Clock Domain Crossing

EEL 4720/5721 – Reconfigurable Computing

Objective:

In this lab, you will learn how to properly communicate across clock domains. If not handled correctly, signals that cross clock domains can become metastable, which if propagated through your circuit will likely cause errors. In this lab, you will learn how to create ***synchronizers*** that guarantee (with a high probability) that signals have stabilized before being used in the destination domain. In this lab, we will be looking at 3 types of synchronizers: dual-flop, handshake, and FIFO synchronizers.

You will again be working in groups of 2 on this project. Unless you are an EDGE student or have special permission from the instructor, you must work in a group of 2. Note that those in any overflow sections are *not* EDGE students, and must work in a group of 2. Please only submit once per group.

EDGE INSTRUCTIONS:

EDGE students must complete parts 1 and 3. Part 2 may be done for extra credit.

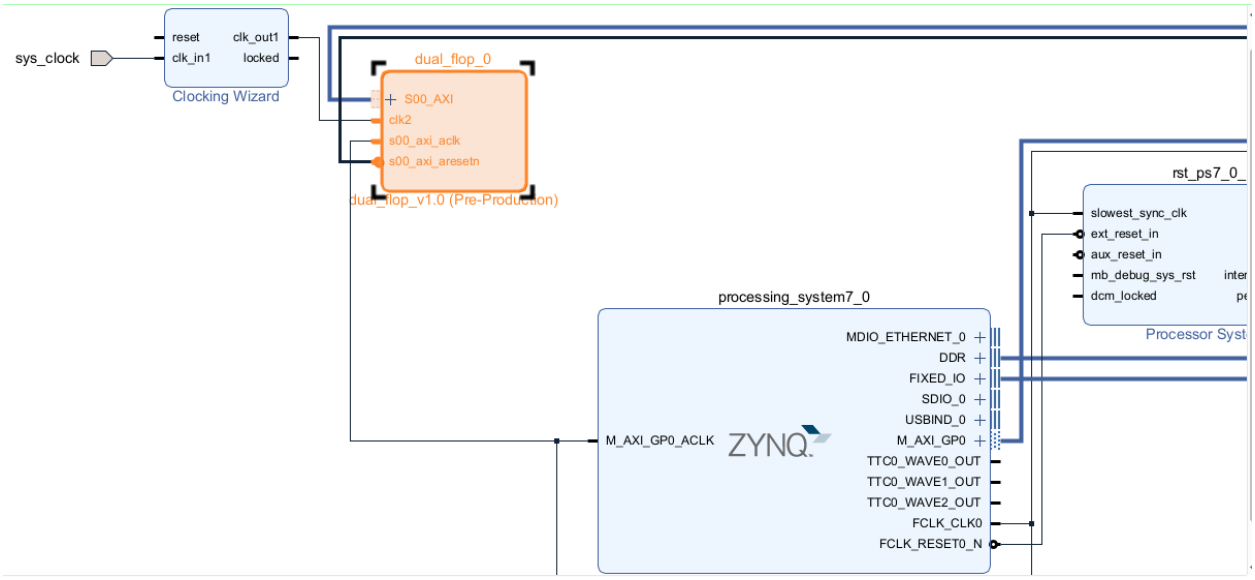
Part 1 – Dual-flop synchronizers

In the simplest case of clock domain crossing, a single bit must be synchronized. This commonly occurs for simple control signals, such as a *go* or *enable*. Many beginners assume that metastability will not be a problem for these signals, because even if the proper value isn't used on the current cycle, it will eventually stabilize and be correctly seen. Although there may be situations where such assumptions can be valid, it is much safer to properly synchronize these signals and avoid unanticipated issues.

In this part of the lab, you will see this problem. The provided code (see the part 1 directory) provides VHDL code for the ZedBoard that uses two domains. user_app.vhd is the top level. In the first domain, there is an entity that produces a memory-map-specified number of pulses on a signal that crosses the clock domains. The destination domain monitors this signal and counts the number of times that the pulse transitions from 0 to 1. The provided testbench user_app_tb.vhd shows that the provided implementation simulates without errors. However, there is C code provided that shows the VHDL does not work correctly on the board. The reason is that the pulse signal is not synchronized with the destination domain.

For part 1), do the following steps:

1) Simulate the provided VHDL with the provided testbench (user_app_tb.vhd). Note that there are no errors.
2) Create a Vivado project using the provided accelerator dual_flop_1.0 IP core (see earlier labs).
3) For one of the clock domains, we will use the AXI clock. To create another clock that is not derived from the AXI clock, we will instantiate a PLL, which we can do by adding more IP. Select Add IP and then select 'Clocking Wizard'. Double click the clocking wizard in the block diagram to re-customize the IP. Select the "Output Clocks" tab and change the requested frequency to 70 MHz, then click OK.
4) Run the connection automation, which will establish most of the connections. **IMPORTANT: connect the clk_out1 output from the clocking wizard to clk2 on the dual_flop IP. Next, remove the connection from the reset of the clocking wizard (if there is one). We will leave it disconnected.** Your resulting block diagram should look something like this:



5) Generate a bitfile and rename it to part1.bit.
6) Upload the bitfile and provided C++ code to the class server. Compile the code and run it using:

   zed_schedule.py ./zed_app part1.bit
6) Verify that the circuit does not work correctly. The C++ code will output the difference between the actual count provided by the circuit and the correct result. This difference will likely change every time you run the code.

   Save a screenshot of the output in part1_incorrect.jpg.
7) Add a dual-flop synchronizer to the pulse signal and make any other necessary changes to the code.
8) Simulate using the provided testbench until there are no errors (you may have introduced some).
9) Update your IP in Vivado and repeat 5-6 to verify that your dual-flop synchronizer has fixed any metastability problems.

Part 2 – Handshake synchronizers

Unfortunately, the dual-flop synchronizer cannot reliably be used to transfer multiple-bit signals. To deal with multi-bit signals, one form of synchronization is a handshake. The source domain initially puts data into a register that crosses clock domains. However, the destination domain does not immediately use that data. Instead, the source domain sends a data valid signal to the destination domain, which is then acknowledged by the destination domain. These messages are single bits that can be properly synchronized using dual-flop synchronizers. After receiving the data valid message from the source domain, the destination domain can safely use the multi-bit data, because it should now be stable. After receiving the acknowledgement from the destination domain, the source domain can change the data in the register and start another transfer.

The provided code in the handshake directory shows an incorrect implementation of a handshake synchronizer (user_app.vhd is the top level). In this example, there is an input block

RAM (in clock domain 1) that transfers data to a datapath (in clock domain 2). The datapath then sends outputs to an output block RAM (in clock domain 1). Like in the previous part of the lab, the provided code simulates perfectly, but does not work on the actual FPGA. You will fix the VHDL to properly handle the handshake.

You have two options for your implementation. A level-sensitive handshake implements the send and acknowledge as being asserted at a particular level. Although this works, it requires two round-trips: assert send, wait for ack, deassert send, wait for ack to reset. A more efficient way is to use an implementation that is sensitive to transitions. See the papers provided on the class website for more information. You will receive full credit for either implementation. Note: the level-sensitive handshake is significantly easier (and is included in the provided code), so I would suggest getting this version working before trying the transition-sensitive version. For part 2), repeat the process of part 1 for the handshake entity using the provided fifo_1.0 directory. Name your bitfile part2.bit.

Part 3 – FIFO with multiple clock domains

Although the handshake enables arbitrarily wide data to be transferred across clock domains, it can reduce throughput and complicates control. For example, the pipelined datapath in part 2 never had more than one valid stage of data at a time because the source domain cannot transfer more data until the destination domain is ready (i.e., has acknowledged the previous data). One way of dealing with this problem is to add a buffer or FIFO that stores data in the destination domain so that the source can immediately start a new transfer. While this works, if we are going to use a FIFO, we can completely eliminate the handshake synchronizer.

One common feature of a FIFO is to support different clock domains for reading and writing. Therefore, we can simplify the implementation in part 2, and greatly improve throughput by adding a FIFO between the input memory and datapath inputs, and between the datapath outputs and the output memory.

One advantage of FIFOs is simplified control. Look at the provided VHDL and read the comments in the top-level user_app.vhd file. The input memory writes to the FIFO anytime it is has valid data and the FIFO isn't full. The datapath always reads from the input FIFO, even when it is empty (the data is just marked as invalid). The datapath stalls anytime the output FIFO is full and writes data anytime its outputs are valid. The output memory reads from the output FIFO anytime it isn't empty.

To finish this part of the lab, you will need to create two FIFOs: one that is 32 bits wide (fifo32.vhd) and one that is 17 bits wide (fifo17.vhd). Creating a multiple-clock domain FIFO is not trivial. Fortunately, in this lab, you can use Xilinx Core Generator (CoreGen), which is included as part of Vivado's IP catalog. **Within the IP project**, open the IP catalog, select "Memories and Storage Elements"->"FIFOs"->"FIFO Generator". For both FIFOs, make sure you use the following settings: native interface, independent clocks block RAM for the implementation option, and first-word fall through for the read mode. The interface widths should be 17 or 32 depending on whether or not this is the output or input FIFO. The depth doesn't really matter here, so make it small (e.g., 64). **For the 32-bit FIFO, make sure to select the option for the almost_full flag (see comments in code for explanation).**

Before accepting the configuration settings for the FIFO core, go to IP Location at top of the window. Make sure the specified directory is the src/ folder within the IP folder. Sometimes Vivado will use a different folder outside of the IP folder, in which case you will get black-box errors when synthesizing in the main project.

After creating the FIFO cores, look at the generated instantiation template in Vivado to see how to use them. Then, create an instance of the corresponding cores within fifo32.vhd and fifo17.vhd.

For part 3), repeat the process of part 1 for the handshake entity using the provided fifo_1.0 core. There are two key differences for part 3. First, the provided VHDL should not simulate correctly until you change it. Second, because you are using pre-synthesized FIFO cores, you will have to run your simulations using Vivado's simulator. To do this, add the provided testbench user_app_tb.vhd as a simulation source if it is not already included as one. Next, set the testbench as the top-level entity under simulation sources. Finally, click run simulation, which should open a waveform viewer and start the simulation. Make sure to select "Run all" to ensure that the simulation is complete.

Make sure to name your bitfile part3.bit.

SUBMISSION INSTRUCTIONS (One submission per group)

Make sure all group member names are at the top of every file that you modify, in addition to the readme file!

Create a directory with your UFID. Give it the following structure:

```
UFID/ readme.txt      // Group members, and anything that the grader
                      // needs to be aware of
      part1/
            part1.bit              // DON'T FORGET!!!!
            part1_incorrect.jpg
            dual_flop_1.0/         // IP core from repository for part 1 with this exact
                                   // name. Make sure all VHDL is included
      part2/
            part2.bit              // DON'T FORGET!!!!
            handshake_1.0/         // IP core from repository for part 2 with this exact
                                   // name. Make sure all VHDL is included
      part3/
            part3.bit              // DON'T FORGET!!!!
            fifo_1.0/              // IP core from repository for part 3 with this exact
                                   // name. Make sure all VHDL is included
```

Zip the entire directory and submit the UFID.zip file. Note that you do not need to include any software code. The provided code will be used to test your submissions.

COMMON PROBLEMS

- You may receive the following warning in ModelSim and/or Vivado: "Case choice must be a locally static expression". Allow in general you should avoid this warning, it is safe to ignore for this lab. The warning is caused by the use of a function when defining the constants used in the when statements. Most VHDL tools now ignore this issue.