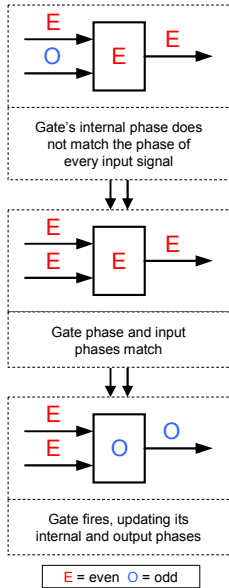


Phased Logic (PL)

- Asynchronous design methodology that supports automated translation of synchronous circuits
- Each signal is implemented using two wires: one for data and one for phase
- Each PL gate and PL signal has a phase, either even or odd
- When the phase of each 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 PL synthesis algorithm, 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
 - 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

PL Example



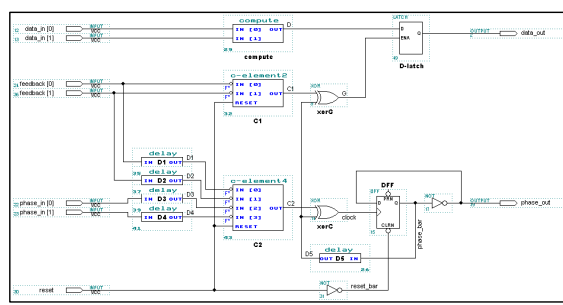
Coarse-Grained PL

- Rather than map each synchronous gate directly to a PL gate, enclose entire logic blocks in PL wrapper logic
- Logic blocks can be reused without modification
- Two types of logic blocks:
 - Barrier blocks, which contain D-Flip-Flops as well as combinational logic
 - Through blocks, which contain only combination logic
- Two speed-up techniques:
 - Time borrowing, which allows the control delay of a slow block to be spread to surrounding blocks
 - Early evaluation, which allows blocks in certain cases to compute their output before all of their inputs have arrived

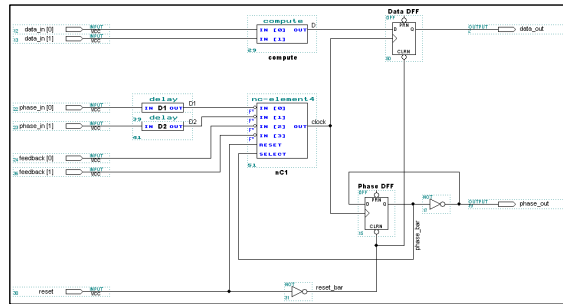
VHDL Modeling

- Existing VHDL models of the PL wrappers were unsuitable for use in ATACS
- New wrapper models were created using the subset of VHDL supported by ATACS
- An environment model was created for each of the wrappers; its purpose was to assign values to the input signals and define the expected behavior of the output signals
- The following assumptions were made:
 - Data, phase, and feedback input signals all arrive at the same time, which is the worst case scenario in terms of hazards
 - Large delay between wrapper firing and receiving new inputs, which is accurate if the wrapper is connected to at least one other block, as it would be in any real-world application

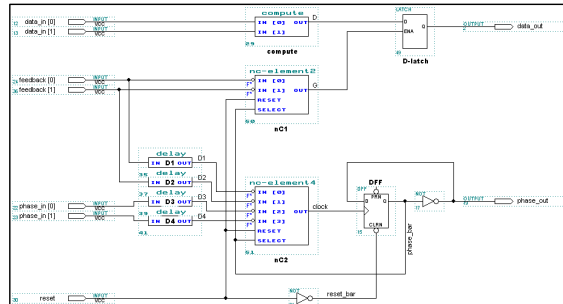
Wrapper Circuits



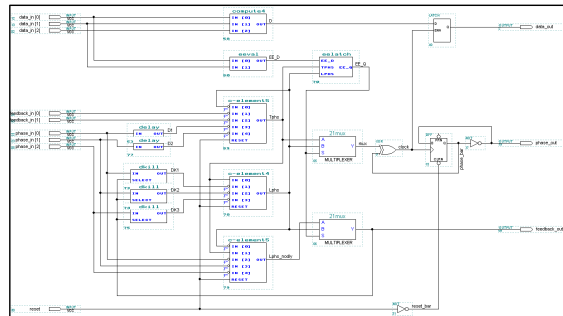
Wrapper for Through Blocks with Time Borrowing



Wrapper for Through Blocks without Time Borrowing



Wrapper for Barrier Blocks without Early Evaluation



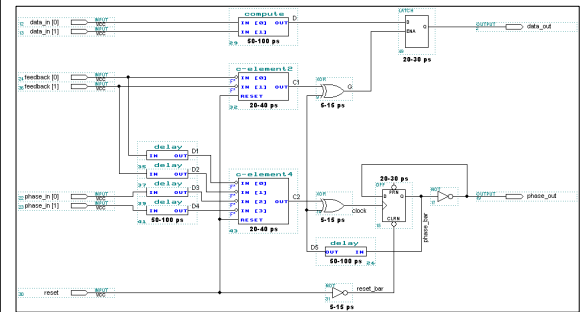
Early Evaluation Wrapper for Barrier and Through Blocks

ATACS

- Automatic Timed Asynchronous Circuit (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
- ATACS supports a wide range of file types, timing methods, and verification methods; the following were exclusively used in this research:
 - File type: VHDL
 - Timing method: POSETS
 - Verification method: hazard checking
- Hazard checking verifies that there are no hazards present in a circuit
- Hazards are "conditions generated by the structure of the circuit or timing relationships between inputs and propagation delays that can cause incorrect behavior" [6]
- Two types of hazards:
 - Acknowledgement 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" [6]
 - Monotonicity hazard: occurs when a signal "is supposed to remain stable but it becomes momentarily excited or it is supposed to make a transition which it makes non-monotonically" [6]
- ATACS automates the process of detecting hazards, allowing us to quickly check the hazard freedom of a given circuit

Verification Example

- Assume that the internal components of the first wrapper have the arbitrary delay ranges shown below:



- Running the VHDL model of this circuit through ATACS verifies that it is hazard free:
 - Selecting for each component any delay value within its range yields a hazard free circuit
 - Delay ranges are intentionally large to demonstrate that the circuit will behave properly within a wide variety of delay values
- Consider increasing the range of delay values for the compute block to 50-100 ps to 100-200 ps:
 - Now the delay through the compute block is long enough that the time from compute block output signal (D) being updated to the latch enable signal (G) going low is less than the latch's setup time
 - Running the modified circuit through ATACS yields the following error trace showing the sequence of signal transitions and indicating that the data_out signal is unable to go low:

```

reset+/-1 -> phase_in0+/-1 -> phase_in0-/-1 -> feedback0+/-1 -> feedback1+/-1 -> phase_out+/-1 ->
reset_bar_wrapper_4+/-1 -> c2_wrapper_4+/-1 -> c1_wrapper_4+/-1 -> d1_wrapper_4+/-1 -> d1_wrapper_4+/-1 ->
d2_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 -> s6+/-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 -> c2_wrapper_4+/-1 ->
d1_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 -> phase_in0+/-1 -> s3+/-1 -> data_in0+/-1 -> data_in1+/-1 -> phase_in1+/-1 ->
phase_in0+/-1 -> feedback0+/-1 -> feedback1+/-1 -> c1_wrapper_4+/-1 -> d1_wrapper_4+/-1 -> d2_wrapper_4+/-1 ->
d1_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!
    
```

- Similar errors can be achieved by shortening the delay through the C-element, the XOR gate, or the delay blocks.

Results

- By systematically modifying delay ranges for each wrapper and using ATACS to verify the hazard-freedom of the resulting circuits, we were able to establish the following relationships between the delays and hazards in each wrapper:

Wrapper for Through Blocks with Time Borrowing

```
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

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

Wrapper for Barrier Blocks without Early Evaluation

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

Early Evaluation Wrapper for Barrier and Through Blocks

Results Forthcoming

- 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
- In order to ensure the proper operation of an entire circuit, a designer must also adhere to the pre-existing PL timing constraints used in the current mapping process

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.

Acknowledgements

- Cherrice Traver, Union College
- Chris Meyers, University of Utah
- Bob Reese, Mississippi State University

References

- 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.
- M.E. Dean, T.E. Williams, and D.L. Dill, "Efficient Self-Timing with Level-Encoded 2-Phase Dual-Rail (LEDR)", *Advanced Research in VLSI*, 1991.
- Robert B. Reese, Mitchell A. Thornton, and Cherrice Traver, "A Course-Grain Phased Logic CPU", 9th IEEE International Symposium on Asynchronous Circuits and Systems (Asyn 2003), Vancouver, Canada.
- Sutherland, "Micropipelines", *Communications of the ACM*, Vol 32, No. 6, June 1989, pp. 720-738.
- Robert B. Reese, Mitchell A. Thornton, and Cherrice Traver, "Asyn 2004 Tutorial - Phased Logic", Workshop Notes, 10th International Symposium on Advanced Research in Asynchronous Circuits and Systems, Crete, Greece, April 2004.
- 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.
- Myers Research Group, "The ATACS User's Manual", University of Utah, <http://www.asyn.ece.utah.edu/tools/atacsman.html>
- Myers, Chris. *Asynchronous Circuit Design*. New York: Wiley, 2001.

Example VHDL Wrapper Model

```
-- Phased Logic wrapper for through blocks with time borrowing
-- (Figure 4 from "ASYN 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;
```

Example VHDL Environment Model

```
-- wrapper environment model 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;
```