Using ATACS for Verification of Hazard-Freedom of Phased Logic Wrappers

Michael Boyer
Advisor: Cherrice Traver
Union College
Summer 2004

Table of Contents

1. Phased Logic 2
2. Wrappers 3
3. ATACS 6
4. VHDL Modeling 6
5. Verification 7
6. Results 9
7. Conclusion 10
8. Acknowledgements 10
9. References 11
Appendix A: VHDL Models 12
  A.1 Figure 1 Model 12
  A.2 Figure 2 Model 13
  A.3 Figure 3 Model 14
  A.4 Figure 4 Model 15
  A.5 Delay Kill Block Model 17
Appendix B: VHDL Environment Files 18
  B.1 Environment File for Figures 1, 2, 3 18
  B.2 Environment File for Figure 4 19

1. Phased Logic

Phased Logic (PL) is an asynchronous methodology introduced in [1] that supports the synchronous design paradigm by allowing automated translation from a clocked netlist implementation of a circuit to a self-timed netlist implementation. The fine-grained version of PL uses two wires for every signal; one for data and the other for timing/phase. Each PL signal and PL gate can have one of two phases, either even or odd, using a Level Encoded Dual Rail signaling scheme [2]. When the phase of all of a gate's input signals matches the gate's own internal phase, the gate updates its data output signal and then toggles its phase output signal, a process known as firing. Using the synthesis algorithm in [1] and the addition of feedback signals, we can assure that a synthesized circuit satisfies two important properties: liveness, meaning that there is no way for the flow of data to halt and the circuit to cease operation; and safety, meaning that there is no way for data to be lost between two gates by the sender of the data updating the signal before the receiver has processed it.

Early work with PL utilized a fine-grain mapping technique in which each individual gate in the clocked netlist was mapped directly to a PL gate. More recent work by Reese, Thornton, and Traver [3], however, has utilized a coarse-grain mapping technique in which entire logic blocks in the clocked netlist are enclosed by PL wrapper logic. Because the wrapper logic interfaces between the external PL circuit and the internal logic block, the logic block itself can be reused without modification. The course-grain Phased Logic is very similar to a two-phase micropipeline system [4], but with phase signals taking the place of request signals and feedback signals, where applicable, taking the place of acknowledge signals. There are two types of logic blocks in this methodology [5]: barrier blocks, which contain D-Flip-Flops as well as combinational logic; and through blocks, which contain only combinational logic. Each of these two types is implemented using a different PL wrapper. These PL wrappers are further modified if the block uses time borrowing or early evaluation. Each of the wrapper circuits is described in detail in [5].

Time borrowing is a speed-up technique that can be used when two or more through blocks are connected in series. Normally, a block with a large compute delay will need an equivalently large control delay, but time borrowing allows the control delay through the slow block to be spread to surrounding blocks with smaller compute delays. For example, consider a circuit with only two blocks, A and B, where the output from block A is the input to block B. Assume that block A has a compute delay of 15 ps, block B has a compute delay of 2 ps, and both blocks have a control delay of 5 ps. Without time borrowing, the control delay through block A will have to be increased by 10 ps, creating a total control delay of 20 ps but a total compute delay of only 17 ps. With time borrowing, the compute delay through block A will only be increased by 7 ps, yielding a total control delay of 17 ps and matching the total compute delay. Another speed-up technique, early evaluation, can be used in either barrier or through blocks. Early evaluation can only be used in blocks whose output in certain cases can be determined before all of its inputs have arrived. For example, a block implementing the carry-out from a three-input addition function can determine its output before the carry-in input arrives if its A and B inputs are either both zero or both one. Using early evaluation in a case like this potentially offers a significant speedup.

2. Wrappers

Figure 1 shows the wrapper for through blocks with time borrowing. Separate C-elements are used on the compute path and the control path to ensure that the control path is fast regardless of loading on the latch enable signal. The delay through the control path is important, especially in blocks whose compute delay is relatively small. The delay blocks on the feedback inputs to the lower C-element are necessary to ensure that the data output signal is updated before the phase output signal. The delay block driving the D5 signal in the bottom right corner of the figure is used to lengthen the pulse width of the latch enable signal.

Figure 1: Wrapper for Through Blocks with Time Borrowing

Figure 2 shows the wrapper for through blocks without time borrowing. It is similar to the previous wrapper, except the C-elements are replaced by faster pseudo C-elements (nC-elements), the XOR gates are removed, and there is no delay block to control the pulse width of the latch enable signal. The XOR gates are not needed in this wrapper because the negation of the phase_bar signal, which drives the select inputs to the nC-elements, causes the negation of the nC-element output signals.

Figure 3 shows the wrapper for barrier blocks without early evaluation. Like the previous wrapper, it does not use XOR gates and uses an nC-element instead of a slower full C-element. Unlike the previous wrapper, however, the data output storage is implemented with a DFF instead of a latch. Since this means that both output storage elements are edge-triggered, the two separate nC-elements in the previous wrapper are combined into one nC-element in this wrapper.

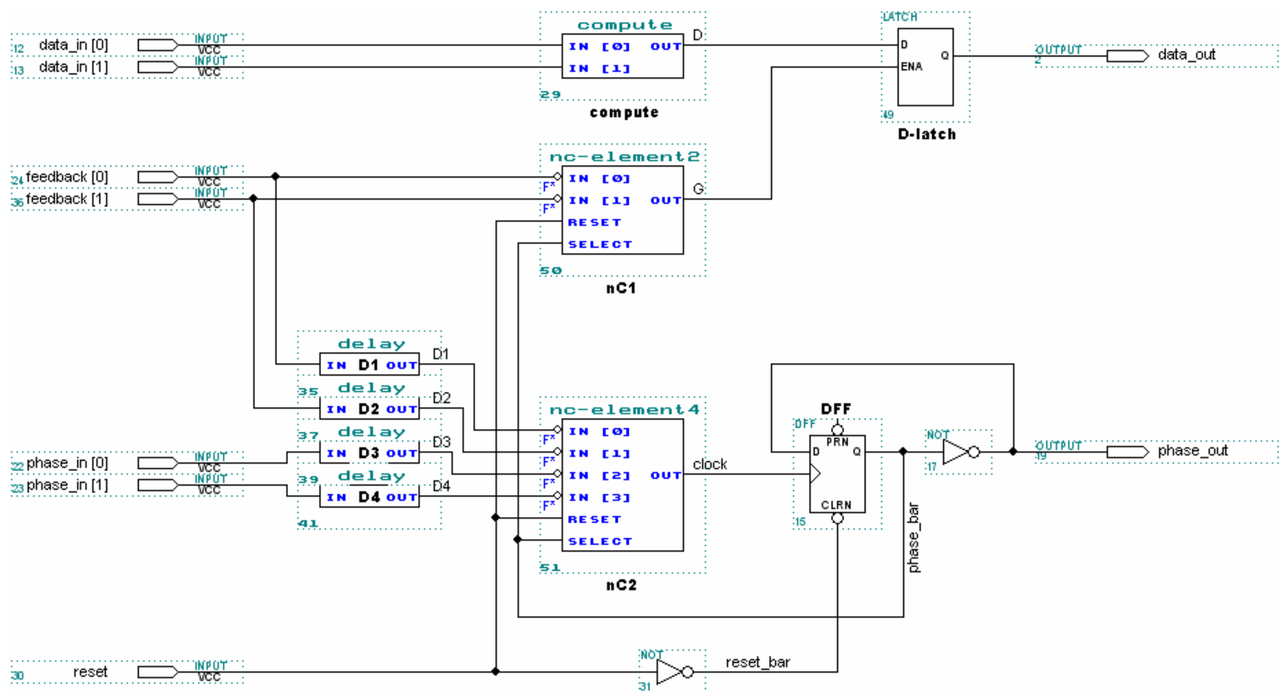

Figure 2: Wrapper for Through Blocks without Time Borrowing

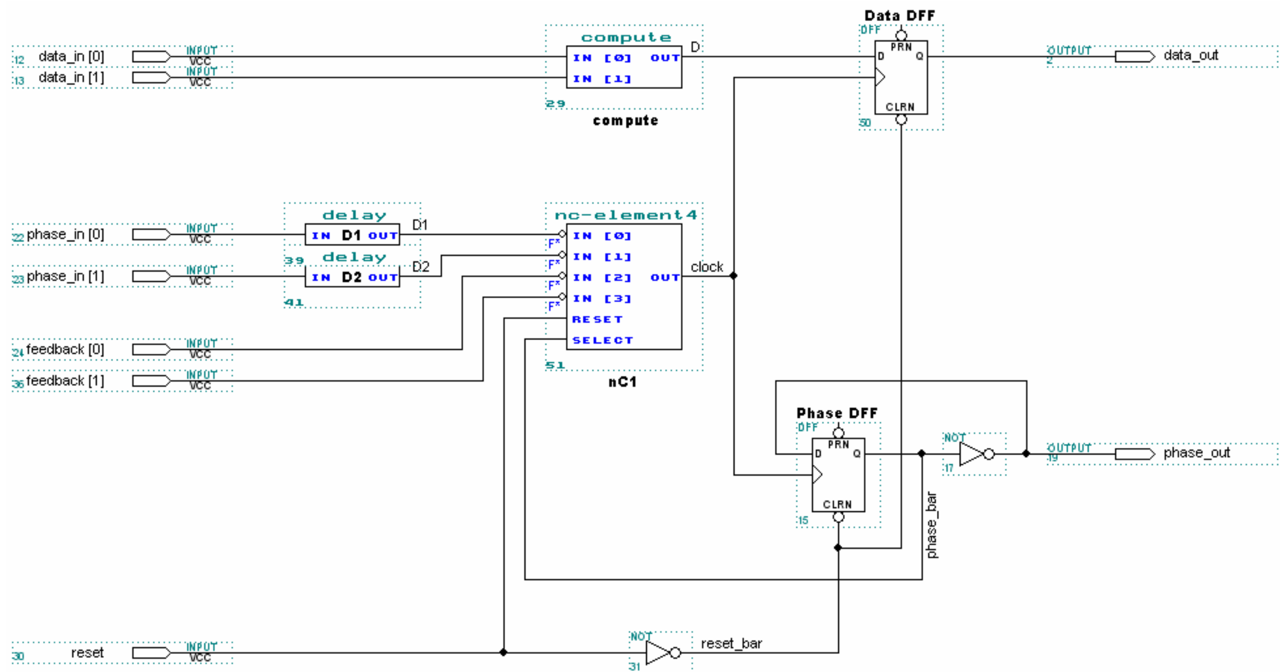

Figure 3: Wrapper for Barrier Blocks without Early Evaluation

Figure 4 shows the wrapper for both barrier and through blocks that use early evaluation. The early evaluation wrapper is by far the most complicated wrapper because its behavior in each cycle depends on whether or not early evaluation is used in that cycle. The term “early fire” refers to the wrapper firing with early evaluation; “late fire” refers to the wrapper firing without early evaluation. The input signals are broken up into early inputs and late inputs, with the possibility that an early input is also a late input. The early evaluation (eeval) block below the compute block outputs a one in the cases where early evaluation can be used and outputs a zero otherwise, based on the values of the early inputs; this value is latched by the early evaluation latch (eelatch). When the eelatch output is low, both multiplexors select the Lphs signal from the late C-element and a late fire occurs. In this case, neither the clock nor the feedback output signals are updated until all of the inputs to arrive. When the eelatch output is high, the top multiplexor selects the Tphs signal from the trigger C-element, the bottom multiplexor selects the Lphs_nodly signal from the all-arrived C-element, and an early fire occurs. In this case, the clock signal will go high after the early phase and feedback signals have arrived; it does not need to wait for the late phase signals to arrive. The negation of the feedback output signal in this case, however, does need to wait for all of the input signals to arrive at the all-arrived C-element, but does not need to wait for any of the input signals to propagate through any delay blocks. As soon as feedback has been provided in an early fire, the new feedback signal arriving at the delay kill (dkill) blocks effectively kills the delay through those blocks.

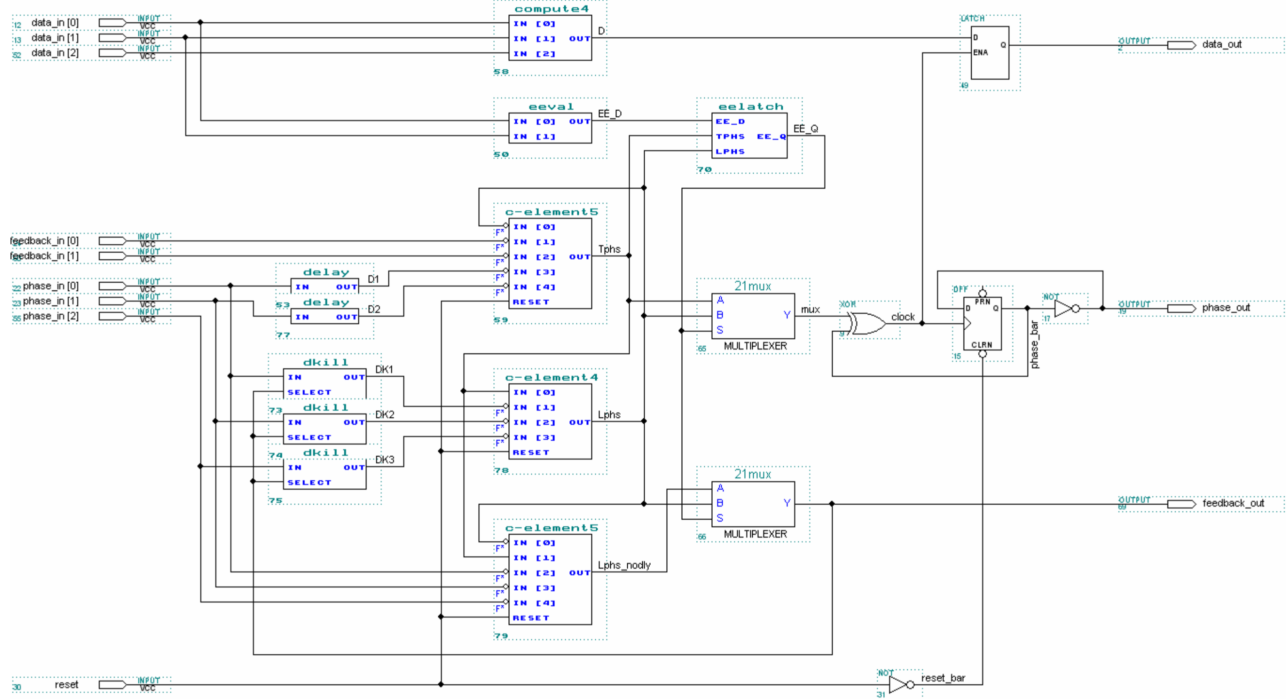

Figure 4: Early Evaluation Wrapper for Barrier and Through Blocks

3. ATACS

Automatic Timed Asynchronous Circuit Synthesis (ATACS) is a tool developed by the Myers Research Group at the University of Utah for the synthesis and verification of timed asynchronous circuits. The use of timed asynchronous circuits allows us to take advantage of known (or assumed) timing information in order to optimize a circuit's performance, as discussed in [6]. ATACS supports a wide range of file types [7]: CSP, handshaking expansion (HSE), VHDL, signal transition graph (STG), burst-mode state machine, (UNC), timed event-rule structure files (ER), timed event/level structure (TEL), and reduced state graph (RSG). We chose to model our circuits exclusively in VHDL because it was the format most familiar to us and was supported by our editing and simulation software. The program supports six different timing methods that affect the verification process: untimed, geometric, POSETs, Bourne Again Geometric (BAG), Bourne Again POSETs (BAP), and BAP Time-Dependent Choice (BAPTDC). The main timing method that we were concerned with was POSETs, which represents timing information using zones and represents timing between events using a POSET matrix.

Although ATACS supports both the synthesis and verification of timed asynchronous circuits, our primary interest for this research was its verification capability. ATACS offers three different types of verification: conformance checking, property checking, and hazard checking. Conformance checking verifies that a specification in one file format and a circuit in another format properly conform to one another. Property checking verifies a Computation Tree Logic (CTL) statement about a circuit. Hazard checking, the type of verification that we used, verifies that there are no hazards present in a circuit. Hazards are defined in [6] as “conditions generated by the structure of the circuit or timing relationships between inputs and propagation delays that can cause incorrect behavior.” [6] describes the two types of hazards: acknowledgment hazards and monotonicity hazards. An acknowledgment hazard occurs when a signal “becomes excited to change to a new value, but its excitation changes value before it can be shown to have stabilized.” A monotonicity hazard occurs when “is supposed to remain stable but it becomes momentarily excited or it is supposed to make a transition which it makes non-monotonically.” [6] also gives algorithms to check for both of these types of hazards; these processes are automated by ATACS, allowing us to check the hazard freedom of our circuits relatively quickly and easily.

4. VHDL Modeling

Although VHDL models of the PL wrappers already existed, they were unsuitable for use in ATACS. This forced us to write completely new models, making sure that they did not contain any code that was incompatible with ATACS. For example, our original model for the D-Flip-Flops used the Boolean expression *clock’event* to identify the rising edge of its clock input signal. However, this type of expression is not supported by ATACS. We were forced to come up with an alternative model for the DFF, and eventually decided on a master-slave flip-flop model. This model was chosen because it only relied on the value of the clock signal and not its transitions, and therefore did not require the use of an incompatible expression.

In order to implement the delay ranges required for our analysis, we used the *delay* function provided by the VHDL package *nondeterminism*, written by Chris Myers [8]. Given the lower and upper bounds of a time range, *delay* returns a randomly-chosen time within that range. This function was used to implement the delay ranges for all of the components in our circuits. For the purposes of our research, the delays through the compute blocks were much more significant than what the blocks did. However, we could not ignore their internal operation altogether; in order for them to be modeled properly, each compute block still needed to actually implement a function. For the three non-early evaluation wrappers, their compute blocks simply implemented a two-input AND function. The compute block in the early evaluation wrapper implemented the carry-out from a three-input addition function. Although technically early evaluation can be accomplished using a two-input AND function, with an early fire occurring when the early input is zero, that is not something that would ever be done in the real world.

Before the VHDL models of the wrappers could be verified by ATACS, we had to somehow specify their expected behavior. We accomplished this by modeling the environment for each of the wrappers in VHDL. The purpose of an environment is to assign values to the input signals and define the expected behavior of the output signals. The process of writing the environment files was simplified by using the VHDL *handshake* package written by Chris Myers [8]. The package features a number of useful functions: *assign*, which assigns values to signals in parallel after a delay chosen from a certain range; *guard*, which waits for a signal to attain a certain value; and *guard_and*, which waits for a set of signals to achieve certain values. The *assign* function was used to assign values to the input signals, while the *guard* and *guard_and* functions were used to specify the expected behavior of the output signals.

We had to make a number of assumptions about the environment in order to properly model them in VHDL. For all of the wrapper circuits, we assumed that the data, phase, and feedback inputs all arrived at the same time. This is a valid assumption because a phase signal cannot arrive before its associated data signal, and a data signal arriving before its associated phase signal cannot possibly introduce a hazard into the circuit. Also, in each of the four wrapper circuits neither the data nor the phase output signals can be updated until all of the feedback signals have arrived, so the circuit effectively waits for the feedback signals before it can proceed. Therefore, the timing of the arrival of the feedback signals has no effect on the hazards present in the circuits. We also assumed a rather large delay between the wrapper firing and receiving new inputs. This is an accurate assumption if the wrapper is connected to at least one other block, as it would be in any real-world application.

The VHDL models for the wrappers and the delay kill block are provided in Appendix A. The VHDL environment files are provided in Appendix B.

5. Verification

As an example of the hazard detection process using ATACS, consider the first wrapper described in this paper, the through block wrapper with time borrowing. Assume that the internal components have the arbitrary ranges of delays show in Figure 5. These ranges are also used in the VHDL model given in Appendix 1. While these specific delay values may or may not represent the delay values that we would find in the real world, the relative order of the values should be accurate. For example, the compute block and the delay blocks have the highest range of delays, while the simple XOR and NOT gates have the lowest range of delays.

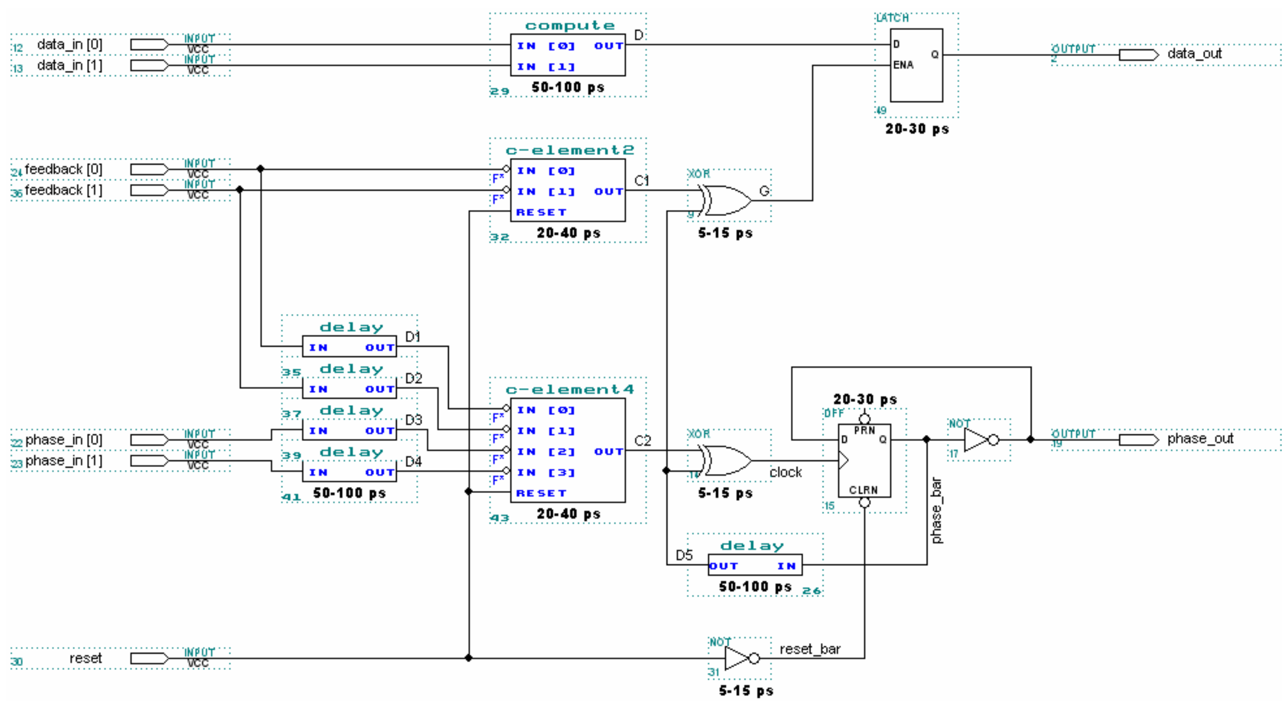

Figure 5: Example Delay Values

Using ATACS, we verified that this wrapper circuit is hazard free within these delay ranges. In other words, selecting for each component any delay value within its given range yields a hazard free circuit. We intentionally made these ranges large to demonstrate that the circuit will operate properly within a wide variety of delay values. In order to demonstrate the conditions under which a hazard arises, consider increasing the range of delay values for the compute block from 50-100 ns to 100-200 ns. Running this modified circuit through ATACS yields the error trace shown in Figure 6 (as well as the same data represented graphically) and an error message indicating that the data_out signal is unable to go low. This is due to the fact that the delay through the compute block is long enough that the time from the compute block output signal (D) being updated to the latch enable signal (G) going low is less than the latch's setup time. Similar errors can be achieved by shortening the delay through the C-element, the XOR gate, or the delay blocks.

```
reset+/1 -> phase_in1+/1 -> phase_in0+/1 -> feedback0+/1 -> feedback1+/1 -> phase_out+/1 ->
reset_bar_wrapper_4-/1 -> c2_wrapper_4-/1 -> c1_wrapper_4-/1 -> d4_wrapper_4+/1 ->
d3_wrapper_4+/1 -> d1_wrapper_4+/1 -> d2_wrapper_4+/1 -> clock_wrapper_4-/1 -> g_wrapper_4-/1 ->
qfs_bar_wrapper_4-/1 -> qfs_wrapper_4+/1 -> reset-/1 -> $6+/1 -> data_in0+/1 -> data_in1+/1 ->
phase_in0-/1 -> phase_in1-/1 -> feedback1-/1 -> feedback0-/1 -> reset_bar_wrapper_4+/1 ->
c1_wrapper_4+/1 -> d3_wrapper_4-/1 -> d4_wrapper_4-/1 -> d2_wrapper_4-/1 -> d1_wrapper_4-/1 ->
g_wrapper_4+/1 -> d_wrapper_4+/1 -> c2_wrapper_4+/1 -> data_out+/1 -> clock_wrapper_4+/1 ->
qf_wrapper_4+/1 -> qf_bar_wrapper_4-/1 -> phase_bar_wrapper_4+/1  -> phase_out-/1 ->
d5_wrapper_4+/1 -> g_wrapper_4-/1 -> clock_wrapper_4-/1 -> qfs_bar_wrapper_4+/1 -> qfs_wrapper_4-
/1 -> $3+/1 -> data_in0-/1 -> data_in1-/1 -> phase_in1+/2 -> phase_in0+/2 -> feedback0+/2 ->
feedback1+/2 -> c1_wrapper_4-/1 -> d4_wrapper_4+/1 -> d3_wrapper_4+/1 -> d1_wrapper_4+/1 ->
d2_wrapper_4+/1 -> g_wrapper_4+/1 -> d_wrapper_4-/1 -> c2_wrapper_4-/1 -> clock_wrapper_4+/1 ->
qf_wrapper_4-/1 -> qf_bar_wrapper_4+/1 -> phase_bar_wrapper_4-/1 -> phase_out+/1 -> d5_wrapper_4-
/1 -> g_wrapper_4-/1 -> FAIL!
```

Figure 6: ATACS Error Trace

6. Results

The automatic hazard detection provided by ATACS gave us the ability to change the component delay ranges for any of the wrappers and promptly verify the hazard-freedom of the resulting circuit. By doing this systematically for each of the wrappers we were able to establish relationships between the component delay ranges and the resulting hazard-freedom of the wrapper. These results are summarized in Figure 7. In these equations, the name of a component (i.e. D3, xorG, etc.) is used to represents the range of delays for that component. The min() and max() functions represent the minimum and maximum values, respectively, of a given delay range. The min(D1, D2, D3, D4) terms represent the minimum delay through any one of those four delay blocks, because the delay range for each block is assumed to be the same.

Wrapper for Through Blocks with Time Borrowing (Figure 1):

- min(D1, D2, D3, D4) + min(D5) ≥ max(C1) + max(xorG) + max(D-latch) – min(C2) – min(xorC) – min(DFF)
- min(D1, D2, D3, D4) + min(D5) ≥ max(compute) + max(D-latch) – min(C2) – min(xorC) – min(DFF)

Wrapper for Through Blocks without Time Borrowing (Figure 2):

- min(D1, D2, D3, D4) ≥ max(compute) + max(D-latch) – min(nC2) – min(DFF) – min(nC1)

Wrapper for Barrier Blocks without Early Evaluation (Figure 3):

- min(D1, D2, D3, D4) ≥ max(compute) + max(Data DFF setup time) – min(nC)

Early Evaluation Wrapper for Barrier and Through Blocks (Figure 4):

- Results Forthcoming

Figure 7: Minimum Delay Block Delays Required for Hazard-Free Circuits

The equations are given in the form above, with all of the terms involving delay blocks isolated on the left hand side, because this is the most useful form for the design and implementation of the wrapper circuits. Given the delay ranges for all of the components other than the delay blocks, a designer can easily calculate the minimum required delay through the delay blocks which will still result in a hazard-free circuit. Of course it is suggested that the designer also add a margin to account for discrepancies between predicted delays and the delays of the final physical layout.

It is important to note that using the above equations to design a wrapper will only guarantee the hazard-freedom of the internal operation of the wrapper, not the external circuit as a whole. In order to ensure the proper operation of the entire circuit, we must also adhere to the pre-existing PL timing constraints used in the current mapping process. Specifically, for non-feedback control inputs, the minimum time from the arrival of the input signal to the updating of the clock input of the phase DFF must be greater than or equal to the maximum time from the arrival of the input signal to the updating of the D input of the D-latch. For feedback inputs, the minimum time from the arrival of the input signal to the updating of the clock input of the phase DFF must be greater than or equal to the maximum time from the arrival of the input signal to the updating of the enable input of the D-latch.

For example, for the delays shown in Figure 5, we calculate the minimum required delay through the delay blocks to be 85 ps. Assume that we choose to implement this delay entirely in the D5 block and make the delay through the D1, D2, D3, and D4 blocks zero. Then the phase output signal can be updated in as little as 45 ps after the arrival of the input signals, while it can take up to 130 ps to update the data output signal. This represents a violation of the PL requirement that every block updates its data output signals before or at the same time that it updates its phase output signal. So even though the resulting wrapper circuit will have no internal hazards, hazards may exist when the wrapper is connected to other blocks.

7. Conclusion

The introduction of the asynchronous design methodology known as Phased Logic gave digital designers a powerful new alternative to the traditional synchronous design approach. PL's support for the automated translation from a clocked netlist to a self-timed netlist greatly simplifies the asynchronous design process and allows asynchronous designers to use existing synchronous development tools. The extension of PL to its current coarse-grain implementation represented yet another simplification of the design process, because it allows existing logic blocks to be enclosed by wrapper logic with no modification to their internal structure. In addition, adhering to the PL timing constraints guarantees the hazard-free operation of the resulting circuit at the block level. Until now, however, there existed no timing constraints governing the internal operation of the wrapper circuits. This research is an attempt to fill that void and simplify the PL design process even further. Using VHDL models of the PL wrapper circuits and the automated hazard-checking in ATACS, we were able to produce equations for each wrapper specifying the minimum delay through the delay blocks required to guarantee hazard-freedom. Using these equations along with the PL timing constraints that are currently implemented in the mapping tool will guarantee the correct and hazard-free operation of a coarse-grain Phased Logic circuit.

8. Acknowledgements

I would like to thank Chris Meyers at the University of Utah, not only for his role in the development of ATACS and related VHDL packages, but also for his personal assistance during all stages of this research. I would also like to thank Bob Reese at Mississippi State University for his help in understanding the Phased Logic wrapper circuits. This research was funded by a grant from IBM/SRC.

9. References

[1] Daniel H. Linder and James C. Harden, "Phased Logic: Supporting the Synchronous Design Paradigm with Delay-Insensitive Circuitry." IEEE Transactions on Computers, Vol. 45, No. 9, September 1996.

[2] M.E. Dean, T.E. Williams, and D.L. Dill, "Efficient Self-Timing with Level-Encoded 2-Phase Dual-Rail (LEDR)," in *Advanced Research in VLSI*, 1991.

[3] Robert B. Reese, Mitchell A. Thornton, and Cherrice Traver, "A Course-Grain Phased Logic CPU", 9th IEEE International Symposium on Asynchronous Circuits and Systems (Async 2003), Vancouver, Canada.

[4] Sutherland, "Micropipelines", *Communications of the ACM*, Vol 32, No. 6, June 1989, pp. 720-738.

[5] Robert B. Reese, Mitchell A. Thornton, and Cherrice Traver, "Async 2004 Tutorial – Phased Logic", Workshop Notes, 10th International Symposium on Advanced Research in Asynchronous Circuits and Systems, Crete, Greece, April 2004.

[6] Curtis A. Nelson, Chris J. Meyers, and Tomohiro Yoneda, "Efficient Verification of Hazard-Freedom in Gate-Level Timed Asynchronous Circuits", 2003 International Conference on Computer-Aided Design, November 2003.

[7] Myers Research Group, "The ATACS User's Manual", University of Utah, http://www.async.ece.utah.edu/tools/atacsman.html

[8] Myers, Chris. Asynchronous Circuit Design. New York: Wiley, 2001.

Appendix A: VHDL Models

A.1 Figure 1 Model

```
-- Phased Logic wrapper for through blocks with time borrowing
-- (Figure 4 from "ASYNC Tutorial 2004 - Phased Logic")

library ieee;
use ieee.std_logic_1164.all;
use work.nondeterminism.all;

entity figure_4 is
  port (
    data_in0, data_in1, phase_in0, phase_in1 : in    std_logic;
    feedback0, feedback1, reset              : in    std_logic;
    phase_out, data_out                      : inout std_logic);
end figure_4;


architecture behavior of figure_4 is
  -- internal signals
  signal D, D1, D2, D3, D4, D5, phase_bar, QFs, QF    : std_logic := '0';
  signal C1, C2, clock, G, QFs_bar, QF_bar, reset_bar : std_logic := '1';

  -- delay values (L = lower bound, U = upper bound)
  constant L_delay   : integer := 50;
  constant U_delay   : integer := 100;
  constant L_D5      : integer := 50;
  constant U_D5      : integer := 100;
  constant L_compute : integer := 50;
  constant U_compute : integer := 100;
  constant L_C1      : integer := 20;
  constant U_C1      : integer := 40;
  constant L_C2      : integer := 20;
  constant U_C2      : integer := 40;
  constant L_xorG    : integer := 5;
  constant U_xorG    : integer := 15;
  constant L_xorC    : integer := 5;
  constant U_xorC    : integer := 15;
  constant L_not     : integer := 5;
  constant U_not     : integer := 15;
  constant L_dlatch  : integer := 20;
  constant U_dlatch  : integer := 30;
  constant L_dff     : integer := 10;   -- 2 * (dff delay) = actual dff delay
  constant U_dff     : integer := 15;

begin
  -- data signal from compute block
  D <= data_in0 and data_in1 after delay(L_compute, U_compute);

  -- delay blocks
  D1 <= feedback0 after delay(L_delay, U_delay);
  D2 <= feedback1 after delay(L_delay, U_delay);
  D3 <= phase_in0 after delay(L_delay, U_delay);
  D4 <= phase_in1 after delay(L_delay, U_delay);
  D5 <= phase_bar after delay(L_D5, U_D5);

  -- latch enable signal
  G <= C1 xor D5 after delay(L_xorG, U_xorG);

  -- dff clock signal
  clock <= C2 xor D5 after delay(L_xorC, U_xorC);

  -- dff reset signal (active low)
  reset_bar <= not reset after delay(L_not, U_not);

  -- c-element output signals
  C1 <= ( ( (feedback0 and feedback1) or (not C1) ) and (feedback0 or feedback1) ) nor reset
        after delay(L_C1, U_C1);
  C2 <= ( ( (((D1 and D2) and D3) and D4) or (not C2) ) and (((D1 or D2) or D3) or D4) ) nor
        reset after delay(L_C2, U_C2);
```

```
  -- d-latch model
  data_out <= '1' after delay(L_dlatch, U_dlatch) when G = '1' and D = '1'
         else '0' after delay(L_dlatch, U_dlatch) when G = '1' and D = '0'
         else data_out after delay(L_dlatch, U_dlatch);

  -- dff model with reset logic
  phase_out <= '1'         after delay(L_dff, U_dff) when reset_bar = '0'
               else QF_bar after delay(L_dff, U_dff);
  phase_bar <= '0'         after delay(L_dff, U_dff) when reset_bar = '0'
               else QF     after delay(L_dff, U_dff);
  QFs       <= '1'         after delay(L_dff, U_dff) when clock = '0' and phase_out = '1' else
               '0'         after delay(L_dff, U_dff) when clock = '0' and phase_out = '0' else
               QFs         after delay(L_dff, U_dff);
  QF        <= '1'         after delay(L_dff, U_dff) when clock = '1' and QFs = '1'       else
               '0'         after delay(L_dff, U_dff) when clock = '1' and QFs = '0'       else
               QF          after delay(L_dff, U_dff);
  QFs_bar   <= '0'         after delay(L_dff, U_dff) when clock = '0' and phase_out = '1' else
               '1'         after delay(L_dff, U_dff) when clock = '0' and phase_out = '0' else
               QFs_bar     after delay(L_dff, U_dff);
  QF_bar    <= '0'         after delay(L_dff, U_dff) when clock = '1' and QFs_bar = '0'   else
               '1'         after delay(L_dff, U_dff) when clock = '1' and QFs_bar = '1'   else
               QF_bar      after delay(L_dff, U_dff);

end behavior;
```

A.2 Figure 2 Model

```
-- Phased Logic wrapper for through blocks without time borrowing
-- (Figure 7 from "ASYNC Tutorial 2004 - Phased Logic")

library ieee;
use ieee.std_logic_1164.all;
use work.nondeterminism.all;

entity figure_7 is
  port (
    data_in0, data_in1, phase_in0, phase_in1 : in    std_logic;
    feedback0, feedback1, reset              : in    std_logic;
    phase_out, data_out                      : inout std_logic);
end figure_7;


architecture behavior of figure_7 is
  -- internal signals
  signal D, D1, D2, D3, D4, phase_bar, QFs, QF : std_logic := '0';
  signal clock, G, QFs_bar, QF_bar, reset_bar  : std_logic := '1';

  -- delay values (L = lower bound, U = upper bound)
  constant L_delay      : integer := 80;
  constant U_delay      : integer := 100;
  constant L_compute    : integer := 50;
  constant U_compute    : integer := 100;
  constant L_nc_element : integer := 15;
  constant U_nc_element : integer := 30;
  constant L_not        : integer := 5;
  constant U_not        : integer := 15;
  constant L_dlatch     : integer := 20;
  constant U_dlatch     : integer := 30;
  constant L_dff        : integer := 10;  -- 2 * (dff delay) = actual dff delay
  constant U_dff        : integer := 15;

begin
  -- data signal from compute block
  D <= data_in0 and data_in1 after delay(L_compute, U_compute);

  -- delay blocks
  D1 <= feedback0 after delay(L_delay, U_delay);
  D2 <= feedback1 after delay(L_delay, U_delay);
  D3 <= phase_in0 after delay(L_delay, U_delay);
  D4 <= phase_in1 after delay(L_delay, U_delay);

  -- dff reset signal (active low)
  reset_bar <= not reset after delay(L_not, U_not);
```

```
  -- latch enable and dff clock signals (nc-element output signals)
  G     <= feedback0 and feedback1              after delay(L_nc_element, U_nc_element) when
                                                phase_bar = '1' else
           not (feedback0 or feedback1 or reset) after delay(L_nc_element, U_nc_element);
  clock <= (((D1 and D2) and D3) and D4)        after delay(L_nc_element, U_nc_element) when
                                                phase_bar = '1' else
           not (D1 or D2 or D3 or D4 or reset)  after delay(L_nc_element, U_nc_element);

  -- d-latch model
  data_out <= '1'      after delay(L_dlatch, U_dlatch) when G = '1' and D = '1' else
              '0'      after delay(L_dlatch, U_dlatch) when G = '1' and D = '0' else
              data_out after delay(L_dlatch, U_dlatch);

  -- dff model with reset logic
  phase_out <= '1'         after delay(L_dff, U_dff) when reset_bar = '0' else
               QF_bar after delay(L_dff, U_dff);
  phase_bar <= '0'         after delay(L_dff, U_dff) when reset_bar = '0' else
               QF     after delay(L_dff, U_dff);
  QFs       <= '1'         after delay(L_dff, U_dff) when clock = '0' and phase_out = '1' else
               '0'         after delay(L_dff, U_dff) when clock = '0' and phase_out = '0' else
               QFs         after delay(L_dff, U_dff);
  QF        <= '1'         after delay(L_dff, U_dff) when clock = '1' and QFs = '1'       else
               '0'         after delay(L_dff, U_dff) when clock = '1' and QFs = '0'       else
               QF          after delay(L_dff, U_dff);
  QFs_bar   <= '0'         after delay(L_dff, U_dff) when clock = '0' and phase_out = '1' else
               '1'         after delay(L_dff, U_dff) when clock = '0' and phase_out = '0' else
               QFs_bar     after delay(L_dff, U_dff);
  QF_bar    <= '0'         after delay(L_dff, U_dff) when clock = '1' and QFs_bar = '0'   else
               '1'         after delay(L_dff, U_dff) when clock = '1' and QFs_bar = '1'   else
               QF_bar      after delay(L_dff, U_dff);

end behavior;
```

A.3 Figure 3 Model

```
-- Phased Logic wrapper for barrier blocks without early evaluation
-- (Figure 6 from "ASYNC 2004 Tutorial - Phased Logic")

library ieee;
use ieee.std_logic_1164.all;
use work.nondeterminism.all;

entity figure_6 is
  port (
    data_in0, data_in1, phase_in0, phase_in1 : in    std_logic;
    feedback0, feedback1, reset              : in    std_logic;
    phase_out, data_out                      : inout std_logic);
end figure_6;


architecture behavior of figure_6 is
  -- internal signals
  signal D, D1, D2, phase_bar, QF1s, QF1, QF2s, QF2 : std_logic := '0';
  signal clock, QF1s_bar, QF1_bar, reset_bar        : std_logic := '1';

  -- delay values (L = lower bound, U = upper bound)
  constant L_delay      : integer := 100;
  constant U_delay      : integer := 150;
  constant L_compute    : integer := 50;
  constant U_compute    : integer := 100;
  constant L_nc_element : integer := 15;
  constant U_nc_element : integer := 30;
  constant L_not        : integer := 5;
  constant U_not        : integer := 15;
  constant L_dff        : integer := 10;  -- 2 * (dff delay) = actual dff delay
  constant U_dff        : integer := 15;

begin
  -- data signal from compute block
  D <= data_in0 and data_in1 after delay(L_compute, U_compute);

  -- delay blocks
  D1 <= phase_in0 after delay(L_delay, U_delay);
  D2 <= phase_in1 after delay(L_delay, U_delay);
```

```
  -- dff clock signal (nC-element output)
  clock <= D1 and D2 and feedback0 and feedback1             after delay(L_nc_element,
                                                                         U_nc_element)
                                                             when phase_bar = '1' else
           not (D1 or D2 or feedback0 or feedback1 or reset) after delay(L_nc_element,
                                                                         U_nc_element);

  -- dff reset signal (active low)
  reset_bar <= not reset after delay(L_not, U_not);

  -- data dff model with reset logic
  QF2s     <= '1'  after delay(L_dff, U_dff) when clock = '0' and D = '1'    else
              '0'  after delay(L_dff, U_dff) when clock = '0' and D = '0'    else
              QF2s after delay(L_dff, U_dff);
  QF2      <= '1'  after delay(L_dff, U_dff) when clock = '1' and QF2s = '1' else
              '0'  after delay(L_dff, U_dff) when clock = '1' and QF2s = '0' else
              QF2  after delay(L_dff, U_dff);
  data_out <= '0'  after delay(L_dff, U_dff) when reset_bar = '0'            else
              QF2  after delay(L_dff, U_dff);

  -- phase dff model with reset logic
  QF1s      <= '1'      after delay(L_dff, U_dff) when clock = '0' and phase_out = '1' else
               '0'      after delay(L_dff, U_dff) when clock = '0' and phase_out = '0' else
               QF1s     after delay(L_dff, U_dff);
  QF1s_bar  <= '0'      after delay(L_dff, U_dff) when clock = '0' and phase_out = '1' else
               '1'      after delay(L_dff, U_dff) when clock = '0' and phase_out = '0' else
               QF1s_bar after delay(L_dff, U_dff);
  QF1       <= '1'      after delay(L_dff, U_dff) when clock = '1' and QF1s = '1'      else
               '0'      after delay(L_dff, U_dff) when clock = '1' and QF1s = '0'      else
               QF1      after delay(L_dff, U_dff);
  QF1_bar   <= '0'      after delay(L_dff, U_dff) when clock = '1' and QF1s_bar = '0'  else
               '1'      after delay(L_dff, U_dff) when clock = '1' and QF1s_bar = '1'  else
               QF1_bar  after delay(L_dff, U_dff);
  phase_out <= '1'      after delay(L_dff, U_dff) when reset_bar = '0'                 else
               QF1_bar  after delay(L_dff, U_dff);
  phase_bar <= '0'      after delay(L_dff, U_dff) when reset_bar = '0'                 else
               QF1      after delay(L_dff, U_dff);

end behavior;
```

A.4 Figure 4 Model

```
-- Phased Logic early evaluation wrapper for barrier and through blocks
-- (Figure 8 from "ASYNC Tutorial 2004 - Phased Logic")

library ieee;
use ieee.std_logic_1164.all;
use work.nondeterminism.all;

entity figure_8 is
  port (
    data_in0, data_in1, data_in2, phase_in0, phase_in1 : in    std_logic;
    phase_in2, feedback_in0, feedback_in1, reset       : in    std_logic;
    phase_out, data_out, feedback_out                  : inout std_logic);
end figure_8;


architecture behavior of figure_8 is
  -- internal signals
  signal D, D1, D2, clock, mux, phase_bar, QF, QFs   : std_logic := '0';
  signal DK1, DK2, DK3, EE_D, EE_Q, Lphs, Lphs_nodly : std_logic := '1';
  signal QFs_bar, QF_bar, reset_bar, Tphs            : std_logic := '1';

  -- delay values (L = lower bound, U = upper bound)
  constant L_delay   : integer := 50;
  constant U_delay   : integer := 100;
  constant L_compute : integer := 50;
  constant U_compute : integer := 100;
  constant L_eeval   : integer := 25;
  constant U_eeval   : integer := 50;
  constant L_C1      : integer := 20;
  constant U_C1      : integer := 40;
  constant L_C2      : integer := 20;
  constant U_C2      : integer := 40;
  constant L_C3      : integer := 20;
  constant U_C3      : integer := 40;
```

```
  constant L_xor     : integer := 5;
  constant U_xor     : integer := 15;
  constant L_not     : integer := 5;
  constant U_not     : integer := 15;
  constant L_dlatch  : integer := 20;
  constant U_dlatch  : integer := 30;
  constant L_dff     : integer := 10;    -- 2 * (dff delay) = actual dff delay
  constant U_dff     : integer := 15;
  constant L_mux     : integer := 10;
  constant U_mux     : integer := 20;

  component dkill
    port (
      in0, select0 : in    std_logic;
      out0         : inout std_logic);
  end component;

begin
  -- data signal from compute block
  D <= (data_in0 and data_in1) or (data_in0 and data_in2) or (data_in1 and data_in2) after
delay(L_compute, U_compute);

  -- early evaluation block
  EE_D <= not (data_in0 xor data_in1) after delay(L_eeval, U_eeval);

  -- early evaluation latch
  EE_Q <= '1'  after delay(L_dlatch, U_dlatch) when (Tphs xor Lphs) = '1' and EE_D = '1' else
          '0'  after delay(L_dlatch, U_dlatch) when (Tphs xor Lphs) = '1' and EE_D = '0' else
          EE_Q after delay(L_dlatch, U_dlatch);

  -- delay blocks
  D1 <= phase_in0 after delay(L_delay, U_delay);
  D2 <= phase_in1 after delay(L_delay, U_delay);

  -- delay kill blocks
  dkill_1 : dkill
    port map (
      in0     => phase_in0,
      select0 => feedback_out,
      out0    => DK1);
  dkill_2 : dkill
    port map (
      in0     => phase_in1,
      select0 => feedback_out,
      out0    => DK2);
  dkill_3 : dkill
    port map (
      in0     => phase_in2,
      select0 => feedback_out,
      out0    => DK3);

  -- C-element outputs
  Tphs    <= ( ( (Lphs and feedback_in0 and feedback_in1 and D1 and D2) or (not Tphs) ) and (Lphs
             or feedback_in0 or feedback_in1 or D1 or D2) ) nor reset after delay(L_C1, U_C1);
  Lphs    <= ( ( ((not Tphs) and DK1 and DK2 and DK3) or (not Lphs) ) and ((not Tphs) or DK1 or
             DK2 or DK3) ) nor reset after delay(L_C2, U_C2);
  Lphs_nodly <= ( ( ((not Lphs) and Tphs and phase_in0 and phase_in1 and phase_in2) or (not
                Lphs_nodly) ) and ((not Lphs) or Tphs or phase_in0 or phase_in1 or phase_in2) )
                nor reset after delay(L_C3, U_C3);

  -- 2 to 1 multiplexor outputs
  mux          <= '1'          after delay(L_mux, U_mux) when EE_Q = '0' and Lphs = '1'       else
                  '0'          after delay(L_mux, U_mux) when EE_Q = '0' and Lphs = '0'       else
                  '1'          after delay(L_mux, U_mux) when EE_Q = '1' and Lphs_nodly = '1' else
                  '0'          after delay(L_mux, U_mux) when EE_Q = '1' and Lphs_nodly = '0' else
                  mux          after delay(L_mux, U_mux);
  feedback_out <= '1'          after delay(L_mux, U_mux) when EE_Q = '0' and Lphs = '1'       else
                  '0'          after delay(L_mux, U_mux) when EE_Q = '0' and Lphs = '0'       else
                  '1'          after delay(L_mux, U_mux) when EE_Q = '1' and Tphs = '1'       else
                  '0'          after delay(L_mux, U_mux) when EE_Q = '1' and Tphs = '0'       else
                 feedback_out after delay(L_mux, U_mux);

  -- xor gate (clock signal)
  clock <= mux xor phase_bar after delay(L_xor, U_xor);

  -- dff reset signal (active low)
  reset_bar <= not reset after delay(L_not, U_not);
```

```
  -- d-latch model
  data_out <= '1'      after delay(L_dlatch, U_dlatch) when clock = '1' and D = '1' else
              '0'      after delay(L_dlatch, U_dlatch) when clock = '1' and D = '0' else
              data_out after delay(L_dlatch, U_dlatch);

-- dff model with reset logic
  phase_out <= '1'     after delay(L_dff, U_dff) when reset_bar = '0'                 else
               QF_bar  after delay(L_dff, U_dff);
  phase_bar <= '0'     after delay(L_dff, U_dff) when reset_bar = '0'                 else
               QF      after delay(L_dff, U_dff);
  QFs       <= '1'     after delay(L_dff, U_dff) when clock = '0' and phase_out = '1' else
               '0'     after delay(L_dff, U_dff) when clock = '0' and phase_out = '0' else
               QFs     after delay(L_dff, U_dff);
  QF        <= '1'     after delay(L_dff, U_dff) when clock = '1' and QFs = '1'       else
               '0'     after delay(L_dff, U_dff) when clock = '1' and QFs = '0'       else
               QF      after delay(L_dff, U_dff);
  QFs_bar   <= '0'     after delay(L_dff, U_dff) when clock = '0' and phase_out = '1' else
               '1'     after delay(L_dff, U_dff) when clock = '0' and phase_out = '0' else
               QFs_bar after delay(L_dff, U_dff);
  QF_bar    <= '0'     after delay(L_dff, U_dff) when clock = '1' and QFs_bar = '0'   else
               '1'     after delay(L_dff, U_dff) when clock = '1' and QFs_bar = '1'   else
               QF_bar  after delay(L_dff, U_dff);

end behavior;
```

A.5 Delay Kill Block Model

```
library ieee;
use ieee.std_logic_1164.all;
use work.nondeterminism.all;

entity dkill is
  port (
    in0, select0 : in    std_logic;
    out0         : inout std_logic);
end dkill;


architecture behavior of dkill is
  -- internal signals
  signal A1, A2, A3 : std_logic;
  signal B1, B2, B3 : std_logic;

  -- delay values (L = lower bound, U = upper bound)
  constant L_and : integer := 25;
  constant U_and : integer := 50;
  constant L_or  : integer := 25;
  constant U_or  : integer := 50;
  constant L_mux : integer := 20;
  constant U_mux : integer := 30;

begin
  -- AND gates
  A1 <= in0 and select0 after delay(L_and, U_and);
  A2 <= in0 and A1      after delay(L_and, U_and);
  A3 <= in0 and A2      after delay(L_and, U_and);

  -- OR gates
  B1 <= in0 or select0 after delay(L_or, U_or);
  B2 <= in0 or B1      after delay(L_or, U_or);
  B3 <= in0 or B2      after delay(L_or, U_or);

  -- multiplexor
  out0 <= '1'  after delay(L_mux, U_mux) when select0 = '0' and B3 = '1' else
          '0'  after delay(L_mux, U_mux) when select0 = '0' and B3 = '0' else
          '1'  after delay(L_mux, U_mux) when select0 = '1' and A3 = '1' else
          '0'  after delay(L_mux, U_mux) when select0 = '1' and A3 = '0' else
          out0 after delay(L_mux, U_mux);

end behavior;
```

Appendix B: VHDL Environment Files

B.1 Environment File for Figures 1, 2, 3

```
-- Environment for Phased Logic wrapper for: [1] through blocks with time borrowing
--                                          [2] through blocks without time borrowing
--                                          [3] barrier blocks without early evaluation

library ieee;
use ieee.std_logic_1164.all;
use work.nondeterminism.all;
use work.handshake.all;

entity environment_4 is
end environment_4;


architecture env_4 of environment_4 is
  component figure_4
    port (
      data_in0, data_in1, phase_in0, phase_in1 : in    std_logic;
      feedback0, feedback1, reset              : in    std_logic;
      phase_out, data_out                      : inout std_logic);
  end component;

  signal data_in0, data_in1, phase_in0, phase_in1         : std_logic;
  signal feedback0, feedback1, reset, phase_out, data_out : std_logic;

begin
  wrapper_4 : figure_4
    port map (
      data_in0  => data_in0,
      data_in1  => data_in1,
      phase_in0 => phase_in0,
      phase_in1 => phase_in1,
      feedback0 => feedback0,
      feedback1 => feedback1,
      reset     => reset,
      phase_out => phase_out,
      data_out  => data_out);

  process
  begin

    -- reset the circuit
    assign(reset, '1', 1, 2);
    -- reset the input signals
    --assign(data_in0, '0', 1, 2, data_in1, '0', 1, 2);
    assign(phase_in0, '1', 1, 2, phase_in1, '1', 1, 2);
    assign(feedback0, '1', 1, 2, feedback1, '1', 1, 2);
    -- wait until the phase dff has been reset
    guard(phase_out, '1');
    wait for delay(100, 150);
    assign(reset, '0', 1, 2);

    loop

      -- assign data values
      assign(data_in0, '1', 0, 0, data_in1, '1', 0, 0);
      -- assign phase values
      assign(phase_in0, '0', 0, 0, phase_in1, '0', 0, 0);
      -- assign feedback values
      assign(feedback0, '0', 0, 0, feedback1, '0', 0, 0);
      -- wait for data and phase outputs to change
      guard_and(data_out, '1', phase_out, '0');
      wait for delay(200, 250);

      -- assign data values
      assign(data_in0, '0', 0, 0, data_in1, '0', 0, 0);
      -- assign phase values
      assign(phase_in0, '1', 0, 0, phase_in1, '1', 0, 0);
      -- assign feedback values
      assign(feedback0, '1', 0, 0, feedback1, '1', 0, 0);
      -- wait for data and phase outputs to change
```

```
      guard_and(data_out, '0', phase_out, '1');
      wait for delay(200, 250);

    end loop;
  end process;

end env_4;
```

B.2 Environment File for Figure 4

```
-- Environment for Phased Logic early evaluation wrapper for barrier and through blocks

library ieee;
use ieee.std_logic_1164.all;
use work.nondeterminism.all;
use work.handshake.all;

entity environment_8 is
end environment_8;


architecture env_8 of environment_8 is
  component figure_8
    port (
      data_in0, data_in1, data_in2, phase_in0, phase_in1 : in    std_logic;
      phase_in2, feedback_in0, feedback_in1, reset       : in    std_logic;
      phase_out, data_out, feedback_out                  : inout std_logic);
  end component;

  signal data_in0, data_in1, data_in2, phase_in0, phase_in1, phase_in2        : std_logic;
  signal feedback_in0, feedback_in1, reset, phase_out, data_out, feedback_out : std_logic;

begin
  wrapper_8 : figure_8
    port map (
      data_in0     => data_in0,
      data_in1     => data_in1,
      data_in2     => data_in2,
      phase_in0    => phase_in0,
      phase_in1    => phase_in1,
      phase_in2    => phase_in2,
      feedback_in0 => feedback_in0,
      feedback_in1 => feedback_in1,
      reset        => reset,
      phase_out    => phase_out,
      data_out     => data_out,
      feedback_out => feedback_out);

  process
  begin

    -- reset the circuit
    assign(reset, '1', 1, 2);
    -- reset the input signals
    assign(data_in1, '1', 1, 2);
    assign(phase_in0, '1', 1, 2, phase_in1, '1', 1, 2, phase_in2, '1', 1, 2);
    assign(feedback_in0, '1', 1, 2, feedback_in1, '1', 1, 2);
    -- wait until the phase dff has been reset
    guard(phase_out, '1');
    wait for delay(500, 500);
    assign(reset, '0', 1, 2);

    loop

      -- EARLY FIRE
      -- assign data values
      assign(data_in0, '1', 0, 0, data_in2, '1', 0, 0);
      -- assign phase values
      assign(phase_in0, '0', 0, 0, phase_in1, '0', 0, 0, phase_in2, '0', 0, 0);
      -- assign feedback values
      assign(feedback_in0, '0', 0, 0, feedback_in1, '0', 0, 0);
      -- wait for data and phase outputs to change
      guard_and(data_out, '1', phase_out, '0');
      wait for delay(500, 500);

      -- LATE FIRE
      -- assign data values
```

```
      assign( data_in1, '0', 0, 0);
      -- assign phase values
      assign(phase_in0, '1', 0, 0, phase_in1, '1', 0, 0, phase_in2, '1', 0, 0);
      -- assign feedback values
      assign(feedback_in0, '1', 0, 0, feedback_in1, '1', 0, 0);
      -- wait for data and phase outputs to change
      guard_and(data_out, '0', phase_out, '1');
      wait for delay(200, 250);

      -- EARLY FIRE
      -- assign data values
      assign(data_in0, '0', 0, 0, data_in2, '0', 0, 0);
      -- assign phase values
      assign(phase_in0, '0', 0, 0, phase_in1, '0', 0, 0, phase_in2, '0', 0, 0);
      -- assign feedback values
      assign(feedback_in0, '0', 0, 0, feedback_in1, '0', 0, 0);
      -- wait for data and phase outputs to change
      guard_and(data_out, '1', phase_out, '0');
      wait for delay(200, 250);

      -- LATE FIRE
      -- assign data values
      assign(data_in1, '1', 0, 0);
      -- assign phase values
      assign(phase_in0, '1', 0, 0, phase_in1, '1', 0, 0, phase_in2, '1', 0, 0);
      -- assign feedback values
      assign(feedback_in0, '1', 0, 0, feedback_in1, '1', 0, 0);
      -- wait for data and phase outputs to change
      guard_and(data_out, '0', phase_out, '1');
      wait for delay(200, 250);

    end loop;
  end process;

end env_8;
```