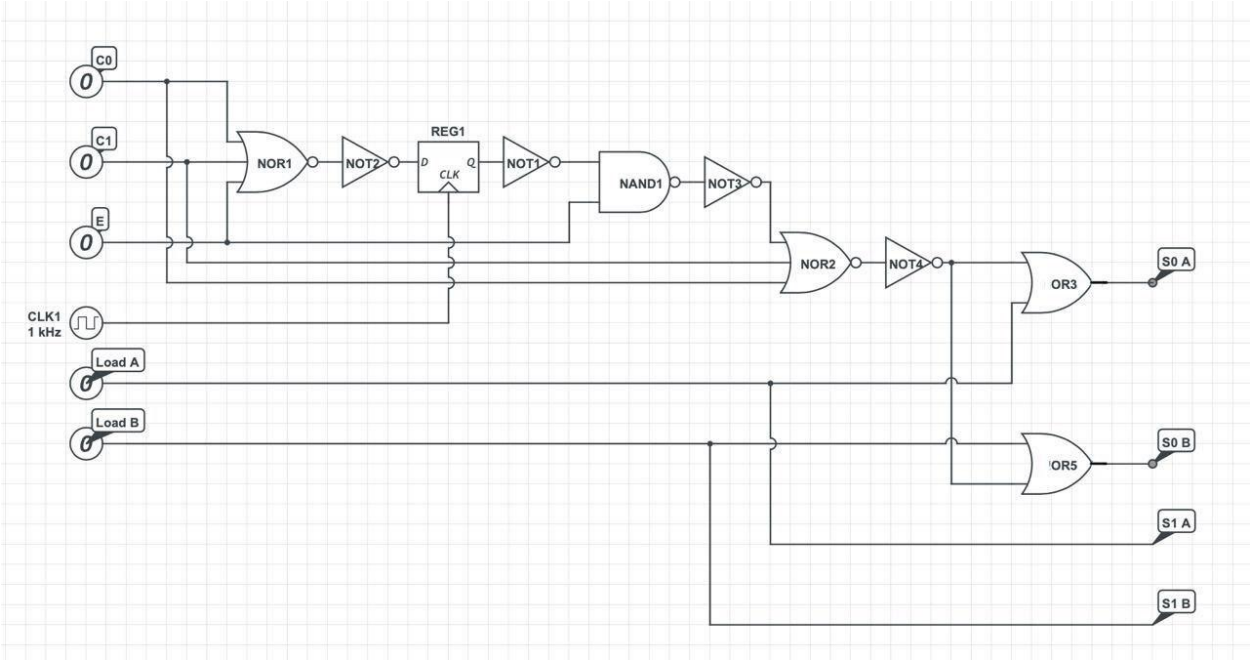

Figure 8: Control unit with additional logic for parallel loading of register A and B

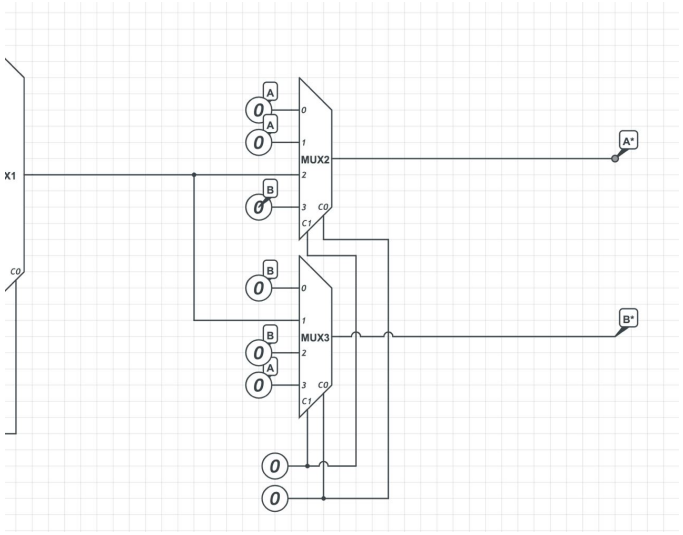

Figure 9: Routing unit comprised of a dual 4:1 MUX

PIN DIAGRAM

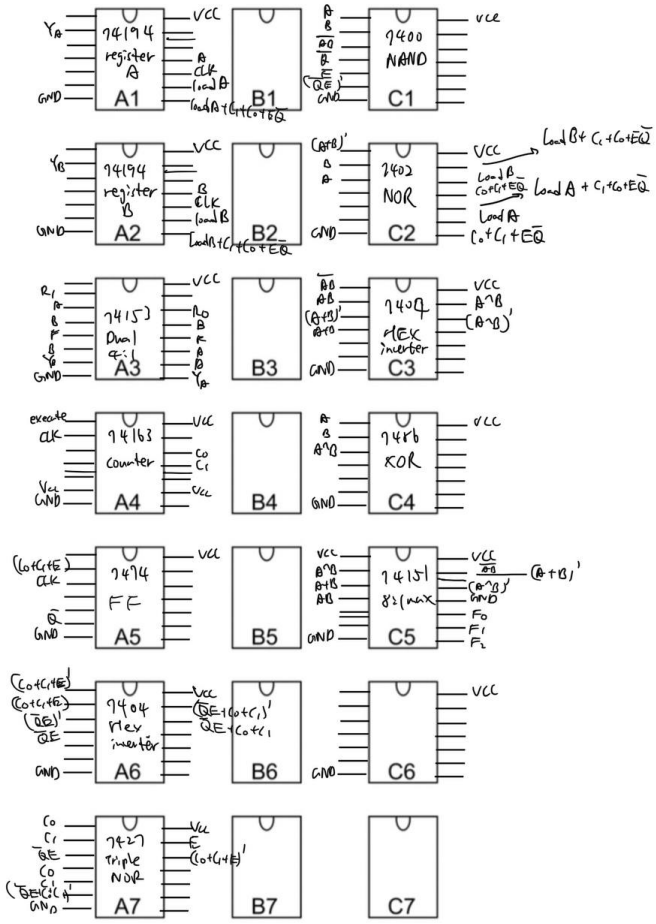

Figure 10: Pin diagram of the entire 4-bit processor circuit

After constructing a 4 bit processor on the breadboard, System Verilog code was provided for a 4 bit processor. This was then extended to an 8-bit processor with the same functionalities. Each SV (System Verilog) module description and purpose is detailed below.

Module: Processor.sv
Inputs: Clk, Reset, LoadA, LoadB, Execute, [7:0] Din, [2:0] F, [1:0] R
Outputs: [3:0] LED, [7:0] Aval, [7:0] Bval, [7:0] hex_seg, [3:0] hex_grid
Description: This is the top level module that calls the sub modules.
Purpose: This module organizes data between all the sub modules and wraps everything. It calls the units e.g. register unit, computation unit, and is responsible for the total data input and output.

Module: Synchronizer.sv
Inputs: Clk, Reset, d
Outputs: q
Description: This acts like a flip flop that takes an asynchronous signal and makes it synchronous. The asynch input is fed into the flip flop and outputs the data on the next clock cycle.
Purpose: This module forces the entire design to be synchronous, or that every module runs on the same clock cycle.

Module: Register_unit.sv
Inputs: Clk, Reset, A_In, B_in, Ld_A, Ld_B, Shift_En, [7:0] D
Outputs: A_out, B_out, [7:0] A, [7:0] B
Description: The register unit calls the 4 big registers A and B.
Purpose: This module loads the select data into the correct register (A or B), and inputs the shift signal into the register_4 module.

Module: Register_4.sv
Inputs: Clk, Reset, Shift_In, Load, Shift_En, [7:0] D
Outputs: [7:0] Data_Out, Shift_Out
Description: This is a positive edge triggered 8 bit register with synchronous reset and parallel load. When Load is high, data from the D register is loaded into Data_Out. When Shift_En is high, Data_Out <= {Shift_In, Data_Out [7:1]} and Shift_Out takes the value of Data_Out[0].
Purpose: The register_4 unit stores the data. It is a sub module of the Register unit, A and B are both a register_4 that will be called in register_unit.sv.

Module: compute.sv
Inputs: [2:0] F, A_In, B_In
Outputs: A_Out, B_Out, F_A_B
Description: This is an 8:1 MUX that takes A_In, B_In then sets F_A_B to the desired output based on [2:0]F and A_Out to A_In and B_Out to B_In
Purpose: The computation module takes the right most bit from the register and generates the output according to the select signal of the 8-1 mux. The resulting output is fed to the router unit.

Module: Router.sv
Inputs: [1:0]R, A_In, B_In, F_A_B
Outputs: A_Out, B_Out
Description: This is a dual 4:1 MUX that assigns takes A_In, B_In, F_A_B and assigns the desired values to A_Out and B_Out based on [1:0]R
Purpose: The router module sends the A and B output, high, low, or original input to the desired register.

Module: Control.sv
Inputs: Clk, Reset, LoadA, Load B, Execute
Outputs: Shift_En, Ld_A, Ld_B
Description: This is a Mealy FSM that has 8 count states. Using this design, we don't need a counter to count which state we are in. The flip flop will tell the control unit when to go to the next state. However, the reason why this is a mealy machine is that the last state still depends on the input execute.
Purpose: We need this unit to tell us when we shift the bits, and output the shifting signal into the register to control the shift.

Module: HexDriver.sv
Inputs: Clk, reset, [3:0] in[4], [3:0] nibble
Outputs: [7:0] hex_seg, [3:0] hex_grid, [7:0] hex
Description: The hex driver takes in the data in the binary format and outputs the number to hex that can be shown on the LED segment display.
Purpose: The LED segment display doesn't take in numbers in binary so we need this module to translate it to display.

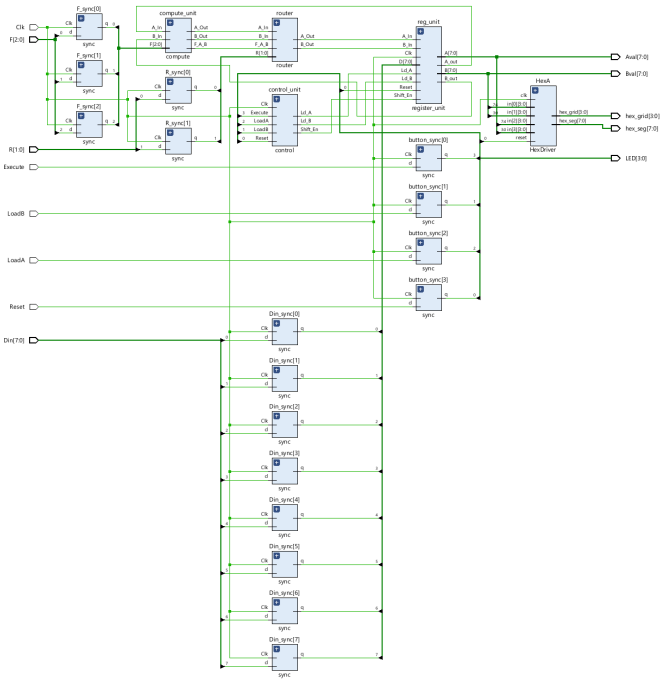

Figure 11: RTL Diagram of 8-bit processor

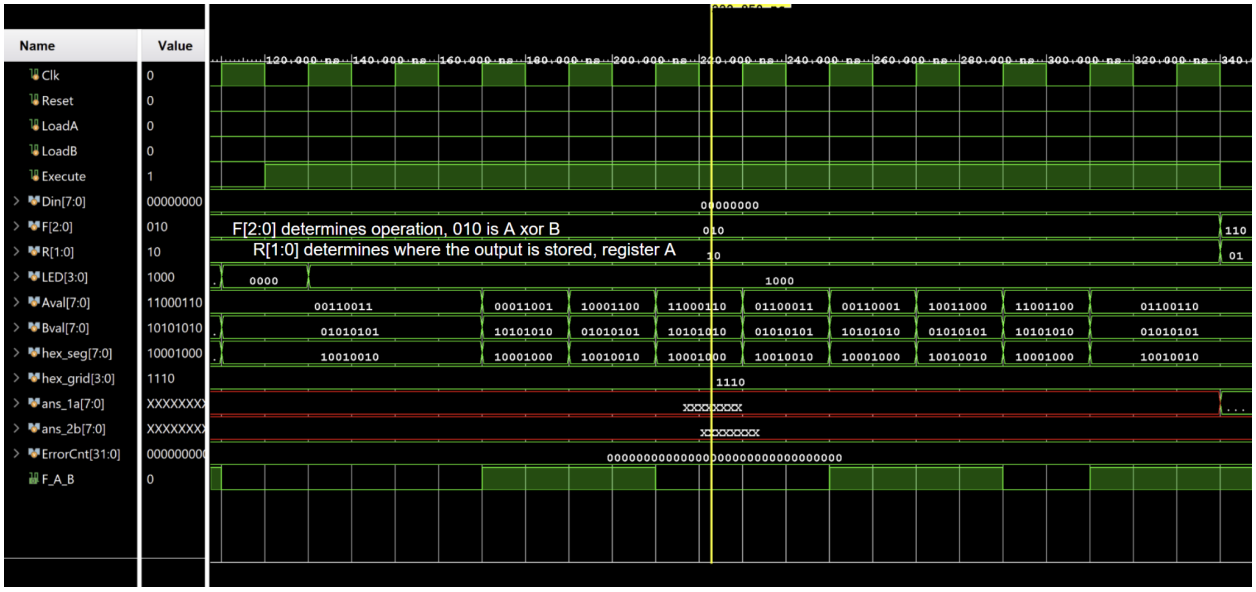

Figure 12: Waveforms from synthesis view

Synthesizing System Verilog is the first step to programming the FPGA. This tests the logic / expected function of the HDL. The above is a view of the simulation waveforms. 00110011 is loaded into A, as evidenced by the initial values of Aval[7:0]. 01010101 is loaded into B, as evidenced by Bval[7:0]. F[2:0] is set to 010 or XOR, and

R[1:0] is set to 10, where the output is sent to register A. Execute is toggled high and then the operation begins. Shifting takes two clock cycles to begin due to synchronization; a button is asynchronous so it is fed into a flip flop and that signal is outputted on the next clock cycle; this renders the circuit synchronous but delays execution by one clock cycle.

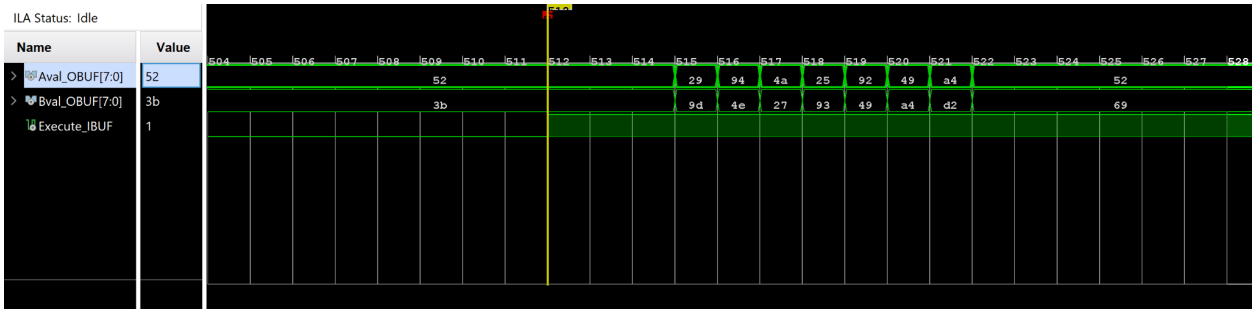

Figure 13: Waveforms captured on FPGA from debug core

Debug cores visualize the signals on the actual implementation of HDL on the FPGA (as opposed to the expected performance from synthesis). Once a debug core has been made and the FPGA programmed, it will redirect to a debug core window. Add Aval, Bval both as data type and the Execute signals as the trigger type operator ==. Then press run trigger, as the debug core gathers 512 ms before the trigger is tripped and 512 ms after from the default sampling time.

Following that, load in desired values and select computation/routing operations. 52 was loaded into A by toggling D7-D0 to 01010010 respectively with 0 being off and 1 being on, then pressing button 2 (load B). 3b was loaded into B by toggling D7-D0 to 00111011 respectively, then pressing button 1 (load A). Then hit button 3 to execute. The above waveforms will be generated.

PostLab

Describe the simplest (two input one output) circuit that can optionally invert a signal (i.e., one input determines if the output is equal to the other input or equal to the other input inverted). Explain why this is useful for the construction of this lab.

Constructing a complicated breadboard circuit such as this invariably results in bugs. Floating inputs can result in undefined behavior for chips, so care was taken to make sure every unused input was either high or low. A color scheme for wires was also utilized to make signals / circuit parts more clear to facilitate debugging.

The simplest circuit (2 inputs, 1 outputs) where one input determines if the output is the other input or the negation of the other input is an XOR gate. Given an input I1, if I0 = I1 then the output is the inverse of I1. Otherwise, I0 != I1 the output is I1.

Explain how a modular design such as that presented above improves testability and cuts down development time.

A modular design improves testability as smaller parts are tested. Testing smaller parts reduces complexity of the debugging process, and makes errors easier to identify. If we write all the code in one single module, it will take longer to synthesize through the whole thing, but when written into different files, they will synthesize at the same time.

Discuss the design process of your state machine, what are the tradeoffs of a Mealy machine vs a Moore machine?

A Moore state machine is one where outputs depend only on current state. This design is simpler to plan out logically. As such a Moore machine is typically designed first when planning the circuit. A Mealy machine is one where outputs depend on inputs and current state. This can greatly simplify the amount of digital logic needed, however it is more complicated to design. In this lab a Moore machine was collapsed down to a Mealy to simplify circuit design.

What are the differences between vSim and Vivado Debug Cores?
Although both systems generate waveforms, what situations might vSim be preferred and where might debug cores be more appropriate?

vSim is from your testbench and is a synthesis level test, it tests your HDL / logic. Debug cores are an implementation level test and checks the circuit on the FPGA itself. vSim is preferred when testing the logic of your HDL / synthesis outputs as it is much faster. Debug cores are to test the function of the circuit on the FPGA, after vSim is functioning as expected as it is slower.

Conclusion

A 4 bit logical processor was designed to take two 4 bit values A, B and perform a bitwise operation on these. A modular design consisting of 4 components: register, computing, routing and control unit. Following this, System Verilog code was provided for a 4-bit processor and we extended the functionality to an 8 bit processor.

During construction of the 4-bit breadboard processor, loose wires were an issue. The wiring could have been tighter, as we used wires that were rather loose. The System Verilog portion had no noticeable debug issues.