ECE 385

Fall 2023

Experiment 2.1 and 2.2

Designing a bit serial logic operation processor

Cher Rui Tan, Adithya Balaji
Section: Friday 11.30am
TA: Gene Lee

Introduction

In this lab we design a 4 bit-serial logic operation processor on a breadboard using two 4-bit shift registers, multiplexers (MUXs), and a counter. The aim of the circuit would be to calculate 8 functions and to route these function results in 4 ways, utilizing a Finite-state machine (FSM) as the control unit of the circuit. The 8 operations that the circuit performs are: AND, OR, XOR, forced high output, NAND, NOR, XNOR, and forced low output. This circuit was then implemented in SystemVerilog extending the processor to 8 bits and implemented on the Urbana board FPGA using Vivado.

Operation of logic processor

In order for a user to load data into the A register, use SW0 to SW7 to get the required number, and then input a high Load A signal by holding button KEY2.

In order for a user to load data into the B register, use SW0 to SW7 to get the required number, and then input a high Load b signal by holding button KEY1.

In order to perform a computing and routing operation after loading in A and B, choose the desired computation to perform using switches SW8 to SW10 for F[0] to F[2], using the truth table in figure 1. Additionally, choose the required routing operations using SW11 and SW12, which correspond to R[0] and R[1] respectively, using the truth table in figure 2. Finally, hit the execute, which is push-button KEY3, to perform the entire operation based on the inputs and the selected computation and routing logic.

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

Figure 1: Computation truth table

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 2: Routing truth table

Written description

Shift Register Unit:

Each 8 bit register unit utilizes two 4-bit BIDIR shift registers to hold data of A and B. This register is chosen as it has a holding state which is crucial for us to use as a display of the bits. Furthermore, the data sheet for this shift register was more clear in its description. The shift register takes a one bit data input for A* and B* from the routing unit as well as a shift input for S1 and S0 to determine the shift holding.

Computing Unit:

The computing unit takes in a one bit input from the register unit and uses NAND and NOR logic to compute logical bitwise operations one bit at a time for 4 clock cycles, then directs the output to the Routing Unit.

Routing unit:
The Routing unit takes a one bit input of A, B and the output of the function f(A,B) before it utilizes two 4-1 Muxes, one for each register to decide on what inputs to output back into the shift registers. This selection is based on the select bits placed at S11 and S12, refer to Figure 2.

Control unit:

The control unit is the most complex part of the circuit. Due to the presence of a timer, there are only 2 required states within this **Mealy** Machine, with the 4 clock cycle shift implemented by the counter.

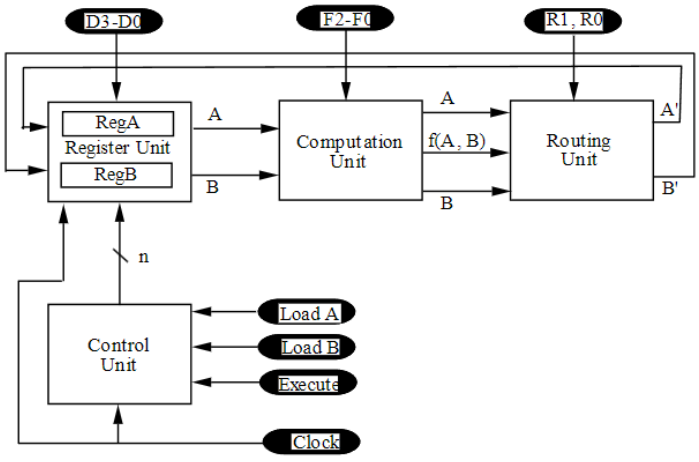

Figure 3: Block Diagram

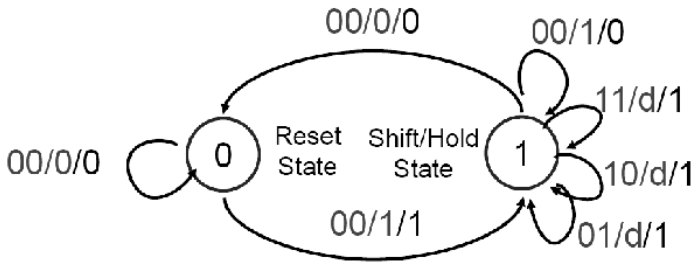

Figure 4: State Diagram

As seen from Figure 3, we have the reset state represented by the flip flop value of 0 and the Shift/Holding state represented by the flip flop value of 1. The states have the label XX/X/X, with the first 2 bits being the counter bits C1C0, followed by the Execute ("E") input and finally the Register Shift ('S') output which indicates that the counter is running and the operation is being executed.

Design steps

Register unit:

No k-maps or truth tables were used. Two BIDIR shift registers were used for A and B, as opposed to the sync counter, as we needed shift registers with clock input and capable of dealing with parallel input.

Computation unit:

For the computation unit, we used the following truth table in Figure 5.

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

Figure 5: Truth Table for computation unit

From the truth table, to obtain the outputs for each of the 4 functions (in the case where F2 is 0),

instead of using purely NAND logic, we simply used its corresponding inverted gate along with an inverter. In the case of an AND gate, we used two NAND gates. the first connected to A and B signals, and the second to invert the output of the original NAND gate. Similarly for the OR gate, we use a NOR gate wired to A and B then another NOR gate to invert the first's output. For XOR we used a 4x 2 input XOR chip and for "1111", we used a forced high directly from the power rails. When this final output is passed through an XOR gate with F2 the answers are flipped.

We observe that F1 and F0 are the select signals with F2 simply acting as an inverting bit by producing opposing outputs of the first 4 outputs when it is high. A AND B, when F2 is 0, becomes A NAND B, when F2 is 1. Likewise for the other operations. Thus, we designed the circuit such that we would obtain the output f(A,B) from the signals F1 and F0, we then use F2 to selectively invert the signal by XORing it with the output.

The figure below presents a CAD drawing of the circuit logic, using AND/OR gates.

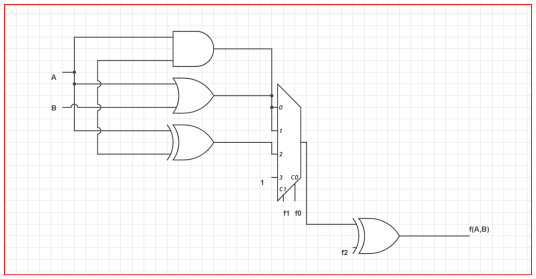

Figure 6: Circuit CAD schematic of computation unit using AND/OR gates

This circuit is modified to accommodate using NAND and NOR gates, and the final CAD schematic is shown below.

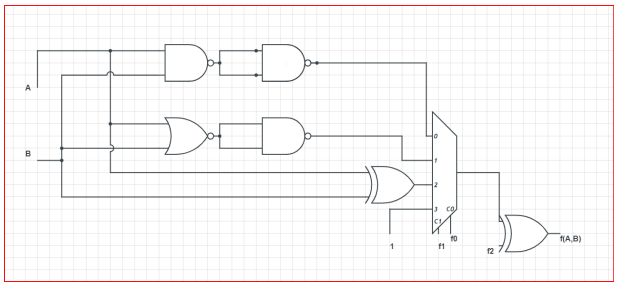

Figure 7: Circuit CAD schematic of computation unit using NAND/NOR gates

Routing unit:

For the routing unit, we used a dual 4-to-1 MUX chip, with one MUX used to output A* and the other used to output B*. The truth table for this is shown below in Figure 7.

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 8: Truth table for routing unit

The select for R1 and R0 would later be routed to SW11 and SW12 on the FPGA.

Control unit:

For the control unit, we design a Mealy FSM, using the truth table presented below: We decided on using a Mealy machine as we observed that the Moore machine would require many more states resulting in next state logic being more complex. However, with our implementation using a reset state and a shift/holding state as well as a 4 clock cycle counter, we are able to implement a four bit shift which allows for the bitwise computation to take place. With C1 and C0 abstracted from the table, we determine the next state logic for Q our only state by drawing the K-map.

Exec. Switch ('E')	Q	C1	C0	Reg. Shift ('S')	Q^+	$C1^+$	$C0^+$
0	0	0	0	0	0	0	0
0	0	0	1	d	d	d	D
0	0	1	0	d	d	d	D
0	0	1	1	d	d	d	D
0	1	0	0	0	0	0	0
0	1	0	1	1	1	1	0
0	1	1	0	1	1	1	1
0	1	1	1	1	1	0	0
1	0	0	0	1	1	0	1
1	0	0	1	d	d	d	D
1	0	1	0	d	d	d	D
1	0	1	1	d	d	d	D
1	1	0	0	0	1	0	0
1	1	0	1	1	1	1	0
1	1	1	0	1	1	1	1
1	1	1	1	1	1	0	0

Figure 9: Truth Table for Mealy machine

Based on this truth table we derive the K-maps and equations, to design the logic.

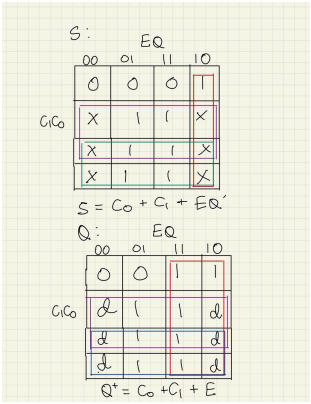

Figure 10: K-maps and equations for control unit

Following this truth table and drawing out K-maps, we present the CAD drawing of the circuit layout using AND/OR gates below:

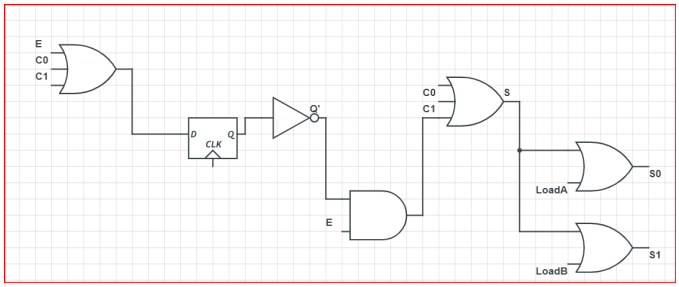

Figure 11: Circuit CAD schematic of control unit using AND/OR gates

Subsequently, the modified final circuit for the control unit that uses NAND/NOR gates is shown below.

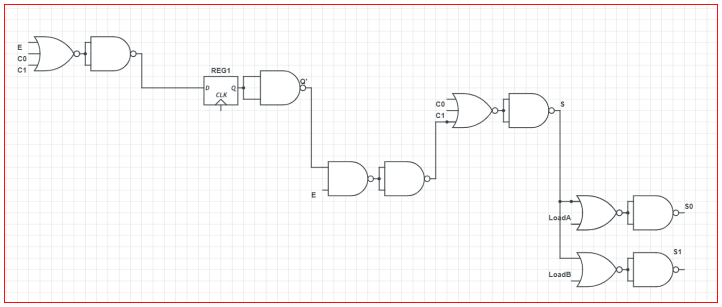

Figure 12: Circuit CAD schematic of computation unit using NAND/NOR gates

Having designed the individual module, the figure below shows the entire circuit, with each

module being represented by blocks.

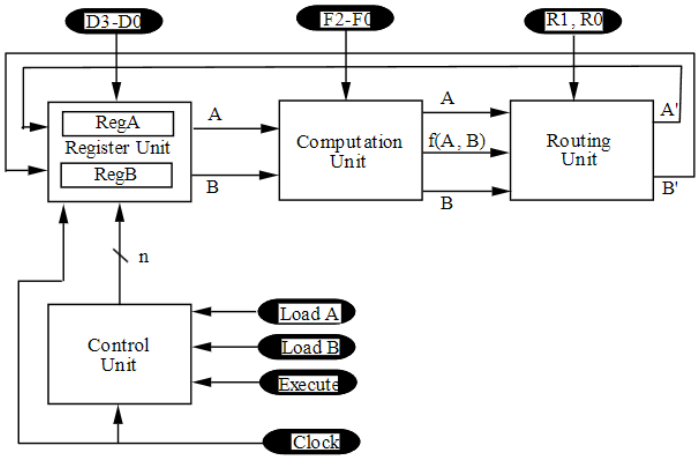

Figure 13: Top level layout

We present the chip layouts for the circuit using the manual layout sheet in the figure below.

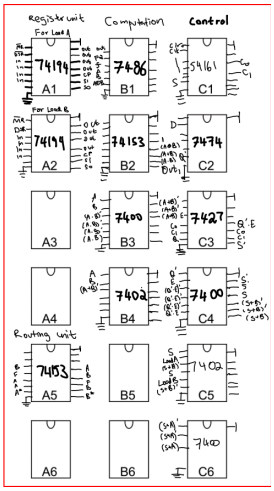

Figure 14: Manual Chip layout with units labeled

8-bit logic processor on FPGA

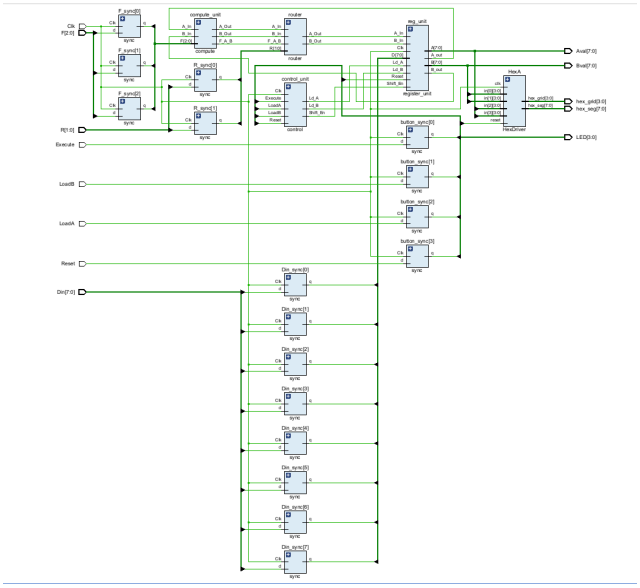

Figure 15: RTL Block Diagram

SystemVerilog Modules

Below is a summary of all the modules. The key changes made to extend the processor to 8-bits were to change the number of bits for Aval, Bval, hex_seg, D_in, A, B, D and Data-out from [3:0] to [7:0] to extend the same operation from 4 to 8 bits.

Module: Processor.sv:

Inputs: logic clk, reset, loadA, loadB, execute,
[7:0]D_in (input data)
[2:0]F (function select)
[1:0]R (routing select)
[3:0]LED (debugging)
[7:0]Aval, [7:0]Bval (debugging)

Outputs: [3:0]LED (debugging)
[7:0]Aval, [7:0]Bval (debugging)
[7:0]hex_seg (for hexadecimal display control)

[3:0]hex_grid (for hexadecimal display control)

Description: Declare local logical variables. Instantiate all the modules. Call Hexdriver to view the upper nibble of registers by converting digital logic outputs to integer outputs. Input synchronisers for asynchronous inputs (from the switches).

Purpose: Module encompassing the entire logic process.

Module: reg_unit (Register_unit.sv)

Inputs: logic type: Clk, Reset, A_In, B_In, Ld_A, Ld_B, Shift_En,
input logic [7:0] D

Outputs: output logic A_out, B_out,
output logic [7:0] A,
output logic [7:0] B);

Description:
Top level register unit which comprises two 8 bit shift registers reg_A and reg_B. These instantiated classes accept the right shift bit of A_In and B_In from the routing unit, output both left bits into the computational unit and the final 8 bit output to the Hex driver for the display.

Purpose: To control interaction between other modules and the register, instantiating the 8 bit shift registers representing data values A and B.

Module: reg_4 (Reg_4.sv)

Inputs: input logic Clk, Reset, Shift_In, Load, Shift_En,
input logic [7:0] D,

Outputs: output logic Shift_Out,
output logic [7:0] Data_Out);

Description:
This module comprises a 8-bit shift register unit which shifts while the Shift_EN signal is high. It also contains a reset function which resets all data values to 0 when the Reset signal is high and a load function which loads in [7:0]D when the Load signal is high.

Purpose:
To control the shifting of the 8 bit register unit such that the data is transferred one bit at a time to the computational unit for the bitwise operation to be carried out. The Shift_En also allows for a holding state before and after computation of 8 cycles have taken place.

Module: compute (compute.sv)

Inputs: input logic [2:0] F,
input logic A_In, B_In,

Outputs: output logic A_Out, B_Out, F_A_B);

Description:
This module mainly contains the combinational logic for the bitwise computation selected by the user using the F0, F1, F2 corresponding to switches SW8, SW9, SW10. This combinational logic can be seen from the circuit schematic for the computational unit.

Purpose:
The computational units is used to compute bitwise operations on one bit of A and B based on the selection of the signals of F2, F1 and F0

Module: router (router.sv)

Inputs: input logic [1:0] R,
input logic A_In, B_In, F_A_B,

Outputs: output logic A_Out, B_Out);

Description:
Acts as a multiplexer, selecting one of the input values of A_In, B_In, F_A_B based on the 2 bit select signal R and forwards it to either A_out or B_out.

Purpose: To act as a routing unit to control which register the output is fed into.

Module: control (control.sv)

Inputs: input logic Clk, Reset, LoadA, LoadB, Execute,

Outputs: output logic Shift_En, Ld_A, Ld_B ;

Description:
The module represents a finite state machine that controls the loading of data in registers A and B as well as the shift operations. The state machine transitions between a series of states, allowing for bitwise operations to take place at each state after the execute signal is pressed. The module also has a reset function that resets A and B data values to 0 when on high.

Purpose: To implement the FSM which dictates movement of data in the routing unit and allows for bitwise operations to take place in the computation unit.

Module: hexdriver (hexdriver.sv)

Inputs: input logic clk,
input logic reset,
input logic [3:0] in[4],

Outputs: output logic [7:0] hex_seg,
output logic [3:0] hex_grid

Description: Implements a 4-digit hexadecimal display driver by using a counter to cycle through 4 digits, and selecting the appropriate pattern based on the input nibbles

Purpose: To display hexadecimal numbers on the 4-digit 7-segment display

Module: sync (synchronizers.sv)

Inputs: input logic Clk, d,

Outputs: output logic q

Description: Takes asynchronous signals and holds them in a flip flop allowing for them to be implemented on a clock signal. This results in the signals being converted from asynchronous to synchronous signals.

Purpose: To eliminate asynchronous components such that the entire circuit remains synchronous to reduce the chance of static hazards arising from timing issues and ensure

proper signal handling in FPGA designs.

Simulated waveform

The simulated waveform with annotations is presented below. From the waveform, we see that Aval and Bval are loaded with 8 bits each, and F2, F1 and F0 is initially set to 010 (leading to A XOR B) and R1, R0 is set to 10, leading to f(A,B) to be stored in A* and B to be stored in B*. We see that it worked based on a clock as the shifting is done in 8 clock cycles. We see that the operation also takes two cycles after Execute is hit, as it needs to be made synchronous with the rest of the circuit.

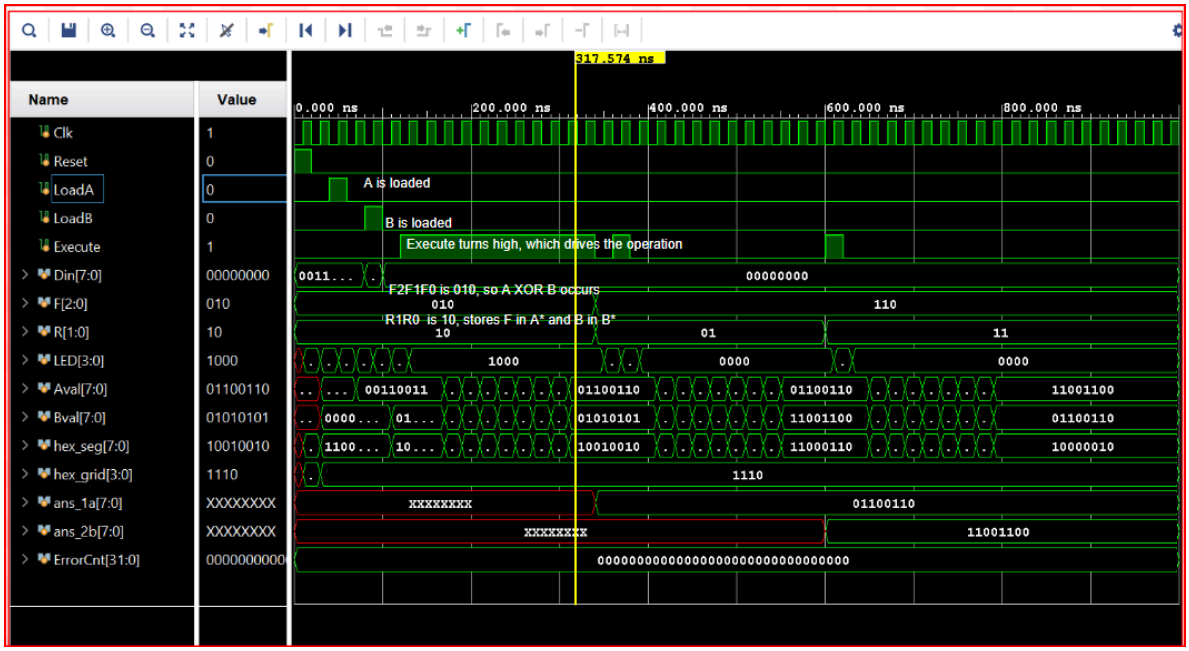

Figure 15: Annotated simulated waveform

Debug Cores

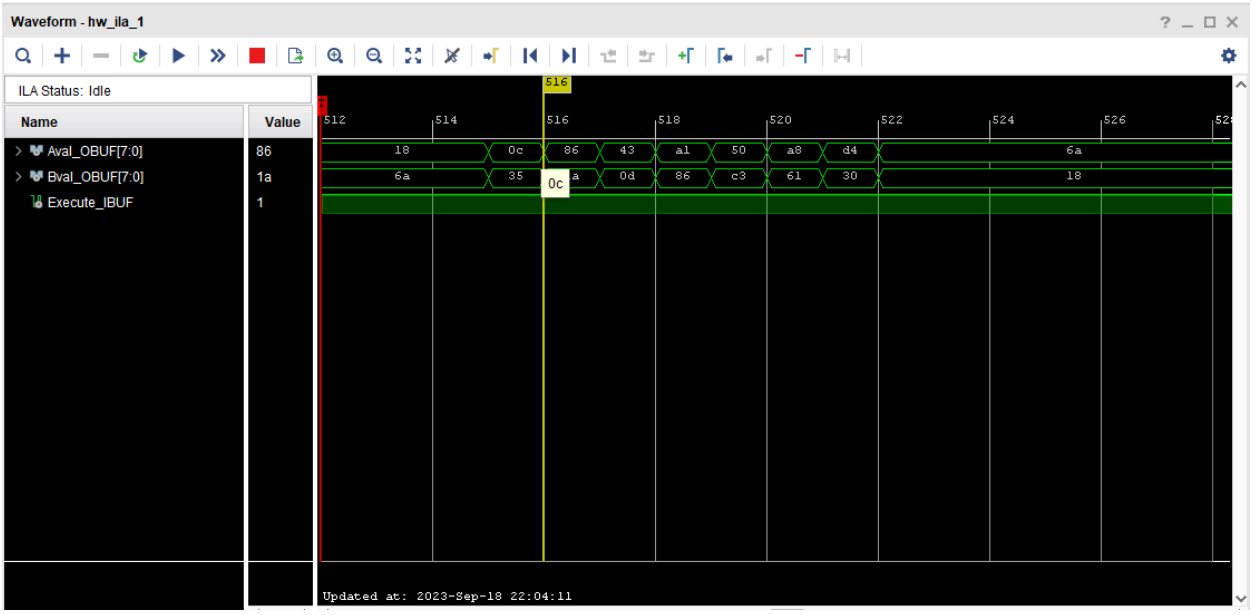

Figure 16: Debug Cores Waveform

To generate a Debug core, we first Set Up the debug core, this involves adding the Aval, Bval and assigning them as data values followed by adding the Execute_OBUF value as a data and trigger value. Next, we left the sampling at 1024ms, after generating bit stream and programming the board, and running the ILA Debug core, we see that the Debug core has sampled half the data 512ms before the trigger has been pressed. After the trigger is pressed we can see that the Debug core has sampled the other half of the data and we are presented with the data of Aval and Bval after the computation.

We can see that the swap operation has been carried out where R1 and R0 are high. This causes a bitwise replacement of A with B resulting in the Aval changing from x18 to x6a and the Bval changing from x6a to x18. Throughout the 8 cycles we can see the slow replacement of bits for Avals from x18 to x0c then x06 and x43. This would allow us to determine if there were intermediate errors taking place and at what steps they were incurred.

Answers to Post-Lab Questions

Question 1: 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.

The simplest circuit to achieve this, which we did use in the lab, is to have a dual-input XOR gate wired to the inverting input as well as an inverting signal, thus achieving a select inverter which inverts when the inverting signal is high. This is useful for the lab as it allows us to avoid using an 8-to-1 MUX, which would be more complicated to implement as well requiring more gates to implement the inverted logic.

Question 2: Explain how a modular design such as that presented above improves testability and cuts down development time.

By using a modular design, it is possible to test whether the logic for each of the 4 units works independently of one another, by forcing the inputs to these units to be what we need for testing (either high or ground). This also ensures that we do not have to troubleshoot the entire operator at once, but only the portion that has some test cases failed, which cuts down the time spent on troubleshooting.

Question 3: Discuss the design process of your state machine, what are the tradeoffs of a Mealy machine vs a Moore machine?

While designing our state machine, we decided to use a Mealy machine as it has fewer states than a Moore machine for the implementation of the control unit. This is because outputs do not depend solely on the current state, they can depend on both the state and the inputs. This allows us to implement a counter for bit shifting in the circuit, saving us 4 states of counting.

Question 4: 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 a simulated verison of the logic, primarily used in early stage testing, right after running synthesis, and does not run in real time or on the FPGA, thus it manages to check whether the logic in the SystemVerilog code should work in theory. However, Vivado Debug Cores are better for hardware or timing related debugging on the actual FPGA, after running implementation, generating a bitstream and programming it to the board. Thus, Vivado Debug Cores would be more suitable for troubleshooting any discrepancies between the expected logic and the actual logic that the FPGA outputs, in real-time.

Conclusions

In conclusion, we showed how we can construct the bit serial operation logic processor for 4 bits of A and 4 bits of B. We managed to obtain working models for both the DE-10 lite board implementation as well as the Vivado implementation. We also answer the Post-lab questions in the section below. In terms of documentation, we felt that everything was fairly clear.

Throughout the process of these two labs, we have gained much knowledge of hardware implementation as well as System Verilog. To summarize knowledge gained, we group it into the hardware aspects and software aspects.

For hardware, we learnt the concept of toggle switches, which prevent floating values when they are turned off. Debouncer circuits to prevent the mechanical bounce of switches were also key for preventing the multiple triggering of the Execute switch in our circuit.

For software, we gained conceptual understanding of different types in SystemVerilog, such as logic and integers. We also gained knowledge of processes like always and initial understanding what modules and variables should be declared where. Furthermore, the difference between declaration and instantiation of a module, where one is analogous to a chip design of what each pin does, and the other more akin to placing the chip on the board and wiring it.

For the most part our circuit worked, with some mistakes present in wiring between chips. In the future, we hope to improve by designing our hardware circuit using KiCad to simulate it first before actually wiring it on the board. This removes small issues such as accidental wiring mistakes and allows us to debug our circuits more easily.