ECE/CS 6770 – Lab Assignment 5

Due 21 March 2017 via Canvas

1 Introduction

All previous labs have employed clocked design methodologies. In this lab you will get a brief introduction to self-timed design. Asynchronous design arguably has several potential advantages over clocked design styles, including lower power and higher performance. One irrefutable advantage is the modularity and compositionality of asynchronous design. This is due to the formal handshaking protocols that are used to control data transfers and frequency of operation between two adjacent blocks. In this lab you will compose some simple linear pipelines using two different asynchronous handshake controllers. These controllers operate with different communication protocols.

2 Linear Asynchronous Pipelines

Figure 1 shows a two deep asynchronous pipeline. A *communication channel* in an asynchronous design consists of control information and data. The control information of a communication channel identify validity of the data, and synchronize data transactions between two or more hierarchical design elements.

This figure shows a *bundled data* asynchronous pipeline where data consists of n boolean data bits, and the control information is encoded with a request (req) and acknowledge (ack) signal. The behavior of this channel is such that the data becomes valid to the input of the latch before the rising edge of the req signal. A positive transition on the ack signal indicates that data has been stored in the latch and that the sender is free to modify the data. For a *4-phase* channel protocol the control signals return to the idle state by first lowering the req signal, and in response lowering the ack signal. Thus the req and ack signals alternate back and forth with the req signal leading the transitions. The data is always valid on the rising edge of request. This is the **channel protocol** that is used for the controllers in this lab.

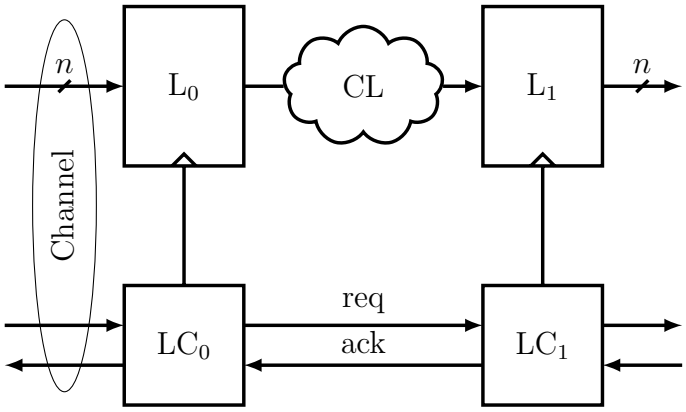

Figure 1: An asynchronous linear pipeline, denoting communication channel

Note that the pipeline storage elements in Figure 1 are composed of latches (L) rather than flip-flops as would be the case in a traditional clocked design. This results in nearly half the sequential area in the design of an asynchronous pipeline when compared to that of a similar clocked design. Also note that the sequentials are not clocked with the global clock, but rather they are controlled by a *linear pipeline controller* (LC). These controllers generate the clock signal to the latches and also synchronize the upstream and downstream channels.

The specific protocol used to synchronize the upstream and downstream channels determines the behavior, and the logic implementation, of the LC blocks. This simple channel protocol where req and ack toggle back and forth results in 137 different *speed-independent*[1] protocols, when one abstracts out the various methods of how the controllers can generate the clock signal to the latches or flip-flops! If timed protocols are employed this number grows to over 500 choices, again only considering the req and ack control signal behavior.

The protocol effects a number of properties of an asynchronous pipeline such as the buffering capacity, the area and power, forward and backward latency, and cycle time (or nominal design frequency). In particular, note that in a clocked design, only one data is stored for every pair of latches (a flop consists of two latches per flop). In an asynchronous design, some protocols allow data to be stored in every latch, resulting in double the pipelined storage density of an asynchronous design compared to a clocked design.

As soon as a request arrives at the input, an asynchronous pipeline will immediately begin processing the data. Thus, asynchronous designs are called *reactive*. If no data is presented to the design, there will be zero active power dissipation. This is unlike a clocked design, where the clock must continuously run. A continuously running clock is necessary in clocked designs because there are no explicit control signals as part of its communication channels. Therefore the clocked design must continuously sample data to determine if there is a valid operation that must be performed. Even with clock gating, energy is expended to distribute the clock across the chip and to block it from driving the clock pins on the flip-flops. The ratio of energy required to distribute the clock and drive the latches versus energy to distribute the clock and gate the clock at the local nodes determines the efficiency of clock gating. However, it does not take many idle clock gated cycles, even in a clock gated design, for an asynchronous design to demonstrate significant power savings over a comparable clocked pipeline.

The communication channel and the associated controllers that connect the channel in an asynchronous design implement a controllable ring oscillator. Thus each communication channel in an asynchronous design has a native frequency of operation based on the delay through the ring oscillator. For the communication channel to be implemented correctly, the cycle time of the ring oscillator implemented in the control signals must be greater than the latency through the data path. Therefore, the frequency of operation of an asynchronous pipeline is determined by the maximum of the delays in the LC blocks and the combinational delay of the data pipeline.

In this lab we are using a channel protocol where the rising of the req signal in the control channel indicates a new valid data is ready on the input of the downstream end of

[1]The speed-independent, or SI, timing assumption is common to asynchronous design. It assumes that the logic elements of a design can take arbitrary delay and that the communication elements have zero delay.

the channel. This means that, in Figure 1, data moving from L_0 must arrive at L_1 before the req signal from LC_0 arrives at LC_1. Note that due to the channel protocol, data moving from L_0 must arrive at L_1 before the req signal from LC_0 arrives at LC_1. If the LC controllers are small and fast, and there is substantial combinational logic in the data path, the req signal must be delayed between the LC blocks.

3 LC Protocols

One of the goals of this lab is to demonstrate the ease of composing asynchronous systems together, even when they operate at different frequencies. The *modularity* advantage of asynchronous designs is thus demonstrated. By simply connecting the output channel of one asynchronous block to the input channel of another, a correct design will be created. This will be shown to be the case even if the linear controllers implement different protocols! (Note that this is only guaranteed to hold if the protocols consist of the same protocol family.)

As we have mentioned in the lectures, the design of asynchronous *systems* is simplified due to the modularity of asynchronous handshake protocols. This results in my claim that the hard problem of design – system level design – can be simplified with asynchronous design. However, the design of the individual modules can be *very* difficult.

Not only is controller design challenging, the search space is large and varied. We will discuss the choice space that exists for the implementation of protocols in a simple asynchronous protocol family.

3.1 Concurrency Reduction of an Asynchronous Protocol

You will be provided two different LC controllers. Both of these designs are members of a single protocol family. Each member of a protocol family can be placed in a lattice based on the amount of concurrency for each design, and each member can be derived using concurrency reduction from the protocol with maximum concurrency. Each member of a protocol family has a specific protocol that it implements, and its behavior can be formally verified to conform to that protocol. The amount of concurrency reduction will allow one to compare the protocols in a family. In general, as concurrency reduction is performed on a protocol family, the circuit design of members with more concurrency reduction becomes simpler because the state space and richness of behavior is reduced. This often results in faster response times through the controllers. However, there is also a reduction in protocol concurrency. Certain classes of protocol concurrency can be removed without reducing the overall performance of a design. This results in a faster overall design. However, if too much concurrency is removed from a design, a performance penalty will occur.

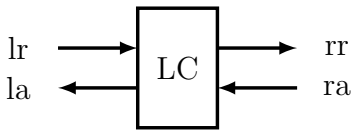

Figure 2: Linear pipeline controller (LC) with control signals for the left (upstream) and right (downstream) channels

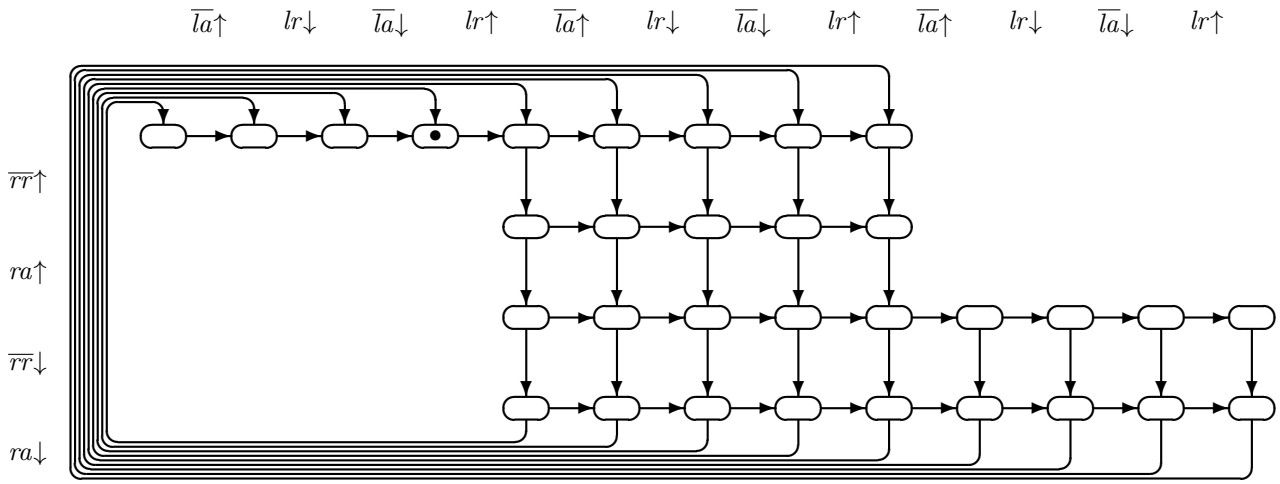

Figure 3: State graph for the protocol of maximumal concurrency

We will abstract out the datapath to simplify this lab, and to clarify the differences in concurrency between pipeline controllers. Figure 2 shows how a linear pipeline controller interfaces between two channels. The incoming channel is on the left, so the left request is labeled lr and the left acknowledgment is labeled la. Likewise the downstream or right channel control signals are labeled rr and ra.

Figure 3 shows the minimized graph for the protocol with maximum concurrency possible for the asynchronous protocol family described in the first part of Section 2. The state with the bullet is the starting state. This protocol family has 32 different states based on possible concurrency between the left and right channels that maintain correct behavior of the two independent channels (including the relationship with the datapath which is abstracted out of this protocol).

In order to simplify the representation of concurrency and concurrency reduction, the state graph can be drawn as a *shape*. This simplifies the representation by removing the arcs showing the transitions between the states and the transition labelings. The format remains the same as in Figure 3. The shape for the protocol of maximal concurrency is shown in Figure 4. The blue ball represents the start state. Concurrency reduction will be performed by removing states from the shape. A set of rules are applied that maintain properties for correctly reducing concurrency in an asynchronous protocol. Applying concurrency reduction according to these rules allows us to generate all valid protocols for this circuit family. Thus we can compare the performance, power, latency, area, and other properties of two different protocols in a family.

3.2 Specification of LC Controllers Used in Lab

You will use two different protocols in the design of this lab. The first protocol is based on the Sutherland Micropipeline circuit. This design employs substantial concurrency reduction, removing 16 states from the most concurrent specification, resulting in a protocol with 16

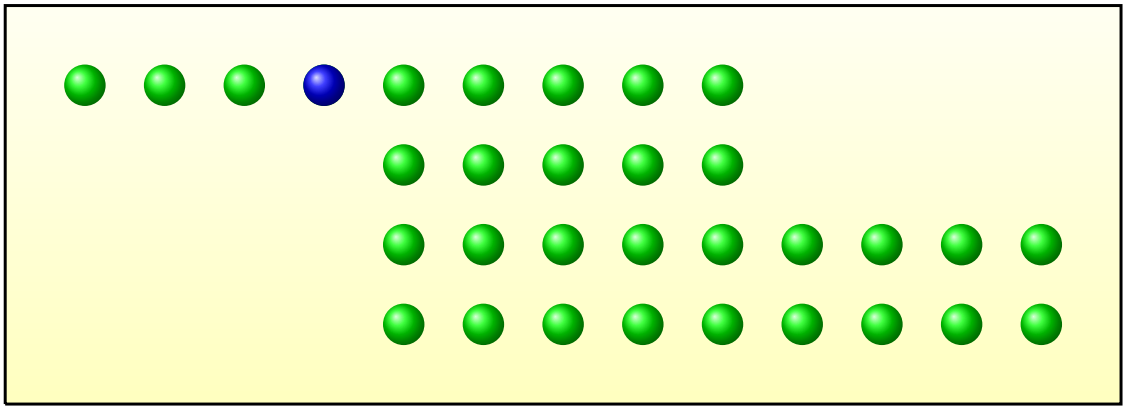

Figure 4: Shape of protocol with maximum concurrency

states. Its shape is shown in Figure 5. The symmetric nature of the concurrency reduction of this design results in the specification of a C-element. We have discussed various designs for this circuit in class, including several dynamic as well as static implementations. You will be provided the structural design using combinational gates in the Artisan library for a C-element.

The second controller is a timed protocol. It assumes that the delays internal to the controller are less than the delays required by the environment to respond to outputs of the controller. This implements a protocol based on a *burst-mode* state machine specification. The specification for the design after applying concurrency reduction is the shape shown in Figure 6. An implementation of this circuit mapped to combinational gates will also be provided to you as part of the lab in a structural Verilog format.

Sequential circuit controller designs must be protected from modification through the EDA tool flows. Clocked EDA tools can only reliably synthesize combinational logic. Hazards and unintentional behavior can occur when "optimizing" sequential designs built from basic logic gates through clocked synthesis algorithms such as those employed in Design Compiler. Therefore, each asynchronous controller must be represented structurally and protected from logical modification by the EDA tools, or hazards may be introduced.

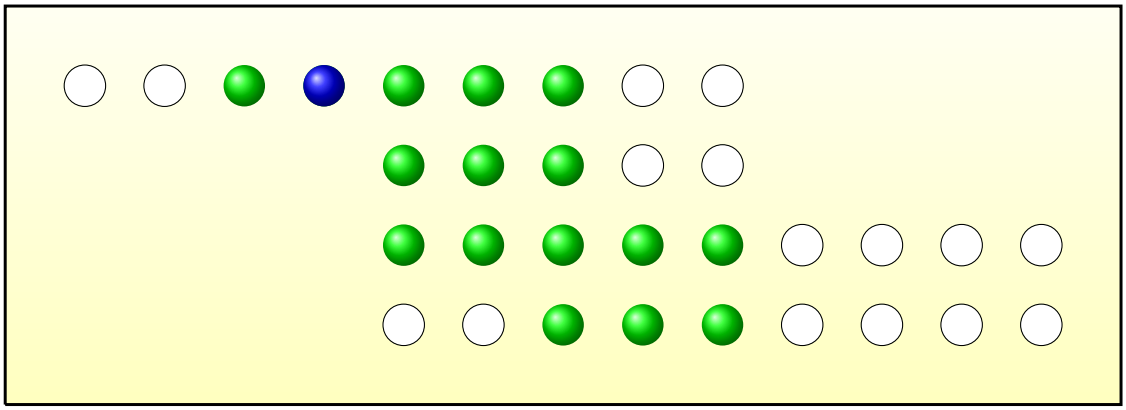

Figure 5: The shape for the Sutherland Micropipeline

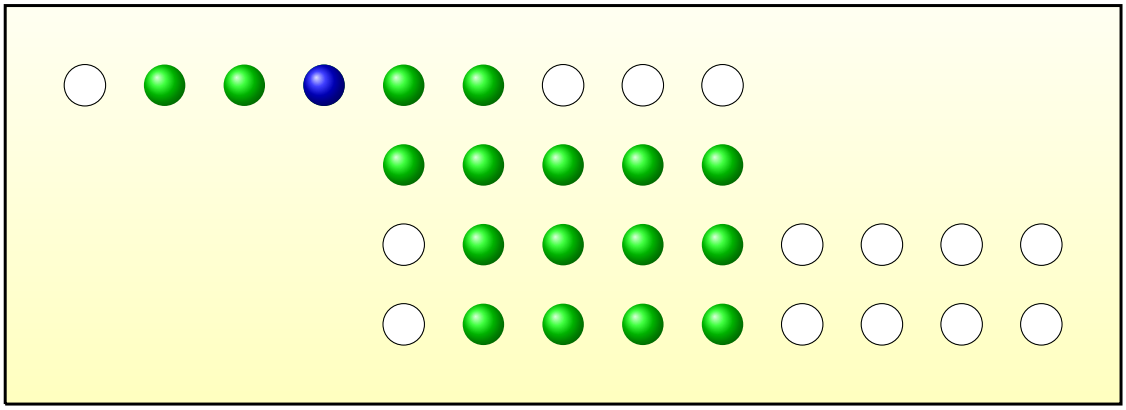

Figure 6: The shape for a Burst-Mode Controller

Note that one of these controllers consist of 16 states and the other 18 states. There are both similarities and differences in the states that have been removed from these two controllers. You will make observations on your design based on the behavior of these two controllers. Their different protocol behaviors will result in substantially different performance and pipelining properties.

4 Project Definition

You will be provided the following components for this lab.

1. The structural design of a burst-mode controller.
2. The structural design of a C-element.
3. Two blocks that control token production and consumption in a linear pipeline.
4. A delay block that allows control signals to match combinational pipeline delays.
5. A simulation test bench written in TCL for execution in ModelSim.

4.1 Pipeline Design

You will build and analyze six different linear pipelines using the blocks provided. Each pipeline will use one or both of the linear pipeline controllers, and two interface controllers.

The leftmost channel of each 4-deep pipeline will be connected to a LHS (Left Hand Side) block, which controls the input data stream behavior. The LHS controller will reset the pipeline and produce a stream of tokens into the pipeline. The behavior of the LHS block is controlled by the simulation test bench and its interaction with its downstream handshake channel.

The rightmost channel of the 4-deep pipeline will be connected to a RHS (Right Hand Side) block, which controls the output data stream behavior. The RHS controller will consume tokens that propagate through the pipeline. This block is also under the control of the simulation test bench as well as the upstream handshake channel.

One Verilog file will be provided that has all of the necessary design modules. I suggest you build your six pipelines in this file because it will interface better with our simulation test bench, simplify some tool issues, and make it easier for us to grade. (It's always a good idea to keep your grader happy!)

All of your designs will be *structural* Verilog. Create each design by structurally connecting the ports between design modules. For the designs to be properly evaluated from the simulation script provided, the two input signals must be named go_l and go_r, the left channel of the pipeline into the LHS needs to use signals lr and la, and the right channel into the RHS block needs to use signals rr and ra. Each handshake channel between the other pipeline stages will need a unique name. I typically use the notation of rn for the request control channel and an for the acknowledgment control channel, where n indicates the depth of the particular pipeline stage. The definition of such a two deep asynchronous linear pipeline is shown here:

```
// two deep example pipeline using the LC_BM controller
module example (go_l, go_r);
   input go_l, go_r;
   wire  lr, la, rr, ra, r1, a1, rst;

   LHS    l0  (.go_l(go_l), .rst(rst), .out(lr), .in(la));
   LC_BM  p0  (.lr(lr), .la(la), .rr(r1), .ra(a1), .rst(rst));
   LC_BM  p1  (.lr(r1), .la(a1), .rr(rr), .ra(ra), .rst(rst));
   RHS    r0  (.go_r(go_r), .in(rr), .out(ra));

endmodule // example
```

One of the pipeline designs that you will build and analyze consists of four of the burst-mode controllers (LC_BM). Connect these four controllers together in a linear fashion implementing a FIFO. The right channel of the each of the first three controllers will connect to the left channel of one of the last three burst-mode LC blocks in a linear chain. You will then connect LHS to the left channel of the first controller and the RHS component to the right channel of the last controller in the design.

The second pipeline is exactly like the first pipeline, but uses an LC_C block implemented with a C-element and a NOR gate.

The third pipeline will consist of two burst-mode LC controllers LC_BM and two C-Element LC controllers LC_C. You are free to decide the order in which you place these controllers in the four-deep linear pipeline.

The other three pipelines will be modifications of the above three pipelines, where you are adding in three different DELAY blocks between pairs of linear controllers. You will place the delay blocks between the rr and lr ports of adjacent controllers, effectively delaying the observation of the rr event by the downstream controller. There are three such channels. Look at the delay blocks in the Verilog file and modify them to have 1, 3, and 5 units of delay, rather than use the global delay provided as a parameter. (Make sure you scale your delays to match the value of the unit delay used in the Verilog library.) Pick one of the delay elements, and place it in the three handshake channels for the two homogeneous controllers.

You should now have an LC_C and LC_BM pipeline with identical delay elements between the handshake channels. For the final design, pick your design with the two different controllers, and insert each of the three delay elements into one of the three channels (only one delay element per channel, please!). You are free to choose the order in which you place the delays in the pipeline, but these three delays must exist in each pipeline. This will give you six designs, three without any delays, and three with delay elements added. Four of the designs are completely homogeneous in terms of controllers and delay elements (if any) used; the other two are not homogeneous.

4.2 Simulation

You are to simulate each pipeline and report on the results that are provided by the simulation script. You will need to run ModelSim, as done in Lab 2. You will need to create a new project, add in the Verilog file(s) that contains the structural design modules and components. Make sure to include a link to the 65 nm Artisan library so that you can reference the structural cells used in this library.

This simulation will be performed without providing a .sdc file. Thus the simulation will run using the default delay value of the cell library. This should result in a "unit delay" number based on this value.

The simulation script will report forward latency, backward latency, cycle time, and buffering depth of the pipeline. The forward latency is defined as the time from which a left request is sent to the pipeline until a right request emerges from the pipeline in an empty pipeline. The backward latency is defined as the delay from receiving a rising right acknowledgment on the output of the pipeline until a rising left acknowledgment occurs on the input channel of a pipeline that has stalled because the pipeline contains as many tokens as the controllers will permit. The maximum cycle time is the worst case delay for inserting tokens into a pipeline when the receiver does not stall.

If your simulation does not complete or report correct results then you have probably made one of two mistakes. The first potential mistake is that you incorrectly connected the components in your structural design. For example, make sure that rr ports from a linear pipeline controller connect to lr ports in a channel between two control modules. Make sure the wire net names are used consistently. A second potential error could be that you did not follow the correct naming conventions used by the script. Make sure the the leftmost channel of the pipeline uses lr and la, and the rightmost channel uses rr and ra.

Note 1: Simulations using the module LC_C must use the `-novopt` flag or the internal rm signals get optimized out of the design since they are just an "assign" of the lm signals. (In the Transcript window, load the simulation using the command: `vsim -novopt work.DESIGN` where DESIGN is the name of your simulation module.)

Note 2: The results are in unit gate delays where each gate has a delay of 1ns. Make sure to report your results in terms of unit gate delays (ns).

4.3 Writing an Asynchronous Simulation Deck

The simulation deck provided to you is written in `tcl` and uses function commands that interface with Modelsim. Have a careful look at the simulation deck and understand what

it does. In a clocked design data validity is related to the clock. Therefore, the inputs are driven and the outputs are sampled based on the relationship to a clock edge. In an asynchronous design the control signals for each channel dictate data validity. Therefore, one can not periodically sample signals in an asynchronous design and expect the data to be valid. Since asynchronous design is reactive, test benches which operate on asynchronous design usually should also be reactive.

If there were data in the pipeline, then one would need to sample the data based on status of the control signals on a channel. When the request signal rises at the input controller, data should be correct and stable. In this lab, since there is no data path, the test bench just looks at the control signals.

In order to evaluate the design for latency and cycle time, the test bench is reactive to changes on the control lines. See how the test bench observes the control lines and then reacts to them in order to evaluate the design.

In this lab, you will create from scratch a Verilog version of the `tcl` test bench provided. The functionality of the test bench should be the same as the `tcl` test bench – it needs to provide all the same data and results. When run on your circuits, the results should be identical to those of the `tcl` testbench provided in the lab.

5 Deliverables

Turn the assignment in using Canvas.

All work in this lab must be independent. You may not copy any design or Verilog test bench from any source. The designs and code must all be independently created.

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.

Include the files and results in a .tgz file that has the following:

1. You will create three delay elements, one creating 1 unit of delay, one 3 units of delay and another 5 units of delay.

2. The design of a 4-deep pipeline with the LC_BM controller.

3. The design of a 4-deep pipeline with the LC_C controller.

4. The design of a 4-deep pipeline with two LC_BM controllers and two LC_C controllers.

5. You will create a new version of the above three controllers where a DELAY module has been inserted on the link between rr and lr of pair of linear controllers. For the first two designs, insert the same delay element between each of the controllers. Select the same delay element for both of the first two designs. For the third non-homogeneous design that consists of both handshake controllers, put one each of the three delay elements on arbitrarily pipeline stages.

6. Create a new testbench written in Verilog that performs the same tests and reports the same result as the tcl testbench provided.

7. In your README file or in an excel file, make a table that lists the forward latency, backward latency, maximum cycle time, and buffering depth of each of your six pipelines as reported from Modelsim.

8. In your README file, describe how the simulation script behaves, and how it creates results for latency, buffer depth, and cycle time. Also describe the need for the LHS and RHS blocks. Then describe how you wrote your testbench, and any trouble that you had and lessons you learned through the process. Describe how writing a test bench for an asynchronous design differs from a test bench for a clocked design.

9. In the README, include a writeup that discusses the differences between the linear controllers designs. In particular answer the following numbered questions:

   (a) Why you think these different protocols produced different result values.

   (b) Think about how the results are similar or different. Provide your perspective on why you observe the similarities and differences that you observed.

   (c) What is the effect of combining different protocols into a single pipeline?

(d) What is the effect of adding different pipeline delays into a linear pipeline? Do you think your observations generalizes to *any* linear asynchronous pipeline? Why or why not?

(e) How would you compare the performance values of a clocked pipeline that had a combinational delay equivalent to that of the asynchronous pipeline that you designed with various differing delays per stage?

10. Include 12 screen shots of the simulations. One for each of the six designs running under the tcl and the Verilog simulator that you developed.