Clock Domain Crossing

EEL 4930/5934 – 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.

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 designers 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 dual_flop directory) provides VHDL code for the Nallatech board that uses two domains. Dual_flop_h101.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 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, *even when both domains use the same frequency.* 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 (tb.vhd). Note that there are no errors.
2) Create a Dimetalk project with the basic PCI-X edge, clocks module, and memory map.
3) Import the provided VHDL and generate a bitfile. Dual_flop_h101.vhd is the top level.
4) Run the provided C code and 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.
5) Add a dual-flop synchronizer to the pulse signal and make any other necessary changes to the code.
6) Simulate using the provided testbench until there are no errors (you may have introduced some).
7) Repeat 2-4 to verify that your dual-flop synchronizer has fixed any metastability problems.

Turn in all vhdl files, your Dimetalk DT3 file, and the generated bitfile. There are no changes to the C code so you do not need to submit it.

Part 2 – Handshake synchronizers (only applicable to those working in groups)

Unfortunately, the dual-flop synchronizer cannot reliably be used to transfer multiple-bit signals. There are several reasons for this, but the main reason is because race conditions may cause each bit to arrive in the destination domain on different cycles.

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 (handshake_h101.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.

For part 2), do the following steps:

1) Simulate the VHDL with the provided testbench. Note that there are no errors.
2) Create a Dimetalk project with the basic PCI-X edge, clocks module, and memory map.
3) Import the provided VHDL and generate a bitfile. Handshake_h101 is the top level.
4) Run the provided C code and verify that the circuit does not work correctly. The C code will output the differences between the actual results received in the output block RAM and the correct results. The C code will iterate 1000 times to make sure there are no problems with the code. Note that if the code gets stuck on a particular test, it is because the output address generator is not receiving enough valid data, and is never asserting done. This data is being lost due to incorrect synchronization.
5) Fix the provided handshake entity.
6) Simulate using the provided testbench until there are no errors.
7) Repeat 2-4 to verify that your synchronizer has fixed any metastability problems.

Turn in all vhdl files, your Dimetalk DT3 file, and the generated bitfile. There are no changes to the C code so you do not need to submit it.

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 provide a different clock domain 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 in the fifo directory. Read the comments in the top-level fifo_h101.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 (invalid data is ignored later). 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). Read the Xilinx documentation to learn how to do this (hint: Project->New Source->IP (CORE Generator ….)). On the SelectIP dialog box, select "Memories and Storage Elements"->"FIFOs"->"FIFO Generator". For both FIFOs, you will need to use independent clocks, and you will need support for "First-Word Fall Through". Set the appropriate read and write width for each FIFO. The depth doesn't really matter here, so make it small, otherwise the other block RAMs will be forced to use distributed RAM, which will take much longer to synthesize. For the 32-bit FIFO, make sure to select the option for the almost_full flag (see comments in code for explanation).

After creating a core for each FIFO, ISE will generate a .ngc file for each core. Copy these files into the same directory as your VHDL, and make sure to select them as support files when importing into Dimetalk.

For part 3), do the following steps:

1) Create the FIFO cores in Core Generator.
2) Modify fifo32.vhd and fifo17.vhd to use the generated cores.
3) Simulate the VHDL with the provided testbench and fix any errors. If there are problems, you likely didn't select the correct options when creating the cores.
4) Create a Dimetalk project with the basic PCI-X edge, clocks module, and memory map.
5) Import the provided VHDL and generate a bitfile. Fifo_h101.vhd is the top level.
6) Run the provided C code (same as part2) to verify that the FIFOs work.

Turn in all vhdl files, ngc files, your Dimetalk DT3 file, and the generated bitfile. There are no changes to the C code so you do not need to submit it.

Final Instructions

Put each part of the lab in a separate directory and zip it. You only need one submission per group. Also, include a readme.txt that explains the status of the lab (e.g., unresolved problems, tool bugs, etc.).