ECE/CS 5960/6961 – Lab Assignment 4

1 Introduction

In this lab you will build a design using a pipeline controller that has been characterized for the relative timing flow. You will understand how the characterization information that you have worked on in the last lab and as discussed in class will be applied to drive the synthesis, place and route, and timing validation tools. You will also begin to experiment with performance and area tradeoffs in pipelined asynchronous designs and how the choices differ significantly from clocked design.

At a very high level, one can create asynchronous designs by taking a simple clocked architecture and replacing the `always @ (posedge clk)` Verilog statements with an asynchronous linear pipeline controller. This controller will clock the latch or flip-flop when data is presented on the input of the sequentials. A simple comparison of this straight forward replacement is shown in Fig. 1. In traditional clocked Verilog, the `always @` statements create the pipeline boundaries. If these are replaced with structural controllers and sequentials a very similar design results.

While there are many similarities, there are also significant differences between these two pipelines. The most salient differences to a designer, in terms of performance and area, will be presented as part of this lab.

The properties of any asynchronous design or system are dependent upon the protocols that are employed. For this lab you will be using a burst-mode controller where data is valid upon the rising edge of the request (an early narrow protocol). Fig. 2 shows the logical design of the controller. This controller has been characterized for inclusion into the clocked EDA design flow.

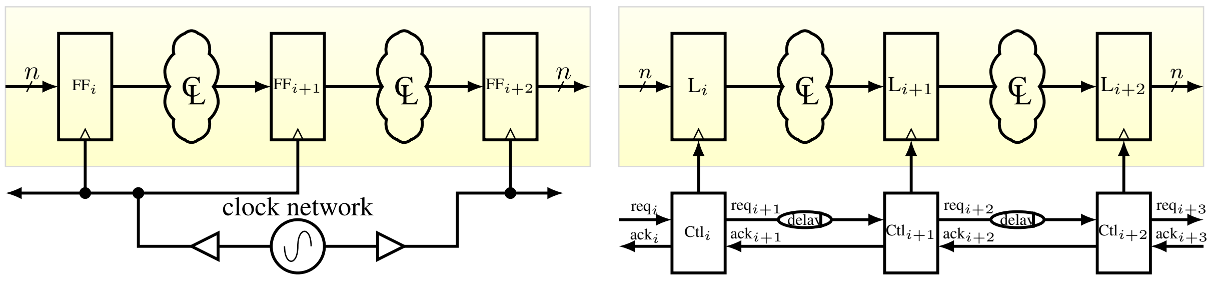

Figure 1: Simple clocked and asynchronous pipelines are easily emulated and translated between each other. The clock distribution network is replaced with handshake controllers which serve the same purpose. Flip-flops in this design structure can be replaced with latches.

2 Pipelines and Rings

The core of any concurrent design that you build will be a pipeline. Understanding pipeline performance in asynchronous systems is critical to building good pipelined architectures. To understand pipeline performance you need to understand the key metrics of a design. The metrics relate to the protocol that has been chosen for the implementation. Since the design that you will be using in this lab is a four phase design with data valid on the rising edge of the request, the performance metrics will be defined in terms of that protocol. Note that they will differ for other protocols.

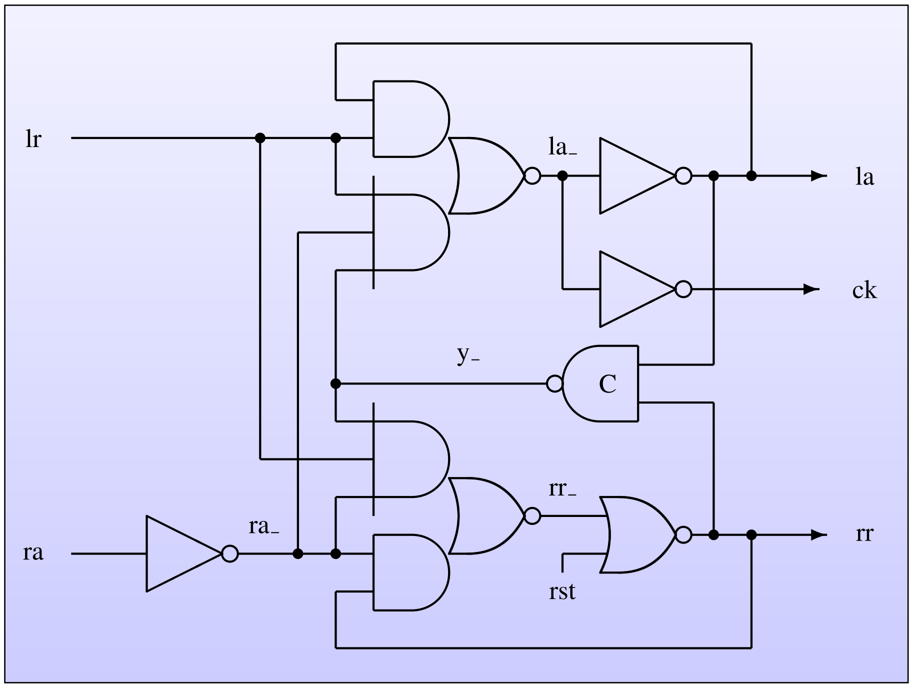

Figure 2: The four-phase asynchronous burst-mode controller employing an early narrow data protocol

.

The key performance metrics for asynchronous designs are as follows. They are the same metrics that the test bench provided to you in Lab 1.

1. **Forward Latency**. This is the delay from when an operation is requested until it has completed and is presented at the output. This is generally measured from the left request signal indicating data validity on the input to the right request at the output of the pipeline. The forward latency per stage is the total latency of the pipeline divided by the number of stages.

2. **Backward Latency**. This the dual of the forward latency. It is the time to propagate a "bubble" or empty pipeline space from the end of the pipeline to the front of the pipeline. This delay is measured as follows. An asynchronous pipeline is filled until it stalls by entering data tokens into the pipeline but not accepting any tokens at the output. At this point each register bank that can hold data will be occupied. Once the pipeline is fully stalled, the output channel is acknowledged. The backward latency is measured from the time data is accepted on the output of a stalled pipeline until a new token may be placed into the pipeline. This is typically measured as the time from when the right acknowledge is asserted until the left acknowledgment is asserted.

3. **Cycle Time**. The cycle time is the frequency at which data can be inserted into a pipeline. It is similar to the clock frequency of a design. We measure this as the maximum delay between two left request (or right request) signals in a pipeline that does not stall. The cycle time can be roughly approximated as the sum of the forward and backward latency.

4. **Throughput.** This metric measures the number of tokens that can be processed through a design over a period of time. This metric is based on forward latency, backward latency, cycle time, and connectivity of a design. The maximum throughput is limited by the cycle time. The maximum throughput can also be limited by forward and backward latency, depending on the architecture of the design.

2.1 Pipeline Stages versus Performance and Correctness

One significant difference between clocked and asynchronous designs are measured in how one creates a concurrent design.

In a clocked design, the location and number of pipeline stages defines the *performance and correctness* of a design. The simple modification of adding a pipeline stage to a clocked design usually changes the functionality of the design so that is becomes incorrect. This is because timing relationships between data elements are measured in terms of the number of clock cycles between events. If this changes, intended data interactions will not occur on the same clock cycle and the function will fail.

In an asynchronous design, the location and number of pipeline stages defines the *performance* of a design. The correctness is not a product of the number of pipeline stages on design paths. The interaction between data is performed based on data validity and handshaking. Thus an arbitrary number of pipeline stages can be placed between any two events that must interact. Until the data validity signals arrive for all necessary data items, the pipeline will stall and wait. Thus one can increase or decrease the number of pipeline stages of an asynchronous design and it will still operate correct.

In this lab, you will experiment with adding or removing pipeline stages in a design in order to create a design of the smallest area and highest performance.

2.2 Canopy Graphs and Asynchronous Pipeline Performance

The design that we will make is a ring. The architecture can be viewed as a linear pipeline where the head and tail are connected and communicate. The ring will be initialized with a number of valid data tokens. The size of the ring will, as you will see, determine the performance of the design.

Assume the simple ring shown in Fig. 3. This design consist of four pipeline controllers (with the data path abstracted out). The black tokens indicate data occupancy. Each stage sends requests down the pipeline to the controller on the right until the end of the pipeline is reached. The last stage sends its request to the first stage, forming a ring.

Assume that the ring is initialized with tokens in each of the four stages of the design. What is the throughput of the ring? Since the ring is initialized with four data tokens, then the ring has a throughput of zero, as there is no place for the data tokens to move. The dual condition also has a throughput of zero: a ring that is initialized without any data tokens. If the ring is initialized with three tokens then the performance is limited by the *backward latency* because the speed of the empty space (bubble) will dictate the throughput of the ring. The dual condition is that of having a single token in the ring. The throughput will be limited by the speed at which the token can propagate forward through the ring. Therefore the performance of this ring initialization will be limited by the *forward latency*. The example shown in Fig. 3 has two tokens in the ring. Every

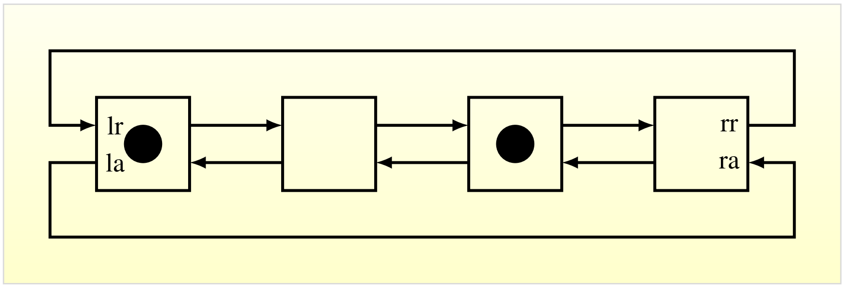

Figure 3: Simple 4-deep ring design (data abstracted out) initialized with two data tokens (circles).

token has an empty pipeline stage downstream. Thus this design will be limited by the cycle time of the design (assuming there is no wire delay and all controllers are identical).

The relationship that forward latency, backward latency, and cycle time have on throughput is shown with *canopy graphs*. Fig. 4 shows canopy graphs for several different clocked and asynchronous FIFO structures. (The clocked FIFO structures can stall, allowing them to mimic asynchronous design and various FIFO occupancy.) The asynchronous designs in this figure use the same controller of Fig. 2 that we will use in this lab. The design in Fig. 4 of most interest to us for this lab is the linear design (that uses circles). Notice the throughput is highly dependent on the occupancy. The forward and backward latency dominate the performance of this design. The maximum throughput is achieved at an occupancy of three (three of the ten pipeline stages are occupied). In this design, the forward latency is smaller than the backward latency. Therefore, maximal throughput is reached with fewer valid tokens in the pipeline.

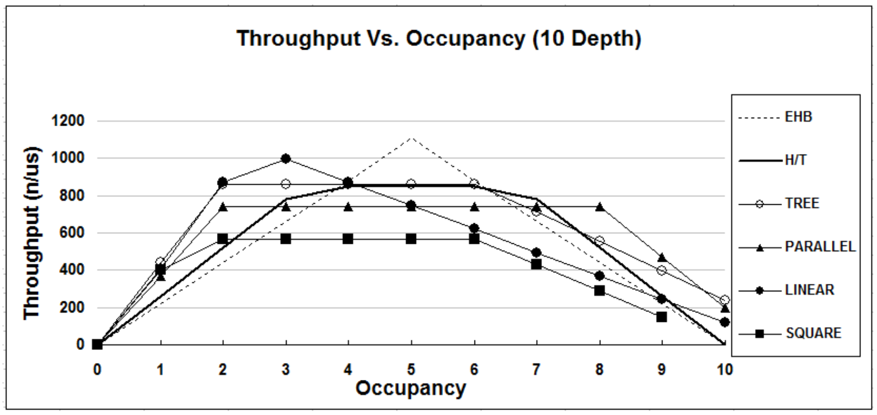

Figure 4: Throughput versus occupancy comparison for FIFO capacities of 10 tokens

Fig. 5 shows the same FIFO architectures, with much deeper designs which hold 50 tokens. The maximum frequency of the pipeline controller we are using in the given technology is 900MHz. Notice that for the linear pipeline, saturation is reached at around 900M tokens per second. Thus the cycle time of the controller dictates the maximum throughput of the design.

Notice that if one designs a ring, and the ring has a constant number of valid tokens in the ring, it will operate at a fixed occupancy. Given forward latency, backward latency, and cycle time one can create a canopy graph for the design. This will tell you what the throughput will be for any occupancy. If a 10-deep linear ring is employed with latencies and cycle times in Fig. 4 one

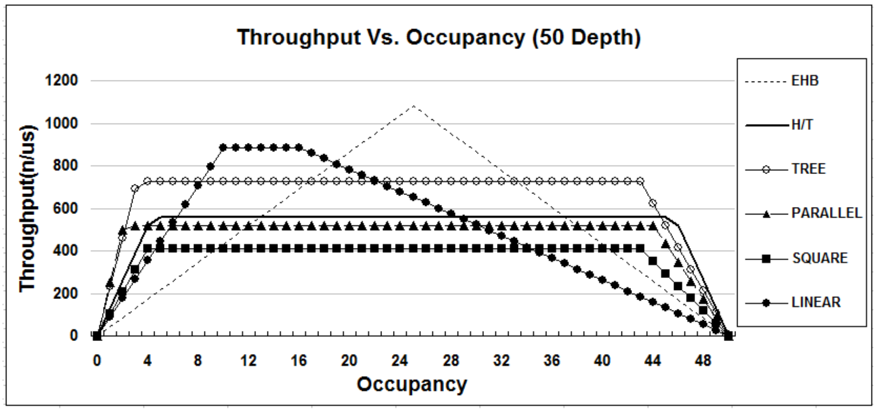

Figure 5: Throughput versus occupancy comparison for FIFO capacities of 10 tokens

can determine throughput at any occupancy. A ring with one token will operate at a throughput of approximately 425M tokens per second. By placing two tokens in the design, the performance will double to 850M tokens per second. So by adding pipelining in the design, the performance of the design increases. If too many tokens are added then the performance decreases until the pipeline completely stalls.

You can calculate the number of pipeline stages necessary for a ring to reach maximum performance beyond a fixed number of tokens in the ring. Assume you are building a ring with three tokens. First create a ring with three tokens and reset them so they are occupied. Then add a pipeline stage to the design and evaluate its performance. Additional stages can be added until performance saturates and then decreases. Based on the canopy graph in Fig. 4, once ten pipeline stages are added, the performance will no longer improve. Thus a design of ten stages with three occupied give the best performance at the smallest area.

3 Project Definition

For this project you will design the message scheduler for the SHA 256 encryption algorithm. For this design you will initialize a ring with 16 32-bit data tokens. The design will output 64 32-bit words. The first 16 words produced are the initial values of the ring. The next 48 words are a permutation of the initial 16 words. Output data stream W_j is produced from input stream M_j where:

```
For $j = 0, 1, \ldots, 15$
    $W_j \leftarrow M_j$
For $j = 16, 17, \ldots, 63$
    $W_j \leftarrow \sigma_1(W_{j-2}) + W_{j-7} + \sigma_0(W_{j-15}) + W_{j-16}$
where
    $\sigma_0(x) = S^7(x) \bigoplus S^{18}(x) \bigoplus R^3(x)$
    $\sigma_1(x) = S^{17}(x) \bigoplus S^{19}(x) \bigoplus R^{10}(x)$
```

The functions S^n applies a right shift by n bits, R^n applies a right rotation by n bits, and $\bigoplus$

is a bitwise XOR. Fig. 6 shows the general design of this message scheduler. This looks just like a 16-deep ring with an add/XOR function that creates new values for the last 48 permutations. Note that this ring structure changes the performance numbers when compared to a simple FIFO. The permutation function is relatively complex with a significant delay. This results in one stage having a very large delay compared to all the rest. Think about how these delays will require different pipeline stages to be inserted into your design.

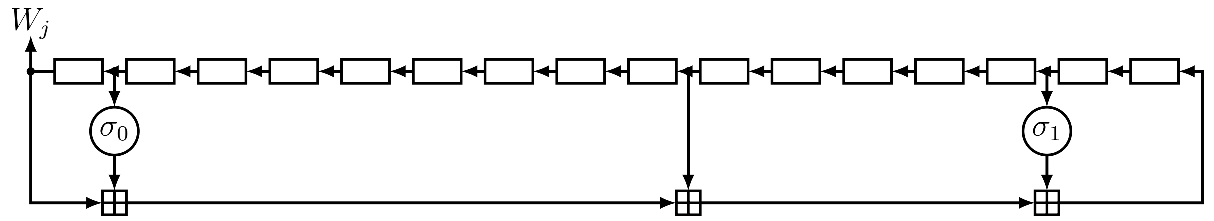

Figure 6: The message schedule function represented as a block diagram.

For the deliverables of this lab, you will create a ring where the memory elements are initialized with a set of data consisting of 16 32-bit words. Fig. 7 shows the top level design interface. A left channel request will present 512 bits of data that are loaded into the registers. 64 output requests follow with the message schedule results. After the final output has been accepted, the input channel is acknowledged.

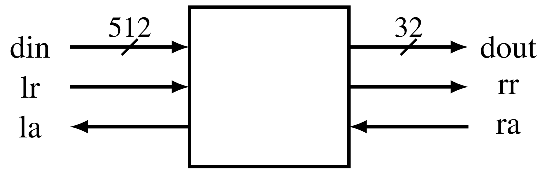

Figure 7: The top level design for this lab.

3.1 Design Notes

From Fig. 6 one can see that the values from four of the pipeline stages all converge to create a single result that is placed into the right side of the ring. Note that control flow in any asynchronous design must follow data flow and indicate its validity. Thus the stage on the far right of the ring will not proceed until inputs from four other pipeline stages, and their handshake control signals, arrive at that stage.

A simple circuit is commonly used to fork and join handshake control signals. This simple circuit is shown in Fig. 8. It consists of a single C-Element. In the fork configuration, the input (left) request is broadcast to the two output (right) channels. When the acknowledgment from both output channels is received, the input channel acknowledgment is generated. The C-Element ensures that the input channel acknowledgment is not forwarded until an even occurs on both of the output channel acknowledgments, `ra0` and `ra1`. The join element is the mirror of the fork element. When both input channel requests arrive, the output channel request is generated. The output channel acknowledgment is broadcast to both input channels. Since the join element is the mirror of the fork element, a single design block suffices.

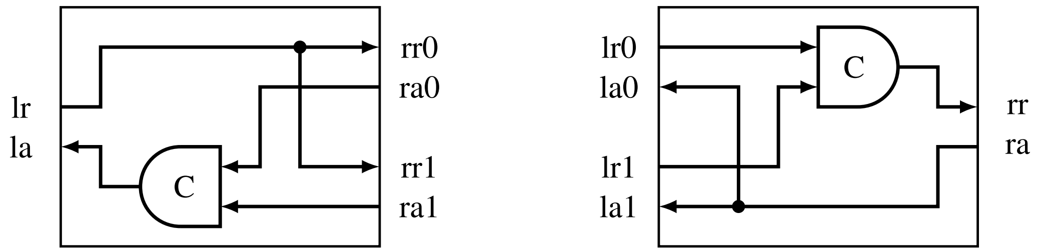

Figure 8: 2-way Fork element (left) and two-way join element (right)

Each of the locations where data forks and joins in your design will use fork element(s) to ensure that data is valid on all paths. Note that a four way join can be built using a two level logarithmic tree of three fork elements.

The most challenging and fun part of an asynchronous system is the design of the control path. Often your time is spent more productively by working on the control design and getting it correct before adding the data path. I suggest you first implement the control structure that will mirror the data path of your design, and simulate that to ensure that it operates correctly. You may even want to start with a simple ring such as that in Fig. 3, and later and the forks and joins that are necessary to implement the data sharing that is part of the message scheduler specification. Once you have a design, add pipeline stages and see what it does to the performance. You can even implement the first design without adding timing to your design. Then you can add the behavioral data path and add the timing to match the delay values.

3.2 Project Design Flow

The design flow for this lab proceeds as follows:

1. Create or modify a behavioral Verilog description of your design.

2. Run the `ArtCOM` tool on your design that will create a constraint file that maps the relative timing constraints for each characterized module onto the architecture of your design.

3. Go over the constraint file and update the timing constraints to reasonable values for your design. Most of the pipeline stages will have the same delay, but the pipeline stage with the adders will have a significantly larger delay. Remember that the area of slow adders is significantly less than that of high performance adders. Pick a power/area tradeoff point that you are targeting, and note that in the README file.

4. Run synthesis on the design, and iterate until there are no errors or timing violations in the design.

5. Make pipeline modifications on your design to reduce the area and improve the performance. Once you are satisfied, move onto the physical design.

6. Simulate the design to ensure that it behaves correctly and matches the results of the golden model.

7. Perform physical layout on the design.

8. Run PrimeTime on your design and observe the timing violations. Some negative slacks are okay for this design. Large negative slacks will need to be fixed by changing the timing targets and running synthesis and place and route again.

9. Simulate the design to ensure that the results meet the golden model.

4 Design Modules and Tools

The design IP provided will be briefly described here, as well as the CAD tools that will help you complete the design correctly.

4.1 Design IP

The pipelined core of this design will be built with the linear controller of Fig. 2. A characterized Verilog module will be provided. You will also be provided FORK and JOIN modules for the design, as shown in Fig. 8. However, you are welcome to design your own fork and join modules from any of the C-element designs you built in Lab 1.

Following is the list of IP provided:

1. **C300R3044** Pipeline controller that resets empty.

2. **C300R3044r1** Pipeline controller that resets full.

3. **bcast_fork** Broadcast fork/join element. Feel free to roll your own.

4. **bcast_fork_r0** This is a broadcast fork/join element that resets to zero when one input ji0 is asserted on reset.

5. **c_element_nand** The C-element built from four NAND gates.

6. **c_element_nand_r0a** The 4 NAND C-element that resets to zero when input a is high upon reset.

7. **latch32** The 32-bit latch to use for this design. It is closed (opaque) when the clock is high.

8. **go64** This is a controller with two input channels and one output channel. This design will perform a handshake that will enable a set of events on the other channel. More detail will be included below.

9. **count64** A 64-element counter that asserts done after 64 events.

10. **sigma0** The SHA-256 σ_0 function.

11. **sigma1** The SHA-256 σ_1 function.

4.1.1 Parallel Load and 64 Iterations

The design spec requires you to perform parallel loading of the 16 registers, and then shift data out 64 times. The 17th through 64th data items will be permutations on the first 16 data items based on the σ functions.

While you are free to build the design however you like, following is a basic approach that you can use. You will start with a ring of 16 latches. Add in fork and join elements where the data from the stage goes to multiple destinations and comes from multiple destinations. This gives you the basic control flow where a handshake indicates data validity on every data path.

Observe that you minimally will require a single empty token following every fork or join element in the design. Otherwise the cycles will have no empty spaces to send the data. Add in those empty stages.

You now have a ring that will come out of initialization and iterate forever. The go64 element will allow you to parallel load data into the latches, kick of a number of iterations determined by a counter, and terminate the iterations when done. The block count64 will assert done when 64 iterations are completed. To test your design with smaller data sets, take the done output from an earlier bit in the sequence.

One channel of the go64 block will be placed in series with a request and acknowledge channel as shown in Fig. 9. The lri, lai and the rr,ra channels constitute the channel that will iterate based on a counter.

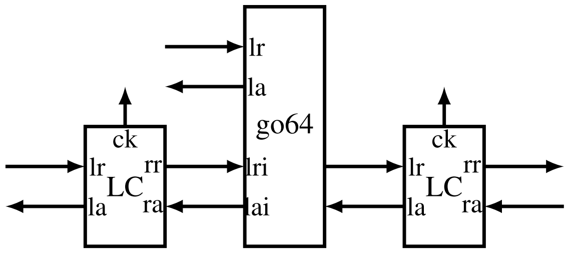

Figure 9: The go64 module showing how the internal channel connects into a pipeline to stall the pipeline, but then allow it to run for a fixed number of iterations when a request arrives on the other channel.

This controller can be placed anywhere in the design, as long as it stalls the rings from operating until it receives a request. Possibly the best place to put it is in the ring where the data forks to the output and feeds back onto the ring. I placed this block to the left of the left-most pipeline stage in Fig. 6. Following that stage, an fork is inserted to send data to the output (W_j) and to feed data back through the adders to the right-most latch. This is a good place to put it because it doesn't allow the design's right request to be asserted until the left request occurs and the input data is loaded into the design.

4.2 Latches and Clocking

You will need to create some sort of muxing mechanism to select between parallel load data and the input data from the ring. This implies that there are *two* clocks that are needed for every latch.

The C300R3044 controllers are designed with normally open latching. However, when the controllers are initialized with data, the clocks are closed (opaque), storing the data. One nice trick to initialize the data is to place a piece of logic that will force the clock signal low to the latches upon reset, and then once a request arrives at the go64 module allow the clock to assert to store data. I did this in my design with the gate in Fig. 10 (which is also used in the go64 module). You will need to place this clock control logic everywhere that the data needs to be stored upon initialization. This will at least be in every controller that is initialized with data (those latches controlled by the C300R3044r1 module). This clock control circuit may also be necessary for other controllers.

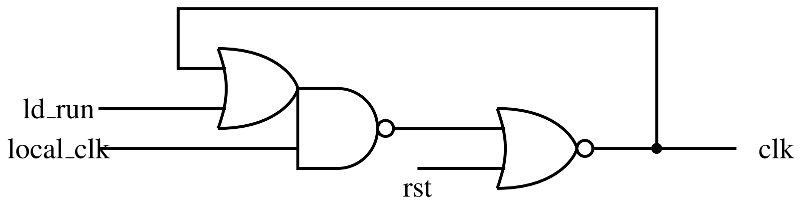

Figure 10: Clock logic that will store data on moving into run mode, and not glitch the clock when moving out of run-mode. Note that ld_run can assert at any time, but must not de-assert when local_clk is changing. If ld_run de-asserts when local_clk is high, the feedback will keep the clk output asserted until local_clk de-asserts.

I used the la signal coming from the go64 module as the ld_run signal to these clock controllers. It asserts upon receiving a request to start the operation, and remains high until all output transfers have occurred. I also delayed the la signal and used that to select between the parallel load data going into the latches that are initialized with data and the input data coming from the adjacent cells in the ring.

4.3 Timing Constraints

You will need to include size only and cycle cutting constraints for the sequential modules used in your design. If you use the modules that I have included in the ring.v file in your design, these constraints are in the ring.cstr.tcl file. They are commented out, so you will need to remove the comments before the appropriate rows of the constraints for the modules that you will use in your design.

Also note that I have not included all of the timing necessary to guarantee correctness for the clock control logic and for the go64 module. This should work for this simulation, but if not I will provide constraints for these modules.

4.3.1 Simulation

I included a second example simulation file testbench2.v that you can use to get an idea for writing a testbench. A golden input stream (Veriloginputs) is provided that you will use to verify your design against the golden result (Goldenoutput). This testbench writes the output stream into the file Verilogoutputs.

For this older cell library, the new Synopsys tools write out a version of the .sdf file that contains constraints that are unsupported by the liberty file. I have put a script in the async bin directory that will fix that called `sdf-postlayout-conditioning-ibm`. If your design file is called ring, run the following command to remove the unsupported constraints to allow back annotation of the delays into ModelSim.

```
sdf-postlayout-conditioning-ibm ring.dcopt.sdf
```

4.3.2 Reset and Initialization

Note that the design as implemented will only allow one transaction. The counter and the sequential storage used for the clock control is only initialized when reset is asserted. In the current design, reset must be asserted in order to load a second word into the design. For extra credit, modify the circuit to allow new transactions to occur without having to reset this logic block in between the transactions.

4.4 Design Tools

Following is a brief description of the design tools you will use to build the design:

4.4.1 Automatic Relative Timing Constraint Mapping

There is a tool named `constr-mapping` installed in the async tool directory on CADE. This tool performs automatic constraint mapping on your design. It loads in your design, reads the relative timing characterization information, and produces a set of mapped constraints between controller modules and the memory elements of your design.

Follow the following recipe for setting up the constraint mapper:

1. Create a directory named `data` under the current working directory where you will run the constraint mapper.

2. Copy all of the files from `/uusoc/facility/cad_tools/Async/lib/constr-mapping` into your `data` directory. These files contain scripts, log format information, and the design templates you will use for your design.

3. Connect to the `data` directory and create symbolic links in that directory to your design. For instance, for the ring design on the web page, if it is in the parent directory, you would type:
   ```
   ln -s ../ring.v .
   ln -s ../ring.cstr.tcl .
   ```

You are then ready to run the constraint mapper. Do so by typing `constr-mapping` in the directory where you created the `data` directory.

This program has *very* chatty output. Best plan is to look in the ARTCoM.out file for errors and warnings. Make sure that you understand and can justify all of the errors and warnings in this output log.

The result of the program is to generate the file `data/mapped_output.txt`. This file contains all of the relative timing constraints necessary for your design to operate correctly. Look over this file, and search for the dollar ($) sign. These are either tcl variables which define delays, or constraints that were not fully mapped in the `constr-mapping` program. You will need to comment out unmapped constraints, and make sure that any tcl delay variables are defined. For the templates that we use, the variables `dpdelay` and `dpdelay2` must be defined. (You can do this with `set dpdelay 0.500` to define a 500ps delay.)

Check to make sure the constraint mapping program created the correct timing constraints for your design, and then copy them into your DESIGN.cstr.tcl file. (If you manually created constraints, as I did in the example file, you can replace them with the results in this file.)

Hints: In the final ring design, you will have minimal delays between most pipeline controllers. There should only be four data-paths that do not use the smallest handshake delay. For the four, set a delay that makes sense for the amount of logic between the controller and its destination register. For the other paths, you can remove the min-delay constraint and only use a max delay.

For example, in the hand generated relative timing constraints provided in ring.cstr.tcl, you will see that I have a minimal delay for the forward propagation of lr from stage c2 to c3, but comment out the min delay constraint.

```
set_max_delay 0.300 -from c2/C300R30442/A1 -to c3/C300R3044r10/A1
set_max_delay 0.300 -from c2/C300R30442/A1 -to c3/C300R3044r12/A1
#set_min_delay $dpdelay  -from c2/C300R30442/A1 -to c3/C300R3044r10/A1
#set_min_delay $dpdelay  -from c2/C300R30442/A1 -to c3/C300R3044r12/A1
```

4.4.2 Timing Evaluation

Selecting the best timing constraints for a design is helped by giving delay targets, and then observing the timing of the resulting circuit. If slacks are negative or tight, the delay targets can be increased. When too much margin has been provided, the delay values can be decreased. Eventually you will converge on a set of timing constraints that work well for your design.

A PrimeTime script is provided to help you evaluation timing. Look at the log files `DESIGN.dcopt.paths`, `DESIGN.dcopt.paths.min`, `DESIGN.dcopt.constraints`, `DESIGN.dcopt.fullpaths`, and `DESIGN.dcopt.fullpaths.min` to see if your design met timing and if not which paths didn't meet timing. You can look at specific individual paths and their timing by running PrimeTime directly. Do this by running `syn-pt`. Inside the PrimeTime shell, load the provided script with the `source primetime.tcl` command. You can then open the DESIGN.cstr.tcl file and observe the timing path for any of the relative timing constraints by typing `report_timing`, and cut and past the `-from XXX -to YYY` commands from the cstr file. If you are checking a min-delay constraint, append `-delay min` at the end of the command. This will allow you to determine if you need to change any of the timing targets in your design to meet timing.

HINTS: You will need to modify most if not all of the .tcl and .csh scripts if you change the top module DESIGN name from `ring` to another name.

4.4.3 Timing Closure

There is a tool called `timing-closure` that will create a design that has no violated timing constraints. The tool will run synthesis and if there is a timing violation, it will modify your `DESIGN.cstr.tcl` file in a manner that should eventually allow a solution to the design. The

`#margin` comments in the `DESIGN.cstr.tcl` file created by the constraint mapper are used to ensure that the min and max delay paths of relative timing constraints do not violate the specified margin. These commands are comments to the regular EDA tools, and only used by the timing closure tool. If you want to use this tool to perform timing closure and you have a standard scripts used in the class, you should be able to run the program with the command `timing-closure -base=DESIGN`.

See the file `DESIGN.tc.log` for the results and a log of the constraints that were modified.

The timing-closure tool is not currently configured to work with the CADE mysql server at this time. Check back later to see if the tool has been properly configured to communicate with the UofU sql services.

5 Deliverables

Include a README file that contains your description of the lab, what you learned and any problems that you encountered. Also describe all of the files that are turned in with a brief description of what they contain. Please turn in only the files listed below, not a tar dump of the whole directory.

Include the files and results in a .tgz file that has the following. (Note: DESIGN in the following deliverables will be replaced by your design name, such as "msgscheduler".)

1. Your README file which contains:
   (a) Comments and feedback about the lab.
   (b) The area and performance of your design.
      i. Area report should be the "Total cell area" from the DESIGN.dcopt.area file.
      ii. For performance, report the total testbench run time and maximum cycle time from your simulations.
   (c) Note the number of pipeline stages, beyond 16, that you added to the design to reach a good performance. Describe how you picked that number.

2. Include the following files:
   - Design, synthesis, and timing constraint files:
     (a) The behavioral Verilog DESIGN.v synthesized DESIGN.dcopt.v files.
     (b) The DESIGN.dcopt.out log file.
     (c) Your DESIGN.cstr.tcl file that contains timing constraints.
     (d) The DESIGN.dcopt.constraints, DESIGN.dcopt.paths, and DESIGN.dcopt.paths.min files.
   - Files from the simulation directory:
     (a) transcript and Verilogoutputs
     (b) Your testbench.v and testbench.tcl files.
     (c) The Verilogoutputs file containing the output stream when run with the Veriloginputs data.
     (d) A screen shot of the ModelSim simulation waveforms.
   - If you used `constr-mapping` rather than generating the RT constraint files by hand, include the constrmap.log and mapped_output.txt files.