COE 758 - Design Process Tutorial

I. Introduction

This tutorial describes a formal design process for the creation of digital systems. The aim of this design process is to provide a systematic approach to the design of digital components, beginning with a design specification, and ending with a working component which meets all specifications. The process being described limits itself to digital designs. Throughout the document, a simple example circuit is used to better illustrate each step.

The process begins with the introduction of Design Specifications, and their analysis. Based on these specifications, a Symbol for the component is created. This describes the system's inputs and outputs, and all interfaces to the surrounding environment. Once the symbol is complete, a block diagram is constructed, based on the required functionality of the system. The elements of the block diagram are then implemented as VHDL code, and integrated into a single design. This design is then implemented as a circuit and testing and verification can take place. Each stage will be described in more detail in the following sections. To recapitulate, the steps of the design process are:

1. Specification Analysis.
2. Symbol Creation.
3. Block Diagram and Behavioural Description Creation.
4. HDL Implementation.
5. System Integration.
6. Synthesis and circuit implementation.
7. Test and Verification.

II. Design Specifications

Design specifications are the starting point of any design. These are the requirements that the digital circuit being implemented must address. Such specifications come in two types: **Functional** and **Technical** specifications. **Functional specifications** describe the functionality that a given circuit must implement. They describe the required behaviour of the circuit. **Technical specifications** describe specific technical aspects associated with the circuit as well as surrounding systems. They augment the functional specifications, by accurately describing the parameters of the circuit behaviour.

EXAMPLE:

Consider the creation of a Reset Block, a circuit which monitors an external reset signal, and generates three auxiliary reset signals which will be used in other designs. This circuit example will be used as an illustrative example for the remainder of this document. Below are listed the functional specifications for the circuit.

Reset Block (Functional Specification):

1. Monitor external reset signal.
2. When external reset is asserted, generate three auxiliary reset signals for use inside local digital systems.
3. The three auxiliary signals must each stay asserted for a specific amount of time from the assertion of the external reset.
4. Once all auxiliary resets are de-asserted, the reset block must be ready to repeat the process.

Reset Block (Technical Specification):

1. The Reset Block is synchronized to the rest of the system via a clock with a frequency of 50 MHz.
2. The external reset signal stays asserted for 4 clock cycles.
3. Auxiliary reset signal rstAux1 must stay asserted for 30 clock cycles after the external reset is first asserted.
4. Auxiliary reset signal rstAux2 must stay asserted for 24 clock cycles after the external reset is first asserted.
5. Auxiliary reset signal rstAux3 must stay asserted for 16 clock cycles after the external reset is first asserted.

III. Symbol Creation

A component symbol gives a "black box" top-level view of the component being designed. The symbol diagram consists of a rectangle representing the component proper, with arrows representing all top level ports. Input ports are represented by arrows pointing towards the rectangle from the left, while output ports are represented as arrows pointing away from the rectangle, towards the right.

The component symbol is the first part of the design which is created; it is important that the symbol is as complete as possible before proceeding to other design steps. While an iterative process may be used in certain situations, it is best to try to clarify the majority of top-level ports before any other steps are taken. Once the component symbol is complete, it corresponds with the top-level Entity declaration for your designed component.

EXAMPLE:

The Reset Block receives one input: the external reset signal. It generates three outputs: the three internal reset signals. The Reset Block must be synchronized with the rest of the systems surrounding it, and receives a clock signals as an input as well. The external reset signal and the clock are single-bit signals. The three internal reset signals generated by the Reset Block are also single-bit signals. Based on this, the symbol for the Reset Block is shown below in Figure 1, with the component entity declaration listed after it.

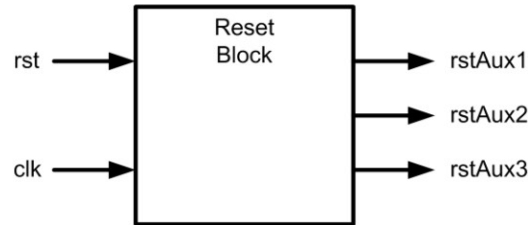

Figure 1: Example Symbol

```
entity resetBlock is
   Port (
      clk        : in std_logic;
      rst        : in std_logic;
      rstAux1    : out std_logic;
      rstAux2    : out std_logic;
      rstAux3    : out std_logic
   );
end resetBlock;
```

IV. Block Diagram Generation

Once the component symbol is created, the internal architecture of the component can be designed, in the form of a Block Diagram. The block diagram is a high-level description of a component's internal architecture. The individual blocks which make up the diagram are selected based on the required functionality of the component being designed. Blocks at this level can include adders/subtractors, decoders, multiplexers, storage elements of various types or finite-state machines; depending on the complexity of the component, the block diagram may be hierarchical, and include further, lower level block diagrams for some of the sub-components used.

In addition to the internal structure of the component, its behaviour must also be determined in detail, based on the design specifications. This behavioural description consists of two types of elements: **process diagrams** and **timing diagrams**. **Process diagrams** are abstract descriptions of the required behaviour, and may group together multiple activities into a single abstract block. **Timing diagrams** are clock-cycle accurate diagrams representing the required behaviour of the sequential circuits being used. The timing diagrams are used to create the state diagrams for the various FSMs present inside the component.

EXAMPLE:

The Reset Block must detect a particular level on an input signal. It must then wait a specific amount of time, and assert certain outputs at specific points in time. Based on these requirements, the design will incorporate an up-counter, used to count clock cycles, and a control unit in the form of a FSM. Finally, the component must detect when certain count values are reached, which implies comparators. Based on these requirements, the block diagram below can be derived. It consists of one FSM, a counter, and three comparators which indicate 16, 24 and 30 clock cycles have been counted.

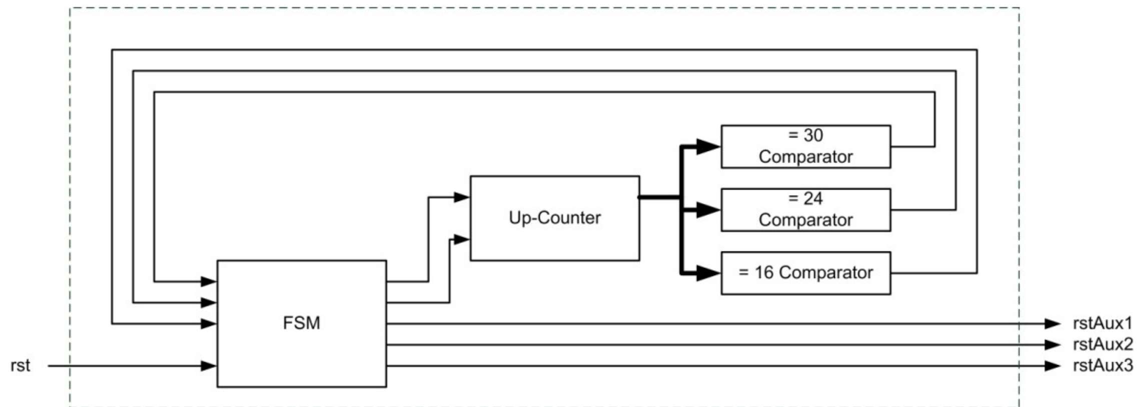

Figure 2: Example Block Diagram

The FSM must assert all three internal reset signals when it detects that the external reset is asserted, and must then de-assert the reset signals as the counter advances. At an abstract level, this behaviour can be split into two main activities: monitoring the external reset signal and generating the auxiliary reset signals, as shown in Figure 3 below. The timing diagram for this process is shown in Figure 4, and describes the required behaviour of the circuit on a clock cycle by clock cycle basis. To accomplish this behaviour, the FSM will be designed to have three states, in which the resets are asserted, one state where the counter is reset, and one rest state, where it waits for the next external reset. The state diagram for the FSM is shown in Figure 5.

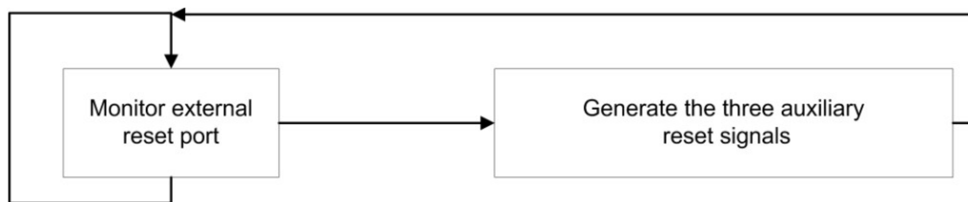

Figure 3: Example Process Diagram

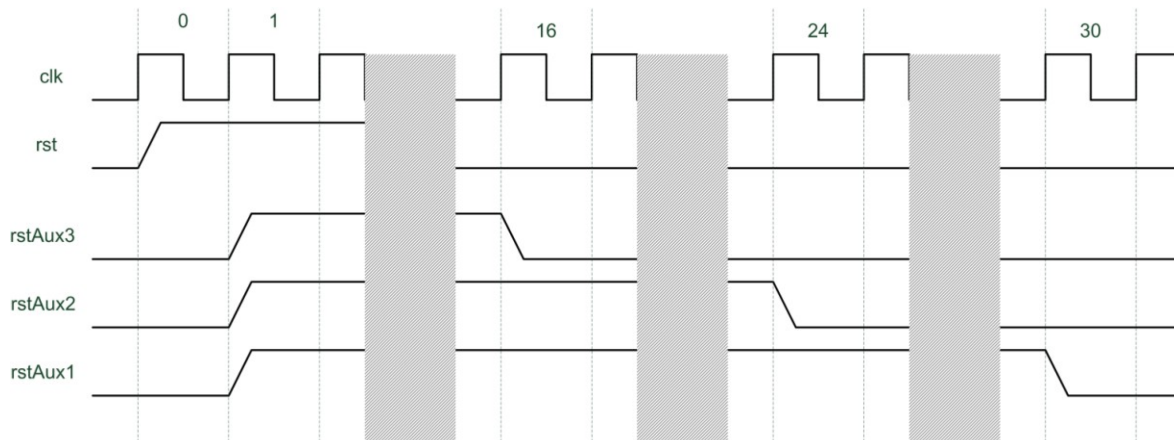

Figure 4: Example Timing Diagram

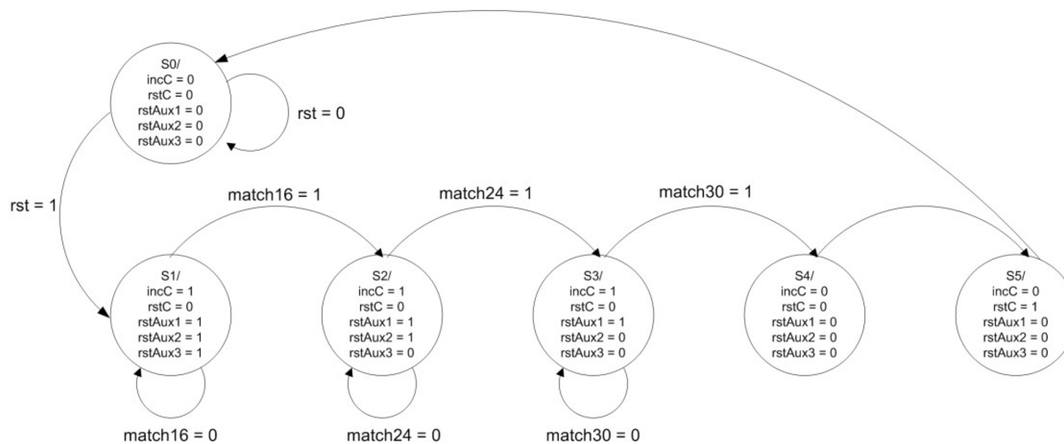

Figure 3: Example State Diagram

V. HDL Implementation

Once a complete block diagram is constructed, the circuit can be implemented using HDL; for the purposes of this tutorial, VHDL is used. Each block in the block diagram should be implemented, using either VHDL components, or processes; certain components, such as simple gates, can also be implemented using simple statements. At this point, a decision must be made as to whether each block should be implemented as a separate component or as a process. This decision is based on a number of parameters, including the size of the block and the complexity of the overall design.

EXAMPLE:

The Reset Block being designed is a simple system, so it may be implemented as a collection of processes. The primary processes being considered are the up-counter, the comparators, and the finite-state machine. Below, code examples are given for the FSM, one of the comparators and the up-counter.

Up-Counter:

```
upCounter: process(clk, rstC, incC)
begin
   if(rstC = '1')then
      countVal <= "00000";
   else
      if(clk'event and clk = '1' and incC = '1')then
         countVal <= countVal + "00001";
      end if;
   end if;
end process;
```

Comparator (pure combinational circuit):

```
comparator16 : process(countVal)
begin
   if(countVal = "10000")then
      match16 <= '1';
   else
      match16 <= '0';
   end if;
end process;
```

Comparator (registered output):

```
comparator16 : process(clk, countVal)
begin
   if(clk’event and clk = ‘1’)then
      if(countVal = “10000”)then
         match16 <= ‘1’;
      else
         match16 <= ‘0’;
      end if;
   end if;
end process;
```

FSM:

```
-- State storage.
stateStorage: process(clk, st1N)
begin
   if(clk'event and clk = '1')then
      st1 <= st1N;
   end if;
end process;

-- Next state generation.
nextStateGen : process(st1, rst, match16, match24, match30)
begin
   if(st1 = "000")then           -- S0: Rest state.
      if(rst = '1')then
         st1N <= "001";
      else
         st1N <= "000";
      end if;
   elsif(st1 = "001")then        -- S1: generate all three reset signals.
      if(match16 = '1')then
         st1N <= "010";
      else
         st1N <= "001";
      end if;
   elsif(st1 = "010")then        -- S2: generate only two reset signals.
      if(match24 = '1')then
         st1N <= "011";
      else
         st1N <= "010";
      end if;
   elsif(st1 = "011")then        -- S3: generate only one reset signal.
      if(match30 = '1')then
         st1N <= "100";
      else
         st1N <= "011";
      end if;
   elsif(st1 = "100")then        -- S4: de-assert all reset signals.
      st1N <= "101";
   elsif(st1 = "101")then        -- S5: reset up-counter.
      st1N <= "000";
   else
      st1N <= "000";
   end if;
end process;
```

```
-- Output generation.
outGen: process(st1)
begin
   if(st1 = "000")then          -- S0: Rest state.
      incC <= '0';
      rstC <= '0';
      rstAux1 <= '0';
      rstAux2 <= '0';
      rstAux3 <= '0';
   elsif(st1 = "001")then       -- S1: generate all three reset signals.
      incC <= '1';
      rstC <= '0';
      rstAux1 <= '1';
      rstAux2 <= '1';
      rstAux3 <= '1';
   elsif(st1 = "010")then       -- S2: generate only two reset signals.
      incC <= '1';
      rstC <= '0';
      rstAux1 <= '1';
      rstAux2 <= '1';
      rstAux3 <= '0';
   elsif(st1 = "011")then       -- S3: generate only one reset signal.
      incC <= '1';
      rstC <= '0';
      rstAux1 <= '1';
      rstAux2 <= '0';
      rstAux3 <= '0';
   elsif(st1 = "100")then       -- S4: de-assert all reset signals.
      incC <= '0';
      rstC <= '0';
      rstAux1 <= '0';
      rstAux2 <= '0';
      rstAux3 <= '0';
   elsif(st1 = "101")then       -- S5: reset up-counter.
      incC <= '0';
      rstC <= '1';
      rstAux1 <= '0';
      rstAux2 <= '0';
      rstAux3 <= '0';
   else
      incC <= '0';
      rstC <= '0';
      rstAux1 <= '0';
      rstAux2 <= '0';
      rstAux3 <= '0';
   end if;
end process;
```

VI. Design Integration

Once all block diagram blocks have been implemented, they must be connected together to create the complete design. The complete design will be assembled into a top-level VHDL file: the entity portion of this file will correspond to the symbol of the component, and the architecture portion will contain all blocks from the block diagram. Blocks implemented as discrete components are instantiated inside the design using the ***port map*** command. On the other hand, blocks implemented as processes are simply copied into this file. All blocks must be connected correctly using signals.

EXAMPLE:

The complete code for the Reset Block is listed below.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity resetBlock is
   Port (
      clk : in  STD_LOGIC;
      rst : in  STD_LOGIC;
      rstAux1 : out  STD_LOGIC;
      rstAux2 : out  STD_LOGIC;
      rstAux3 : out  STD_LOGIC
   );
end resetBlock;

architecture Behavioral of resetBlock is

   -- Up counter.
   signal countVal : std_logic_vector(4 downto 0) := "00000";
   signal rstC, incC : std_logic;

   -- Comparators.
   signal match16, match24, match30 : std_logic;

   -- FSM control unit.
   signal st1 : std_logic_vector(2 downto 0) := "000";
   signal st1N : std_logic_vector(2 downto 0);

begin

   -- System up-counter, used to keep track of time.
   upCounter: process(clk, rstC, incC)
   begin
      if(rstC = '1')then
         countVal <= "00000";
      else
         if(clk'event and clk = '1' and incC = '1')then
            countVal <= countVal + "00001";
         end if;
      end if;
   end process;

   -- Comparators, used to flag specific counter values.
   comparator16 : process(countVal)
   begin
      if(countVal = "10000")then
         match16 <= '1';
      else
         match16 <= '0';
      end if;
   end process;
```

```
comparator24 : process(countVal)
begin
   if(countVal = "11000")then
      match24 <= '1';
   else
      match24 <= '0';
   end if;
end process;

comparator30 : process(countVal)
begin
   if(countVal = "11110")then
      match30 <= '1';
   else
      match30 <= '0';
   end if;
end process;

-- FSM control unit, with 6 states.

-- State storage.
stateStorage: process(clk, st1N)
begin
   if(clk'event and clk = '1')then
      st1 <= st1N;
   end if;
end process;

-- Next state generation.
nextStateGen : process(st1, rst, match16, match24, match30)
begin
   if(st1 = "000")then           -- S0: Rest state.
      if(rst = '1')then
         st1N <= "001";
      else
         st1N <= "000";
      end if;
   elsif(st1 = "001")then        -- S1: generate all three reset signals.
      if(match16 = '1')then
         st1N <= "010";
      else
         st1N <= "001";
      end if;
   elsif(st1 = "010")then        -- S2: generate only two reset signals.
      if(match24 = '1')then
         st1N <= "011";
      else
         st1N <= "010";
      end if;
   elsif(st1 = "011")then        -- S3: generate only one reset signal.
      if(match30 = '1')then
         st1N <= "100";
      else
         st1N <= "011";
      end if;
   elsif(st1 = "100")then        -- S4: de-assert all reset signals.
      st1N <= "101";
   elsif(st1 = "101")then        -- S5: reset up-counter.
      st1N <= "000";
   else
      st1N <= "000";
   end if;
end process;
```

```
   -- Output generation.
   outGen: process(st1)
   begin
      if(st1 = "000")then          -- S0: Rest state.
         incC <= '0';
         rstC <= '0';
         rstAux1 <= '0';
         rstAux2 <= '0';
         rstAux3 <= '0';
      elsif(st1 = "001")then       -- S1: generate all three reset signals.
         incC <= '1';
         rstC <= '0';
         rstAux1 <= '1';
         rstAux2 <= '1';
         rstAux3 <= '1';
      elsif(st1 = "010")then       -- S2: generate only two reset signals.
         incC <= '1';
         rstC <= '0';
         rstAux1 <= '1';
         rstAux2 <= '1';
         rstAux3 <= '0';
      elsif(st1 = "011")then       -- S3: generate only one reset signal.
         incC <= '1';
         rstC <= '0';
         rstAux1 <= '1';
         rstAux2 <= '0';
         rstAux3 <= '0';
      elsif(st1 = "100")then       -- S4: de-assert all reset signals.
         incC <= '0';
         rstC <= '0';
         rstAux1 <= '0';
         rstAux2 <= '0';
         rstAux3 <= '0';
      elsif(st1 = "101")then       -- S5: reset up-counter.
         incC <= '0';
         rstC <= '1';
         rstAux1 <= '0';
         rstAux2 <= '0';
         rstAux3 <= '0';
      else
         incC <= '0';
         rstC <= '0';
         rstAux1 <= '0';
         rstAux2 <= '0';
         rstAux3 <= '0';
      end if;
   end process;

end Behavioral;
```

VII. Implementation

Once the complete design has been prepared, it can be synthesized, translated, mapped, placed and routed. It is also at this time that specific constraints are placed on the design, if necessary. The final result of this process, if FPGA implementation is being targeted, is a configuration bit-stream.

If ChipScope or some other form of integrated analysis and test process is sued, it must be incorporated into the design at this stage. This can be done by instantiating the test circuits inside the component, or by creating a separate top-level test file, and instantiating both the test circuits and the component under test there. Internal signals from the component being tested can be brought out through debug ports, to be monitored by the test circuits.

EXAMPLE:

To test the Reset Block, a new top-level test VHDL file was created, and the Reset Block, ICON, ILA and VIO components were instantiated there. To verify the internal behaviour of the Reset Block, the state registers, counter, and comparator match signals were brought out and connected to the ILA. The complete code for the top-level test file, along with the updated Reset Block (with debug ports) is listed in Appendix A.

VIII. Test and Verification

Once the complete design has been implemented, it must be tested, to ensure the correct behaviour was achieved, as well as to verify that the technical parameters were met. Such testing can take any number of forms, and can rely on any number of tools, including automated test pattern generators, oscilloscopes, logic analyzers or integrated logic analyzers (ChipScope).

EXAMPLE:

The Reset Block was tested using ChipScope, as described in the previous section. The ILA was used to monitor the state, counter and internal signals of the Reset Block, as well as its outputs and inputs. The VIO was used to generate an external reset signal for the Reset Block.

NOTE: the testing process used violates one of the technical specifications, in that the external reset signal remains asserted for over 200 clock cycles (due to the speed limitations of the VIO), meaning that the Reset Block is triggered repeatedly. However, this does not affect the behaviour of the circuit-under-test.

Figures 6 and 7 show that the circuit works correctly. Figure 6 shows the auxiliary reset signals being asserted for the correct periods of time. It also shows the FSM performing the correct state transitions. Figure 7 shows a more detailed view of the up-counter, and also shows the assertion of the match indicator signals on the appropriate count values.

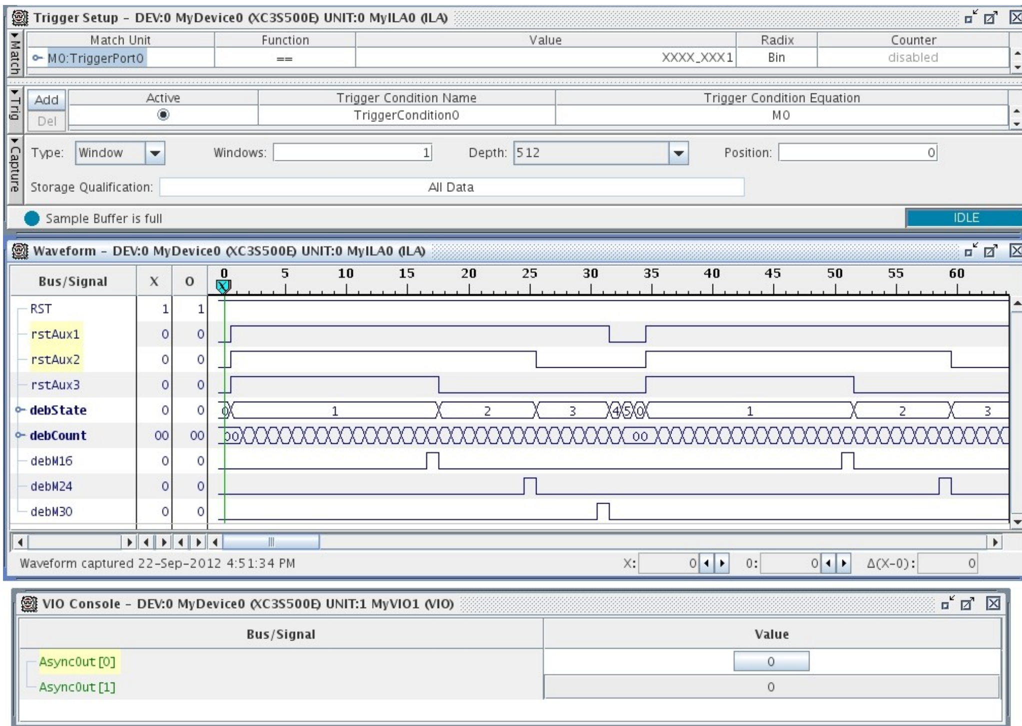

Figure 4: Reset Block Test Results

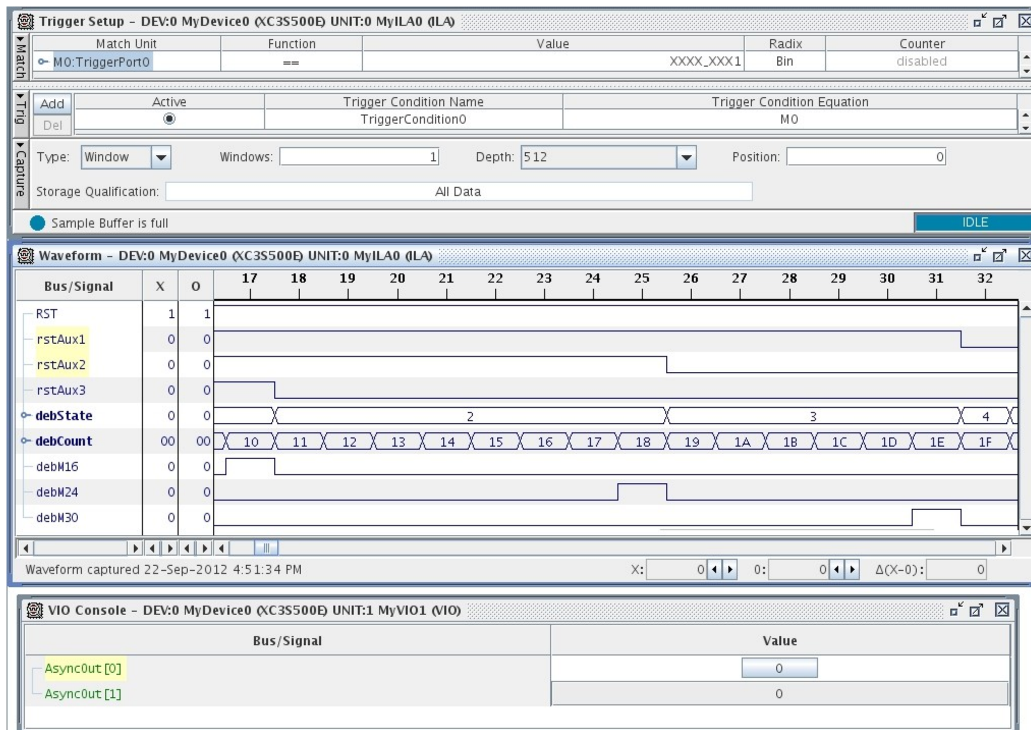

Figure 5: Detailed View of Up-Counter and Comparators

Appendix A – Reset Block Test VHDL Code

Updated Reset Block, with added debug ports.

```
----------------------------------------------------------------------------------
-- Company: 
-- Engineer: 
-- 
-- Create Date:    15:39:41 09/22/2012 
-- Design Name: 
-- Module Name:    resetBlock - Behavioral 
-- Project Name: 
-- Target Devices: 
-- Tool versions: 
-- Description: 
--
-- Dependencies: 
--
-- Revision: 
-- Revision 0.01 - File Created
-- Additional Comments: 
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity resetBlock is
   Port (
      clk : in  STD_LOGIC;
      rst : in  STD_LOGIC;
      rstAux1 : out  STD_LOGIC;
      rstAux2 : out  STD_LOGIC;
      rstAux3 : out  STD_LOGIC;

      -- Debug
      debSt    : out std_logic_vector(2 downto 0);
      debC     : out std_logic_vector(4 downto 0);
      debM16   : out std_logic;
      debM24   : out std_logic;
      debM30   : out std_logic
   );
end resetBlock;

architecture Behavioral of resetBlock is

   -- Up counter.
   signal countVal : std_logic_vector(4 downto 0) := "00000";
   signal rstC, incC : std_logic;

   -- Comparators.
   signal match16, match24, match30 : std_logic;

   -- FSM control unit.
   signal st1 : std_logic_vector(2 downto 0) := "000";
   signal st1N : std_logic_vector(2 downto 0);

begin
```

```
-- System up-counter, used to keep track of time.
upCounter: process(clk, rstC, incC)
begin
   if(rstC = '1')then
      countVal <= "00000";
   else
      if(clk'event and clk = '1' and incC = '1')then
         countVal <= countVal + "00001";
      end if;
   end if;
end process;

debC <= countVal;

-- Comparators, used to flag specific counter values.
comparator16 : process(countVal)
begin
   if(countVal = "10000")then
      match16 <= '1';
   else
      match16 <= '0';
   end if;
end process;

debM16 <= match16;

comparator24 : process(countVal)
begin
   if(countVal = "11000")then
      match24 <= '1';
   else
      match24 <= '0';
   end if;
end process;

debM24 <= match24;

comparator30 : process(countVal)
begin
   if(countVal = "11110")then
      match30 <= '1';
   else
      match30 <= '0';
   end if;
end process;

debM30 <= match30;

-- FSM control unit, with 6 states.

-- State storage.
stateStorage: process(clk, st1N)
begin
   if(clk'event and clk = '1')then
      st1 <= st1N;
   end if;
end process;

debSt <= st1;

-- Next state generation.
nextStateGen : process(st1, rst, match16, match24, match30)
begin
   if(st1 = "000")then           -- S0: Rest state.
      if(rst = '1')then
         st1N <= "001";
      else
         st1N <= "000";
      end if;
   elsif(st1 = "001")then        -- S1: generate all three reset signals.
      if(match16 = '1')then
         st1N <= "010";
```

```
         else
            st1N <= "001";
         end if;
      elsif(st1 = "010")then         -- S2: generate only two reset signals.
         if(match24 = '1')then
            st1N <= "011";
         else
            st1N <= "010";
         end if;
      elsif(st1 = "011")then         -- S3: generate only one reset signal.
         if(match30 = '1')then
            st1N <= "100";
         else
            st1N <= "011";
         end if;
      elsif(st1 = "100")then         -- S4: de-assert all reset signals.
         st1N <= "101";
      elsif(st1 = "101")then         -- S5: reset up-counter.
         st1N <= "000";
      else
         st1N <= "000";
      end if;
   end process;

   -- Output generation.
   outGen: process(st1)
   begin
      if(st1 = "000")then            -- S0: Rest state.
         incC <= '0';
         rstC <= '0';
         rstAux1 <= '0';
         rstAux2 <= '0';
         rstAux3 <= '0';
      elsif(st1 = "001")then         -- S1: generate all three reset signals.
         incC <= '1';
         rstC <= '0';
         rstAux1 <= '1';
         rstAux2 <= '1';
         rstAux3 <= '1';
      elsif(st1 = "010")then         -- S2: generate only two reset signals.
         incC <= '1';
         rstC <= '0';
         rstAux1 <= '1';
         rstAux2 <= '1';
         rstAux3 <= '0';
      elsif(st1 = "011")then         -- S3: generate only one reset signal.
         incC <= '1';
         rstC <= '0';
         rstAux1 <= '1';
         rstAux2 <= '0';
         rstAux3 <= '0';
      elsif(st1 = "100")then         -- S4: de-assert all reset signals.
         incC <= '0';
         rstC <= '0';
         rstAux1 <= '0';
         rstAux2 <= '0';
         rstAux3 <= '0';
      elsif(st1 = "101")then         -- S5: reset up-counter.
         incC <= '0';
         rstC <= '1';
         rstAux1 <= '0';
         rstAux2 <= '0';
         rstAux3 <= '0';
      else
         incC <= '0';
         rstC <= '0';
         rstAux1 <= '0';
         rstAux2 <= '0';
         rstAux3 <= '0';
      end if;
   end process;

end Behavioral;
```

Top-level test file.

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    16:00:53 09/22/2012
-- Design Name:
-- Module Name:    topLevelTest - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity topLevelTest is
   Port (
      clk : in  STD_LOGIC
   );
end topLevelTest;

architecture Behavioral of topLevelTest is

   -- Component under test.
   component resetBlock
   port (
      clk : in  STD_LOGIC;
      rst : in  STD_LOGIC;
      rstAux1 : out  STD_LOGIC;
      rstAux2 : out  STD_LOGIC;
      rstAux3 : out  STD_LOGIC;

      -- Debug
      debSt    : out std_logic_vector(2 downto 0);
      debC     : out std_logic_vector(4 downto 0);
      debM16   : out std_logic;
      debM24   : out std_logic;
      debM30   : out std_logic
   );
   end component;

   -- Component outputs.
   signal rstA1, rstA2, rstA3 : std_logic;

   -- Debug outputs.
   signal debSt : std_logic_vector(2 downto 0);
   signal debC : std_logic_vector(4 downto 0);
   signal debM16, debM24, debM30 : std_logic;

   -- Test circuitry.

   -- ICON.
```

```
   component sysIcon
   PORT (
       CONTROL0 : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0);
       CONTROL1 : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0)
   );
   end component;

   signal control0, control1 : std_logic_vector(35 downto 0);

   -- ILA.
   component sysIla
   PORT (
      CONTROL : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0);
      CLK : IN STD_LOGIC;
      DATA : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
      TRIG0 : IN STD_LOGIC_VECTOR(7 DOWNTO 0)
   );
   end component;

   signal ilaData : std_logic_vector(15 downto 0);
   signal ilaTrig : std_logic_vector(7 downto 0);

   -- VIO
   component sysVio
   PORT (
      CONTROL : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0);
      ASYNC_OUT : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
   );
   end component;

   signal vioOut : std_logic_vector(1 downto 0);

begin

   -- Component under test.
   compUnderTest: resetBlock
   port map(
      clk => clk,
      rst => vioOut(0),
      rstAux1 => rstA1,
      rstAux2 => rstA2,
      rstAux3 => rstA3,
      -- Debug
      debSt => debSt,
      debC => debC,
      debM16 => debM16,
      debM24 => debM24,
      debM30 => debM30
   );

   -- Test circuits.

   -- ICON.
   systemIcon : sysIcon
   port map (
      CONTROL0 => control0,
      CONTROL1 => control1
   );

   -- ILA.
   systemIla : sysIla
   port map (
      CONTROL => control0,
      CLK => clk,
      DATA => ilaData,
      TRIG0 => ilaTrig
   );

   -- VIO.
   systemVio : sysVio
   port map (
      CONTROL => control1,
      ASYNC_OUT => vioOut
```

```
   );

   -- Debug signal assignments.

   -- Normal inputs and outputs.
   ilaData(0) <= vioOut(0);
   ilaData(1) <= rstA1;
   ilaData(2) <= rstA2;
   ilaData(3) <= rstA3;

   -- Debug outputs.
   ilaData(6 downto 4) <= debSt;
   ilaData(11 downto 7) <= debC;
   ilaData(12) <= debM16;
   ilaData(13) <= debM24;
   ilaData(14) <= debM30;

   ilaData(15) <= '0';

   -- Trigger assignment.
   ilaTrig(0) <= vioOut(0);
   ilaTrig(7 downto 1) <= (others => '0');

end Behavioral;
```