Ellis Chen
10/23/2022

ECPE 174 – Lab 7 Report

Problem Statement:

This report discusses the design and implementation results of a 4-way handshaking communication protocol between two Cyclone boards. The 4-way handshaking protocol requires one channel to send data and another channel to receive data for each board. The handshaking needs to be designed in such a way that both boards can determine which is writing and which is being written to; also communicating the status of the target board, whether it is still in the process of receiving or has finished. The signals for data transmission are specified as *devA* and *devB*. A *send* signal must be pulled high on one board to initiate a write transaction to the other board.

In the event of a write transaction from the perspective of board A, for example, board A pulls *devA* high after *send* goes high. After board B sees that *devA* from board A went high, board B pulls *devB* high on board A. The *send* signal can go low after the write transaction starts. Board A pulls *devA* low after its *devB* goes high. After board B sees that *devA* from board A went low, board B pulls *devB* low on board A. The write transaction from the perspective of board A finishes after this process.

In the event of a read transaction from the perspective of board A, a *send* signal is not required to initialize the transaction. Board B starts the transaction by pulling *devB* high on board A. Board A pulls *devA* high after it acknowledges board B. Then, board A pulls *devA* low after board B pulls *devB* low, finishing the read transaction from board A's perspective. Also, the handshaking is bidirectional, so writing and reading can also be done from the perspective of board B.

System Design:

I designed a Moore-based FSM because it is more intuitive to treat the signal *devA* as an output from the perspective of board A. It should be made clear that *devB* and *send* are input signals that determine the state relative to *devA*. My FSM design contains 4 states: idle, write, wait, and read; the states and their inputs/outputs are shown in figure 1.

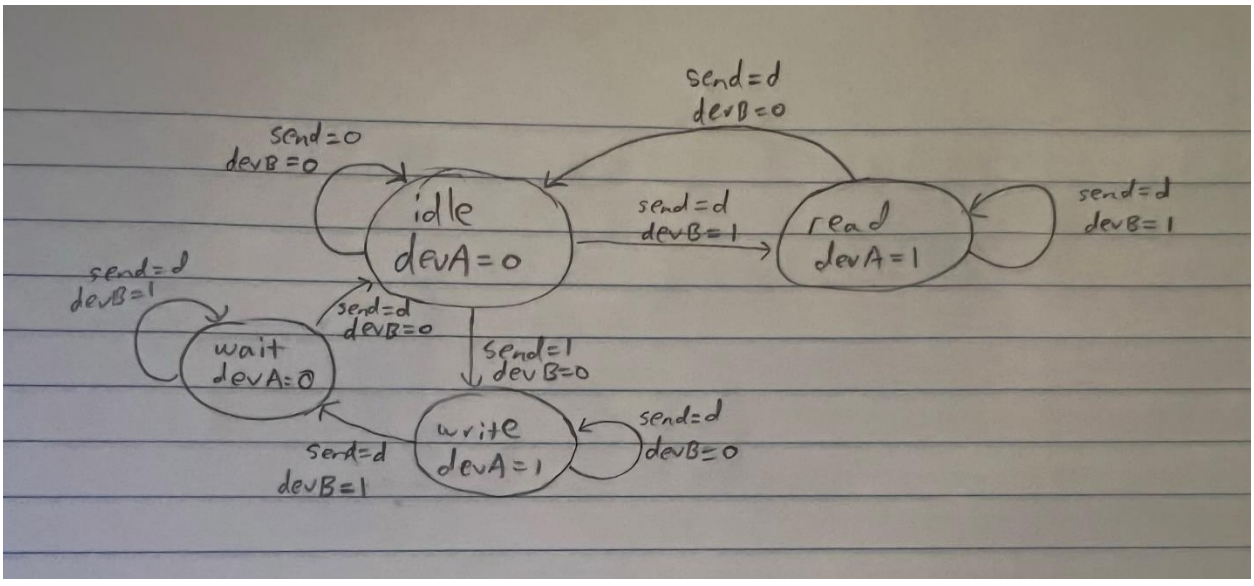

Figure 1. Moore FSM shows states for reading and writing from device A's perspective

Ellis Chen
10/23/2022

The timing diagrams that show the expected signal behavior for both writing and reading transactions are shown in figure 2. Write operations are expected to occur after *send* goes high. *devA* is expected to go high shortly after. *devB* is also expected to go high after *devA*. *devA* is expected to stay high as long as there is valid data needs to be sent; it is expected to go low after there is no more valid data. *devB* goes low shortly after to end the transaction. The FSM from figure 1 is designed in such a way that *devB* cannot transition levels sooner than *devA* during a write. Also, in the event that *send* and *devB* are both asserted high, then *send* will be ignored and a read transaction will begin.

For read transactions, the *send* signal does not matter; the *devB* signal goes high to start the receiving transaction. *devA* is expected to go high to acknowledge the receive. *devB* is expected to go low after it is done sending data, and *devA* is expected to go low shortly after to end the transaction. *devA* should not go low sooner than *devB* during a read transaction.

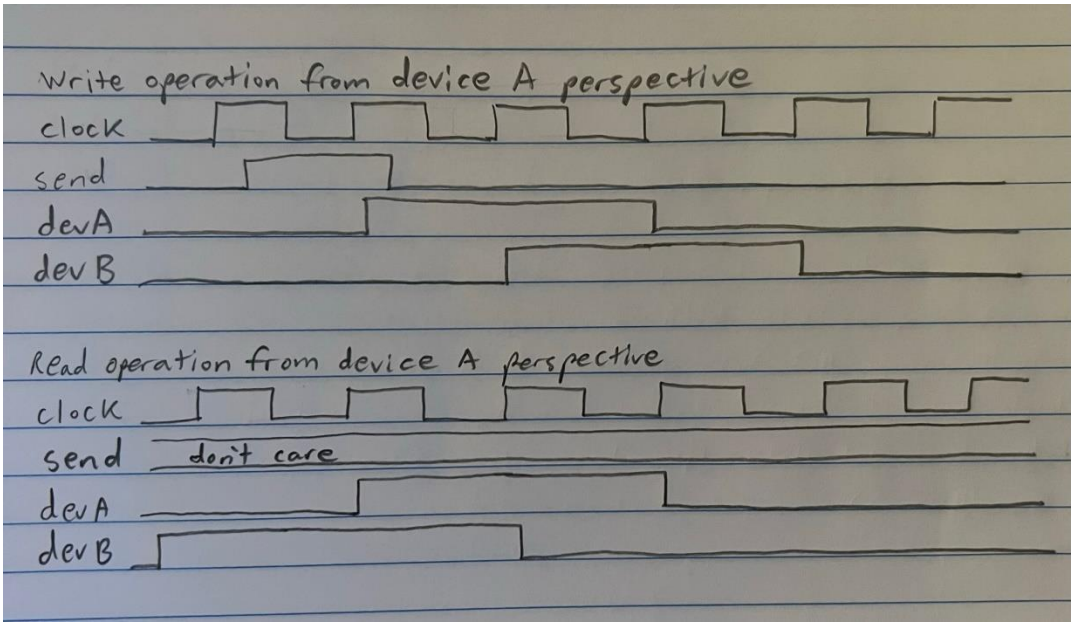

Figure 2. Timing diagrams for reading and writing from device A's perspective

The system is also designed to support reset functionality. The reset is expected to be an asynchronous-assert and synchronous-deassert input signal. The reset signal is designed to stop any transaction in progress and reset the FSM to its idle state.

The input signals, *devB* and *send,* are asynchronous; the system implementation requires two instantiations of synchronizers for these inputs, which are shown in figure 3. The level of *devB* is more important for the design than its edge transition; it is more valuable to compare *devB* to *devA* in terms of logic level. The edge of *send* is more important than its level because only the event of the signal asserting and deasserting is useful.

```
synchronizer syncSend(.clk(oclk), .sig(send), .falling_ind(sendEdge), .reset(rst));
synchronizer syncDevB(.clk(oclk), .sig(devB), .sigSync(devBLevel), .reset(rst));
```

Figure 3. System Verilog synchronizer submodule code shows logic wiring for *send* and *devB*

Ellis Chen
10/23/2022

Testing Approach:

In simulation, I ran one test case for writing, one test case for reading, and one test case for writing with a reset signal. The test cases served as a means to verify that the device under test covers all the states of the FSM from figure 1 correctly based on its inputs.

I tested the write transaction by following the sequence in figure 2. In a testbench task, I used while loops in the testbench task, shown in figure 4, to wait for *devA* to go low; *devA* should be low before any transaction can start. I made send go high to start the transaction. To test if *devA* goes high after I made *devB* and *send* go low, I used a while loop shown in figure 5. The process continues for testing the write transaction using while loops to wait for *devA* to update in the FSM.

I also tested the read transaction by following the timing diagram in figure 2, and using similar techniques shown in figure 4 and 5. I also tested the reset functionality of the system by setting the reset signal low in the process of a write transaction. In the case that the FSM is in the write state, the reset should set the state back to idle.

```
repeat(3) @(negedge clk);
send <= send_in;    // send goes high to initialize transaction

while (devA)
    @(posedge clk); // wait for devA to go low
#1 tc1: assert(send) else $display("%t, Send should be 1'b1", $time);
```

Figure 4. System Verilog code snippet shows task waiting for *devA* to go low

```
repeat(3) @(negedge clk);
devB <= devB_in;    // devB goes low
send <= ~send_in;   // send goes low

while (!devA)
    @(posedge clk); // wait for devA to go high
#1 tc2: assert(!devB) else $display("%t, devB should be 1'b0", $time);
tc3: assert(!send) else $display("%t, Send should be 1'b0", $time);
```

Figure 5. System Verilog code snippet shows task waiting for *devA* to go high

For single board implementation, I also tested the write, read, and write with reset transactions. I tested that the active-low pushbuttons worked with the active-high logic of my FSM design. I also tested my synchronizer design for handling the *send* and the *devB* input signals. I also tested that *devA* toggles the LEDs on the board during the transactions.

Much of the same tests were performed for the two-board implementation. I tested the communication between two boards via GPIOs to verify that only one board is transmitting and one board is receiving at a time.

Ellis Chen
10/23/2022

Results:

My simulation was successful for all proposed testcases. Figure 6 shows the waveforms for writing, reading, and writing with reset testcases. Figure 7 shows that the design compiled successfully.

The cyclone board implementation was also a success. For a single board during a write, *devA* always remained high until *devB* went high for a few seconds. *devA* went low before *devB* went low, which is the behavior that I expected in my design. For reading, *devB* would always stay high until *devA* went high for a few seconds. *devA* went low after *devB* went low, which is lso th behavior I expected for a read transaction. The reset signal also worked, in which the LEDs indicating a transaction in progress turned off as the reset signal went low.

The two-board implementation was also a success. By observing the two boards during a transaction, only one LED updates on each board at a time, which shows that communication between two boards is actually occurring.

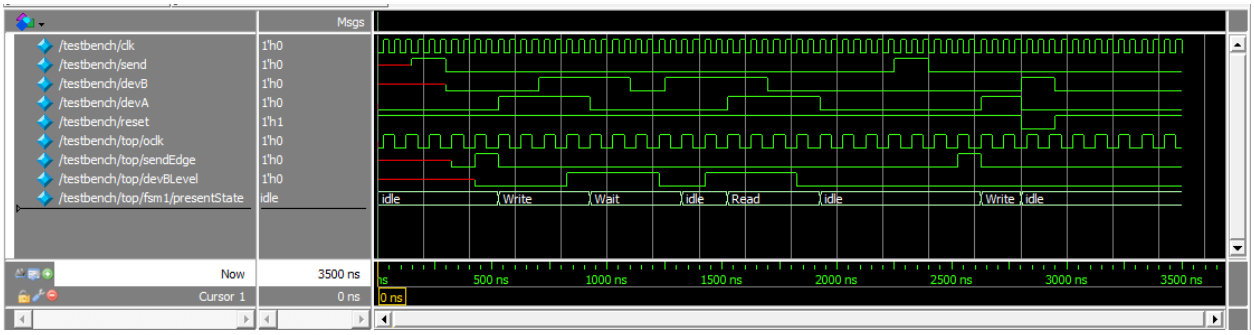

Figure 6. Simulation Waveform shows write transactions with/without reset and a read transaction

```
#
# Top level modules:
#       testbench
# End time: 18:31:31 on Oct 23,2022, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# End time: 18:31:32 on Oct 23,2022, Elapsed time: 0:05:41
# Errors: 0, Warnings: 10
# vsim -voptargs="+acc" testbench
# Start time: 18:31:32 on Oct 23,2022
# ** Note: (vsim-3813) Design is being optimized due to module recompilation...
# Loading sv_std.std
# Loading work.testbench(fast)
# Loading work.toplevel(fast)
# Loading work.clockdiv(fast)
# Loading work.resetsync(fast)
# Loading work.synchronizer(fast)
# Loading work.fsm(fast)
# .main_pane.structure.interior.cs.body.struct
# .main_pane.objects.interior.cs.body.tree
#                    0 ns, Write transaction
#                 1126 ns, Read transaction
#                 2100 ns, Write transaction with Reset
# .main_pane.wave.interior.cs.body.pw.wf
# 0 ns
# 3675 ns

VSIM 3>
```

Transcript | Objects | sim

Figure 7. Transcript Window shows types of transactions throughout the simulation

Ellis Chen
10/23/2022

Analysis:

When *devA* and *devB* are asserted at the same time in my design, my state machine is either changing states to “wait” or “read” according to figure 1. If my state machine was at idle, the dual assertion would put the FSM in the “read” state. If two devices run my code, then this could raise an issue if both devices are trying to read from each other. Neither *devB* nor *devA* would deassert if this scenario occurred, so the devices are locked in a read only state. To alleviate the problem with this dual assertion, I propose implementing timer-like logic that will reset everything back to its known state after a predetermined amount of time.

If I had to modify the system to communicate data, then I would have to include request and acknowledge signals for both reading and writing. Instead of each board using two signals *devA* and *devB*, the modified system would have each board using four signals; two signals dedicated for reading and two signals dedicated for writing.