CDA 4253/CIS 6930 FPGA System Design
Sequential Circuit Building Blocks

Hao Zheng
Dept of Comp Sci & Eng
USF
Outline and Reading

→ Introduction of behavioral modeling with process
  → Sequential statements inside processes
→ Modeling combinational/sequential circuits
  → Using processes and sequential statements
→ Reading — P. Chu, P. Chu, *FPGA Prototyping by VHDL Examples*
  → *Section 3.3 – 3.4, 3.7*
  → *Chapter 4, Regular Sequential Circuit*
VHDL Modeling Styles

VHDL Descriptions

Dataflow
- Concurrent statements

structural
- Components and interconnects

behavioral
- Sequential statements
  - Registers
  - State machines
  - Instruction decoders

Subset most suitable for synthesis

• Testbenches
Processes and Sequential Statements
Process Statements

→ Processes are concurrent statements.
→ Processes describe combinational/sequential behavior
→ Processes in VHDL are very powerful statements
  → Allow to define arbitrary behavior that may be difficult to represent by a real circuit
  → Not every process can be synthesized
→ Use processes with caution in order to write the synthesizable code.
→ Use processes freely in testbenches for simulation.
Concurrent Statements

→ simple concurrent signal assignment
   (↔)

→ conditional concurrent signal assignment
   (when-else)

→ selected concurrent signal assignment
   (with-select-when)

→ Processes
Anatomy of a Process

OPTIONAL

[label:] process [(sensitivity list)]
[declaration part]
begin
  sequential statements
end process [label];
Sequential Statements

→ Used only in processes

→ Evaluated one at a time sequentially

→ Include
  → Signal/variable assignments
  → IF statements
  → CASE statements
  → WAIT statements
  → Other – loop, ...
PROCESS with a SENSITIVITY LIST

- List of signals to which the process is sensitive.
- Whenever there is an event on any of the signals in the sensitivity list, the process fires.
- Every time the process fires, it will run in its entirety.

**label:**
```
process (sensitivity list)
  declarations
  begin
    sequential statements
  end
end process;
```

*wait* statements are NOT ALLOWED in a process with a sensitivity list.
Processes – Semantics

1. An event on a sensitive signal occurs, or
2. Certain amount of delay has passed

1. Finish sequential execution, or
2. encounter wait
Modeling Combinational Circuits
To model a combinational circuit, all input signals and signals on LHS of signal assignments must be included in the sensitivity list!
priority: process
begin
    -- sensitivity list used
    -- wait statement
    wait on a, b;
    c <= a and b;
end process;

To model a combinational circuit, all input signals and signals on LHS of signal assignments must be included in the sensitivity list!
Wrong Processes

priority: process (a, b) begin
  wait on a, b;
  c <= a and b;
end process;

*wait statements are NOT ALLOWED in a process with a sensitivity list.*
Sequential Statement: wait

wait until boolean_condition;
-- wait until a='1' and b='0';

wait on signal_list;
-- wait on a, b;

wait for time;
-- wait for 10 ns;

wait; -- suspend the process forever

Synthesis – Do not use WAIT when modeling a design.
Process for Conditional Concurrent Signal Assignment

architecture arch of ex is
begin
    r <= a + b + c when m=n else a - b when m>0 else c + 1;
end architecture arch;
Process for Conditional Concurrent Signal Assignment – IF Statement

architecture arch of ex is
begin
    process (a, b, c, m, n) begin
        if m=n then
            r <= a + b + c;
        elsif m > 0 then
            r <= a - b;
        else
            r <= c + 1;
        end if;
    end process;
end architecture arch;
If Statement – Syntax

```plaintext
if boolean_expr_1 then
    sequential_statements;
elsif boolean_expr_2 then
    sequential_statements;
elsif boolean_expr_3 then
    sequential_statements;
    ...
else
    sequential_statements;
end if;
```
architecture arch of ex is
begin
  with sel select
    r <= a + b + c when "00" else
    a - b     when "10" else
    c + 1     when others;
end architecture arch;
architecture arch of ex is
begin
    process(a, b, c, sel)
    begin
        case sel is
            when "00" =>
                r <= a + b + c;
            when "10" =>
                r <= a - b;
            when others =>
                r <= c + 1;
        end case;
    end process;
end architecture arch;
Case Statement – Syntax

```plaintext
case case_expression is
  when choice_1 =>
    sequential statements;
  when choice_2 =>
    sequential statements;
  . . .
  when choice_n =>
    sequential statements;
end case;
```
Comparison to Concurrent Statements

large <= a when a > b else b;
small <= b when a > b else a;

process(a, b)
begin
  if a > b then
    large <= a;
    small <= b;
  else
    large <= b;
    small <= a;
  end if;
end process;
Unintended Memory

```
process(a)
begin
  if a > b then
    gt <= '1';
  elsif a = b then
    eq <= '1';
  end if;
end process;
```

Implicit memory is introduced to hold \texttt{gt} and \texttt{eq}.

To model a \textit{combinational} circuit, all input signals and signals on LHS of signal assignments must be included in the sensitivity list!
Avoid Unintended Memory

process(a, b)
begin
  if a > b then
    gt <= '1';
    eq <= '0';
  elsif a = b then
    gt <= '0';
    eq <= '1';
  else
    gt <= '0';
    eq <= '0';
  end if;
end process;

1. All inputs and signals on LHS of signal assignments are included in the sensitivity list.
2. A signal must be assigned in every branch.
Signal Assignments in Processes

-- Is this process the -- right model?
process (d1, d2, d3) begin
    s <= d1 xor d2;
    o <= s and d3;
end process;
Signal Assignments in Processes

-- Is this process the -- right model?
process (d1, d2, d3) begin
  s <= d1 xor d2;
  o <= s and d3;
end process;
Signal Assignments in Processes

-- This process is the -- right model.

process (d1, d2, d3, s)
begin
    s <= d1 xor d2;
    o <= s and d3;
end process;

---

<table>
<thead>
<tr>
<th>d1</th>
<th>d2</th>
<th>d3</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Signal Assignments in Processes

process (d1, d2, d3, s)
begin
    s <= d1 xor d2;
    o <= s and d3;
end process;

process (d1, d2, d3, s)
begin
    o <= s and d3;
    s <= d1 xor d2;
end process;

0

from: 0ps
to: 11ps

get signals
radix ! # 100% $ %
Comb. Design Ex: Find Max of a, b, c
Modeling Sequential Circuits
Processes Modeling Sequential Circuits

```vhdl
priority: process(clk)
begin
  if w(3) = '1' then
    y <= "11" ;
  elsif w(2) = '1' then
    y <= "10" ;
  elsif w(1) = '1' then
    y <= "01" ;
  else
    y <= "00" ;
  end if;
end process;
```

• All signals which appear on the left of signal assignment statement (<=) are outputs e.g. y, z
• All signals which appear on the sensitivity list are inputs e.g. clk
• All signals which appear on the right of signal assignment statement (<=) or in logic expressions are inputs e.g. w, a, b, c
• Note that not all inputs need to be included on the sensitivity list

Priority Encoder

-- Not synthesizable!!
D-Latch

Graphical symbol

Truth table

<table>
<thead>
<tr>
<th>Clock</th>
<th>D</th>
<th>$Q(t+1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>$Q(t)$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Timing diagram
D-Latch: Level Sensitive

-- library not shown
entity latch is
    port ( D, Clock : in STD_LOGIC ;
    Q : out STD_LOGIC);
end latch ;

architecture behavioral of latch is begin
    process (D, Clock)
    begin
        if Clock = '1' then
            Q <= D ;
        end if;
    end process;
end architecture behavioral;

Should be avoided!
Synchronous Circuit Structure

Combination Logic

FF

clock
process (reset, clk) begin
  if reset='1' then
    -- reset logic for memory
  elsif rising_edge(clk) then
    -- functional logic defined
    -- with sequential statements
  end if;
end process;

This template is for synthesizable synchronous designs
process modeling sequential circuits – code template

```vhdl
process (clk)
begin
    if rising_edge (clk) then
        if reset='1' then
            -- reset logic for memory
        else
            -- functional logic defined
            -- with sequential statements
        end if;
    end if;
end if;
end process;
```

This template is for synthesizable synchronous designs
D Flip-Flop: Edge Triggered

Graphical symbol

Truth table

<table>
<thead>
<tr>
<th>Clk</th>
<th>D</th>
<th>Q(t+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>↑</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>↑</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>−</td>
<td>Q(t)</td>
</tr>
<tr>
<td>1</td>
<td>−</td>
<td>Q(t)</td>
</tr>
</tbody>
</table>

Timing diagram
D Flip-Flop: Edge Triggered

```vhdl
-- library not shown
entity dff is
    port( D, Clock : in STD_LOGIC;
         Q      : out STD_LOGIC);
end entity dff;

architecture behavioral of dff is
begin
    process(Clock)
    begin
        if rising_edge(Clock) then
            Q <= D;
        end if;
    end process;
end architecture behavioral;
```
D Flip-Flop: Edge Triggered

-- library not shown
entity dff is
    port ( D, Clock : in STD_LOGIC;
           Q    : out STD_LOGIC);
end entity dff;

architecture behavioral of dff is
begin
    process(Clock)
    begin
        if Clock'event and Clock='1' then
            Q <= D;
        end if;
    end process;
end architecture behavioral;
D Flip-Flop: Edge Triggered

-- library not shown
entity dff is
  port( D, Clock : in  STD_LOGIC ;
        Q    : out STD_LOGIC ) ;
end entity dff;

architecture behavioral of dff is begin
  process(Clock)
  begin
    if falling_edge(Clock) then
      Q <= D ;
    end if;
  end process;
end architecture behavioral;

To be synthesizable, each process is sensitive
on a single clock edge
D Flip-Flop: Not Allowed!

-- library not shown
entity dff is
  port( D, Clock : in STD_LOGIC;
       Q      : out STD_LOGIC);
end entity dff;

architecture behavioral of dff is begin
  process(Clock)
  begin
    if rising_edge(Clock) then
      Q <= D;
    elsif falling_edge(Clock) then
      Q <= not D;
    end if;
  end process;
end architecture behavioral;
D FF with Asynchronous Reset

-- library not shown
entity dff is
  port ( D, clock : in STD_LOGIC;
         resetn : in std_logic;
         Q     : out STD_LOGIC);
end entity dff;

architecture behavioral of dff is
begin
  process(resetn, clock)
  begin
    if resetn = '0' then
      Q <= '0';
    elsif rising_edge(clock) then
      Q <= D;
    end if;
  end process;
end architecture behavioral;
D FF with Synchronous Reset

-- library not shown
entity dff is
  port ( D, clock : in STD_LOGIC;
         resetn : in std_logic;
         Q : out STD_LOGIC);
end entity dff;
architecture behavioral of dff is begin
  process(clock)
  begin
    if rising_edge(clock) then
      if resetn = '0' then
        Q <= '0';
      else
        Q <= D;
      end if;
    end if;
  end process;
end behavioral;
8-Bit Register

--Library not shown
ENTITY reg8 IS
  PORT ( D : in STD_LOGIC_VECTOR(7 DOWNTO 0);
       resetn, clock : in STD_LOGIC ;
       Q : out STD_LOGIC_VECTOR(7 DOWNTO 0));
END reg8 ;

ARCHITECTURE behavioral OF reg8 IS
BEGIN
  PROCESS (resetn, clock)
  BEGIN
    IF resetn = '0' THEN
      Q <= "00000000";
    ELSIF rising_edge(clock) THEN
      Q <= D ;
    END IF ;
  END PROCESS ;
END behavioral ;

Reg8

Resetn

Clock

D

Q
Generic $N$-bit Register

```vhdl
--library not shown
ENTITY regn IS
    GENERIC (N : INTEGER := 16);
    PORT( D : in STD_LOGIC_VECTOR(\text{N-1 downto 0});
         Resetn, Clock : in STD_LOGIC;
         Q : out STD_LOGIC_VECTOR(\text{N-1 downto 0}));
END regn;

ARCHITECTURE behavioral OF regn IS
BEGIN
    PROCESS (Resetn, Clock)
    BEGIN
        IF Resetn = '0' THEN
            Q <= (others => '0');
        ELSIF rising_edge(Clock) THEN
            Q <= D;
        END IF;
    END PROCESS;
END behavioral;
```

![Generic N-bit Register](image)
Use of Keyword OTHERS

others stands for any index value that has not been previously referenced to.

Q <= “00000001” can be written as

Q <= (0 => ‘1’, others => ‘0’)  

Q <= “10000001” can be written as

Q <= (7 => ‘1’, 0 => ‘1’, others => ‘0’)  

or

Q <= (7 | 0 => ‘1’, others => ‘0’)  

Q <= “00011110” can be written as

Q <= (4 downto 1 => ‘1’, others => ‘0’)
Example

\[ x \leq \text{“0000_0001_0000_0001”} \]
other way to write the bit string?
N-bit Register with Enable

```vhdl
-- library not shown
ENTITY regne IS
  GENERIC (N : INTEGER := 8);
  PORT ( Enable, Clock : IN STD_LOGIC;
         D : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
         Q : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END regne;

ARCHITECTURE behavioral OF regne IS
BEGIN
  PROCESS (Clock)
  BEGIN
    IF rising_edge(Clock) THEN
      IF Enable = '1' THEN
        Q <= D;
      END IF;
    END IF;
  END PROCESS;
END behavioral;
```

![Diagram of a register](image)
FF Coding Guidelines (Xilinx)

→ Do not set or reset FFs asynchronously
  → Such FFs not supported, or
  → Requiring additional resources -> hurts performance

→ Do not describe Flip-Flops with both a set and a reset
  → No FFs with both set and reset

→ Always describe **enable**, **set**, or **reset** control inputs of Flip-Flop primitives as active-High
  → additional inverter may hurt performance

→ See XST user guide for more information
Shift Registers
ENTITY sr4 IS
  PORT (Enable: IN  STD_LOGIC;
        Sin   : IN  STD_LOGIC;
        Clock : IN  STD_LOGIC;
        Q     : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END sr4;
ARCHITECTURE behavioral OF sr4 IS
  signal reg: STD_LOGIC_VECTOR(3 downto 0);
BEGIN
  PROCESS (Clock)
  BEGIN
    IF rising_edge(Clock) THEN
      IF Enable = ‘1’ THEN
        -- shift right
        reg <= reg(3 downto 1) & sin;
        END IF;
      END IF;
  END PROCESS;
  -- output logic
  Q <= reg;
END behavioral;
Shift Register With Parallel Load

Load
D(3)
Sin
D(2)

D Q

D(1)

D Q

D(0)

D Q

Q(3) Q(2) Q(1) Q(0)

Enable
Clock
4-bit Shift Register with Parallel Load

ENTITY sr4_pl IS
  PORT (D : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
       Enable: IN STD_LOGIC;
       Load : IN STD_LOGIC;
       Sin  : IN STD_LOGIC;
       Clock : IN STD_LOGIC;
       Q : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END sr4_pl;
ARCHITECTURE behavioral OF sr4_pl IS
  SIGNAL reg : STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
  PROCESS (Clock)
  BEGIN
    IF rising_edge(Clock) THEN
      IF Enable = '1' THEN
        IF Load = '1' THEN
          reg <= D; -- parallel load
        ELSE
          reg <= reg(3 downto 1) & sin; -- shift right
        END IF;
      END IF;
    END IF;
  END PROCESS;
  Q <= reg; -- output logic
END behavioral;
ENTITY srn_pl IS
   GENERIC ( N : INTEGER := 8);
   PORT(D : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
        Enable : IN STD_LOGIC;
        Load   : IN STD_LOGIC;
        Sin    : IN STD_LOGIC;
        Clock  : IN STD_LOGIC;
        Q      : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END srn_pl;

N-bit Shift Register with Parallel Load
ARCHITECTURE behavioral OF shiftn IS
    signal reg: STD_LOGIC_VECTOR(N-1 downto 0);
BEGIN
    PROCESS (Clock)
    BEGIN
        IF rising_edge(Clock) THEN
            IF Enable = '1' THEN
                IF Load = '1' THEN
                    reg <= D;
                ELSE
                    reg <= reg(N-1 downto 1) & sin;
                END IF;
            END IF;
        END IF;
    END PROCESS;
    Q <= reg;
END behavioral;
Counters
architecture behavioral of upcount is

    SIGNAL Count : std_logic_vector(1 DOWNTO 0);

begin
    process(Clock)
    begin
        if rising_edge(Clock) then
            IF Clear = '1' THEN
                Count <= "00";
            ELSE
                Count <= Count + 1;
            END IF;
        end if;
    end process;
    Q <= Count;  -- output internal state
end behavioral;
ARCHITECTURE behavioral OF upcount _ar IS
  SIGNAL Count : STD_LOGIC_VECTOR (3 DOWNTO 0);
BEGIN
  PROCESS (Clock, Resetn)
  BEGIN
    IF Resetn = '0' THEN
      Count <= "0000";
    ELSIF rising_edge(Clock) THEN
      IF Enable = '1' THEN
        Count <= Count + 1;
      END IF;
    END IF;
  END PROCESS;
  Q <= Count;
END behavioral;
N-bit Generic Counter: Exercise

Modify the model below to make it generic

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY upcount_ar IS

PORT (Clock, Resetn, Enable :
     : IN   STD_LOGIC;
     Q : OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
END upcount_ar;
N-bit Generic Counter: Exercise

ARCHITECTURE behavioral OF upcount _ar IS
    SIGNAL Count : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
BEGIN
    PROCESS (Clock, Resetn)
    BEGIN
        IF Resetn = '0' THEN
            Count <= "0000";
        ELSIF rising_edge(Clock) THEN
            IF Enable = '1' THEN
                Count <= Count + 1 ;
            END IF ;
        END IF ;
    END PROCESS ;
    Q <= Count;
END behavioral ;
A Timer that Outputs a Tick per Second
A Timer that Outputs a Tick per Second

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY timer_1s IS
    PORT(
        Clock, Resetn : IN STD_LOGIC_VECTOR(26 DOWNTO 0);
        tick : OUT STD_LOGIC
    );
END timer_1s;
ARCHITECTURE behavioral OF timer_1s IS
  SIGNAL Count : unsigned(26 DOWNTO 0);
BEGIN
  PROCESS (Clock, Resetn)
  BEGIN
    IF Resetn = '0' THEN
      Count <= (others => '0');
    ELSIF rising_edge(Clock) THEN
      if Count = 100000000 then
        Count <= 0;
      else
        Count <= Count + 1 ;
      END IF ;
    END PROCESS;
    tick <= '1' when Count = 100000000 else '0';
  END PROCESS;
END behavioral ;
**Mod-m Counter**

Counts from 0 to \( m-1 \), and wraps around

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY mod_m_counter IS
    generic(
        N : integer : 4; -- number of bits
        M : integer : 10  -- mod-M
    );
    PORT(
        Clock, Reset : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
        q : out std_logic_vector(N-1 downto 0);
        tick : OUT STD_LOGIC
    );
END mod_m_counter;
```
ARCHITECTURE arch OF mod_m_counter IS
  BEGIN
  PROCESS (Clock, Reset)
  BEGIN
  IF Reset = '0' THEN
    reg <= (others => '0');
  ELSIF rising_edge(Clock) THEN
    if (reg = M-1) then
      reg <= (others => '0');
    else
      reg <= reg+1;
    end if;
  END IF ;
  END PROCESS ;
  q <= std_logic_vector(reg);
  tick <= '1' when reg = M-1 else '0';
  END arch;
Another Timer Design

- User sets a time in seconds
- Outputs a tick when time is expired
Mixing Description Styles
Inside of an Architecture
VHDL Description Styles

- **dataflow**
  - Concurrent statements

- **structural**
  - Components and interconnects

- **behavioral**
  - Sequential statements
    - Registers
    - Shift registers
    - Counters
    - State machines

synthesizeable
Mixed Style Modeling

architecture ARCHITECTURE_NAME of ENTITY_NAME is
  -- Declarations: signals, constants, functions,
  -- procedures, component declarations, ...
begin
  -- Concurrent statements:
  -- Concurrent simple signal assignment
  -- Conditional signal assignment
  -- Selected signal assignment

  -- Component instantiation statements

  -- Process statement
    -- inside process you can use
    -- only sequential statements
end ARCHITECTURE_NAME;
ENTITY PRNG IS
  PORT(
    Coeff   : in  std_logic_vector(4 downto 0);
    Load_Coeff : in  std_logic;
    Seed     : in  std_logic_vector(4 downto 0);
    Init_Run : in  std_logic;
    Clk      : in  std_logic;
    Current_State : out std_logic_vector(4 downto 0));
END PRNG;

ARCHITECTURE mixed OF PRNG is
  signal Ands      : std_logic_vector(4 downto 0);
  signal Sin       : std_logic;
  signal Coeff_Q   : std_logic_vector(4 downto 0);
  signal Sr5_Q     : std_logic_vector(4 downto 0);
  -- continue on the next slide
BEGIN

-- Data Flow
Sin <= Ands(0) XOR Ands(1) XOR Ands(2) XOR Ands(3) XOR Ands(4);
Current_State <= sr4_Q;
Ands <= Coeff_Q AND Sr5_Q;

-- Behavioral
Coeff_Reg: PROCESS(Clk)
BEGIN
  IF rising_edge(Clk) THEN
    IF Load_Coeff = '1' THEN
      Coeff_Q <= Coeff;
    END IF;
  END IF;
END IF;
END PROCESS;

-- Structural
Sr4_pl_0 : ENTITY work.Sr4(behavioral)
generic map(N => 5);
  PORT MAP (D => Seed, Load => Init_Run, Sin => Sin,
    Clock => Clk, Q => Sr5_Q);
END mixed;
Sequential Logic Modeling
Sequential Logic – Overview

inputs \rightarrow \text{reg\_next} \rightarrow \text{register} \rightarrow \text{reg} \rightarrow \text{clk} \downarrow \rightarrow \text{output}
architecture arch of seq_template is
    signal reg : std_logic_vector(N-1 downto 0);
begin
    process(clk, reset)
    begin
        if reset='1' then
            reg <= (others => '0');
        elsif rising_edge(clk) then
            reg <= f(inputs, reg);
        end if;
    end process;
end architecture arch;
architecture arch of disp_mux is
  signal reg, reg_next : std_logic_vector(N-1 downto 0);
begin
  process(clk, reset)
  begin
    if reset='1' then
      reg <= (others => '0');
    elsif rising_edge(clk) then
      reg <= reg_next;
    end if;
  end process;
  reg_next <= f(inputs, reg);
end architecture arch;
VHDL Variables
entity variable_in_process is
  port (  
    A,B : in  std_logic_vector (3 downto 0);  
    ADD_SUB : in  std_logic;  
    S : out std_logic_vector (3 downto 0) );
end variable_in_process;

architecture archi of variable_in_process is
begin  
  process (A, B, ADD_SUB)
  variable AUX : std_logic_vector (3 downto 0);
  begin
    if ADD_SUB = '1' then
      AUX := A + B ;
    else
      AUX := A - B ;
    end if;
    S <= AUX;
  end process;
end archi;
Differences: Signals vs Variables

- Variables can only be declared and used within processes or procedures.
  - Used to hold temporary results.
- Signals can only be declared in architecture.
  - Used for inter-process communications.
- Variables are updated immediately.
- Signals are updated after current execution of a process is finished.
- Synthesis results:
  - Variables: similar to signals, but maybe optimized away
  - Signals: wires, registers, or latches.
Differences: Signals vs Variables

process (a, b, c, s) begin
  s <= a and b;
  o <= s xor c;
end process;

process (a, b, c, s) begin
  o <= s xor c;
  s <= a and b;
end process;

process (a, b, c) variable s : std_logic; begin
  s := a and b;
  o <= s xor c;
end process;

process (a, b, c) variable s : std_logic; begin
  o <= s xor c;
  s := a and b;
end process;
Differences: Signals vs Variables

process (a, b, c, s)
begin
  s <= a and b;
  o <= s xor c;
end process;

process (a, b, c)
  variable s : std_logic;
begin
  s := a and b;
  o <= s xor c;
end process;
Differences: Signals vs Variables

process (a, b, c)
  variable s : std_logic;
begin
  o <= s xor c;
  s := a and b;
end process;
Differences:
Signals vs Variables

architecture sig_ex of test is
  signal s : std_logic;
begin
  process (clk)
  begin
    if rising_edge(clk) then
      s <= a and b;
      o <= s xor c;
    end if;
  end process;
end sig_ex;
architecture var_ex of test is
begin
  process (clk)
    variable s : std_logic;
    begin
      if rising_edge(clk) then
        s := a and b;
        o <= s xor c;
      end if;
    end process;
  end var_ex;
Signals vs Variables – Summary

\[ X_1 \rightarrow C_1 \rightarrow S_1 \rightarrow \text{process (clk)} \begin{align*} &\text{if rising_edge(clk) then} \\
&\quad S_1 \equiv C_1(X_1, S_1, S_2); \\
&\quad S_2 \equiv C_2(X_2, S_1, S_2); \\
&\text{end if;} \\
&\text{end process;}
\]

\[ X_2 \rightarrow C_2 \rightarrow S_2 \rightarrow \text{end} \]
Signals vs Variables – Summary

process (clk)
begin
    if rising_edge(clk) then
        V := C1(X1, S);
        S <= C2(X2, S);
    end if;
end process;
Case Studies
Stopwatch
Stopwatch

00.0 → 00.1 → ... → 00.9 → 01.0 → ... → 99.9

AB.C

• A: count of 10 seconds;
• B: count of 1 seconds;
• C: count of 0.1 seconds.

How to measure 0.1 second?
Stopwatch – Concept

Second10 Counter

Second Counter

Msec Counter

Msec Tick Gen

d2

d1

d0

d2_en

d1_en

clk

ms

tick

go

Stopwatch – Concept
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity stop_watch is
  port(clk, go, clr : in std_logic;
       d0 : out std_logic_vector(3 downto 0);
       d1 : out std_logic_vector(3 downto 0);
       d2 : out std_logic_vector(3 downto 0));
end entity stop_watch;
Stopwatch – VHDL Code

architecture caecade_arch of stop_watch is
    constant DVSR : integer := 10000000;
signal ms_reg, ms_next : unsigned(23 downto 0);
signal d0_reg, d1_reg, d2_reg
    : unsigned(3 downto 0);
signal d0_next, d1_next, d2_next
    : unsigned(3 downto 0);
signal d1_en, d2_en : std_logic;
signal ms_tick, d0_tick, d1_tick, d2_tick
    : std_logic;
begin
    -- to continue on the next slide
Stopwatch – VHDL Code

architecture caecade_arch of stop_watch is
    ... -- see previous slide
begin
    -- register update
    process(clk)
    begin
        if rising_edge(clk) then
            ms_reg <= ms_next;
            d0_reg <= d0_next;
            d1_reg <= d1_next;
            d2_reg <= d2_next;
        end if;
    end process;
end process;
Stopwatch – VHDL Code

-- next state logic
-- 0.1 sec tick generator
ms_next <= (others =>'0') when clr='1' or
          (ms_reg=DVSR and go='1') else
          ms_reg+1 when go='1' else
          ms_reg;

ms_tick <= ms_reg=DVSR;  -- generate ms_tick

-- 0.1 second counter
do_next <= "0000" when clr='1' or
          (ms_tick='1' and d0_reg=9) else
          d0_reg+1 when ms_tick='1' else
          d0_reg;

d0_tick <= d0_reg=9;
Stopwatch – VHDL Code

-- next state logic
-- 1 sec counter
d1_en <= ms_tick='1' and d0_tick='1';
d1_next <= “0000” when clr='1' or
          (d1_en='1' and d1_reg=9) else
          d1_reg+1 when d1_en='1' else
          d1_reg;

d1_tick <= d1_reg=9;

-- 10 second counter
d2_en <= d1_en and d1_tick='1';
d2_next <= “0000” when clr='1' or
          (d2_en='1' and d2_reg=9) else
          d2_reg+1 when d2_en='1' else
          d2_reg;
Stopwatch – VHDL Code

-- output logic

d0 <= d0_reg;
d1 <= d1_reg;
d2 <= d2_reg;

end architecture cascade;
Non-Synthesizable VHDL
Testbench Code

-- clock generator
process
begin
  clk <= '0';
  wait for 10ns;
  clk <= '1';
  wait for 10ns;
end process;
Testbench Code

-- test vector generator
process
begin
  -- initialization
  wait until falling_edge(clk);
  -- generate some random inputs
  wait until rising_edge(clk);
  -- generate other random inputs
end process;
Testbench Code

-- test vector generator
process
begin
  ...
  -- generate random inputs
  for i in 1 to 10 loop  --wait for 10 cycles.
    wait until rising_edge(clk);
  end loop;
  -- check if outputs are correct
end process;
Delays

Delays are not synthesizable.

Statements, such as

wait for 5 ns

a <= b after 10 ns

will not produce the required delay, and should not be used in the code intended for synthesis.
Signal Initialization

→ Declarations of signals (and variables) with initialized values, such as

```
signal a : STD_LOGIC := '0;
```
are allowed by Xilinx.

→ Models become less portable

→ *Set and reset signals explicitly instead.*

→ Define reset logic for each synchronous process
Dual-Edge Triggered Register/Counter

→ In FPGAs, registers/counters change only on either rising or falling edge of the clock, but not both.

→ Dual edge triggered processes are not synthesizable.

→ A single process is sensitive to either rising edge or falling edge, but not both, of the clock.
Dual-Edge Triggered Register/Counter

PROCESS (clk)
BEGIN
  IF rising_edge(clk) THEN
    counter <= counter + 1;
  ELSIF falling_edge(clk) THEN
    counter <= counter + 1;
  END IF;
END PROCESS;
Dual-Edge Triggered Register/Counter

PROCESS (clk)
BEGIN
  IF (clk'EVENT) THEN
    counter <= counter + 1;
  END IF;
END PROCESS;

PROCESS (clk)
BEGIN
  counter <= counter + 1;
END PROCESS;
Single Driver Rule for Signals

Given a single signal, the assignments to this signal should only be made within a single process block in order to avoid possible conflicts in assigning values to this signal.

Process 1: PROCESS (a, b)
BEGIN
   y <= a AND b;
END PROCESS;

Process 2: PROCESS (a, b)
BEGIN
   y <= a OR b;
END PROCESS;
Backup
7-Segment Display Multiplexer

Figure 17. An un-illuminated seven-segment display and nine illumination patterns corresponding to decimal digits.

The anodes of the seven LEDs forming each digit are tied together into one "common anode" circuit node, but the LED cathodes remain separate, as shown in Fig. 18. The common anode signals are available as four "digit enable" input signals to the 4-digit display. The cathodes of similar segments on all four displays are connected into seven circuit nodes labeled CA through CG (for example, the four "D" cathodes from the four digits are grouped together into a single circuit node called "CD"). These seven cathode signals are available as inputs to the 4-digit display.

This signal connection scheme creates a multiplexed display, where the cathode signals are common to all digits but they can only illuminate the segments of the digit whose corresponding anode signal is asserted. To illuminate a segment, the anode should be driven high while the cathode is driven low. However, since the Basys3 uses transistors to drive enough current into the common anode point, the anode enables are inverted. Therefore, both the AN0..3 and the CA..G/DP signals are driven low when active.

A scanning display controller circuit can be used to show a four-digit number on this display. This circuit drives the anode signals and corresponding cathode patterns of each digit in a repeating, continuous succession at an update rate that is faster than the human eye can detect. Each digit is illuminated just one-fourth of the time, but because the eye cannot perceive the darkening of a digit before it is illuminated again, the digit appears continuously illuminated. If the update, or "refresh", rate is slowed to around 45 Hz, a flicker can be noticed in the display.

For each of the four digits to appear bright and continuously illuminated, all four digits should be driven once every 1 to 16ms, for a refresh frequency of about 1KHz to 60Hz. For example, in a 62.5Hz refresh scheme, the entire display would be refreshed once every 16ms, and each digit would be illuminated for 1/4 of the refresh cycle, or 4ms. The controller must drive the cathodes low with the correct pattern when the corresponding anode signal is driven high.

To illustrate the process, if AN0 is asserted while CB and CC are asserted, then a "1" will be displayed.
7-Segment Display Controller

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity disp_mux is
  port(clk, reset : in std_logic;
in3, in2, in1, in0 :
in std_logic_vector(7 downto 0);
an : out std_logic_vector(3 downto 0);
ssseg : out std_logic_vector(7 downto 0));
end entity disp_mux;
7-Segment Display Controller

architecture arch of disp_mux is
  constant N : integer := 18;
signal q_reg, q_next : unsigned(N-1 downto 0);
signal sel : std_logic_vector(1 downto 0);
begin
  process(clk, reset)
  begin
    if reset='1' then
      q_reg <= (others => '0');
    elsif rising_edge(clk) then
      q_reg <= q_next;
    end if;
  end process;
  q_next <= q_reg + 1;
7-Segment Display Controller

```
sel <= std_logic_vector(q_reg(N-1 downto N-2));

process(sel, in0, in1, in2, in3)
begin
  case sel is
  when "00" => an <= "1110"; sseg <= in0;
  when "01" => an <= "1101"; sseg <= in1;
  when "10" => an <= "1011"; sseg <= in2;
  when others => an <= "0111"; sseg <= in3;
  end case;
end process;
end architecture;
```

*How long does it take for ‘sel’ to change?*
Generic Component Instantiation
N-bit Register with Enable

ENTITY regn IS
    GENERIC ( N : INTEGER := 8 ) ;
    PORT (Enable, Clock : IN STD_LOGIC;
        D : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
        Q : OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0)) ;
END regn ;

ARCHITECTURE Behavior OF regn IS
BEGIN
    PROCESS (Clock)
    BEGIN
        IF rising_edge(Clock) THEN
            IF Enable = '1' THEN
                Q <= D ;
            END IF ;
        END IF ;
    END PROCESS ;
END Behavior ;
Circuit Built of Components
ENTITY priority_resolver IS
  PORT(
    r : IN  STD_LOGIC_VECTOR(5 DOWNTO 0);
    s : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
    clk : IN  STD_LOGIC;
    en : IN  STD_LOGIC;
    t : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END priority_resolver;

ARCHITECTURE structural OF priority_resolver IS

  SIGNAL  p : STD_LOGIC_VECTOR (3 DOWNTO 0);
  SIGNAL  q : STD_LOGIC_VECTOR (1  DOWNTO 0);
  SIGNAL  z : STD_LOGIC_VECTOR (3 DOWNTO 0);
  SIGNAL  ena : STD_LOGIC;

-- continue on the next slide
BEGIN
    u1: ENTITY work.mux2to1(dataflow)
        PORT MAP( w0 => r(0), w1 => r(1),
            s => s(0), f => p(0));

    p(1) <= r(2);
    p(2) <= r(3);

    u2: ENTITY work.mux2to1(dataflow)
        PORT MAP( w0 => r(4), w1 => r(5),
            s => s(1), f => p(3));

    -- continue on the next slide
Structural Description – VHDL-93

u3: ENTITY work.priority(dataflow)
    PORT MAP (w => p, y => q, z => ena);

u4: ENTITY work.dec2to4 (dataflow)
    PORT MAP (w => q, En => ena, y => z);

u5: ENTITY work.regne(behavioral)
    GENERIC MAP (N => 4)
    PORT MAP (D => z, Enable => En,
              Clock => Clk, Q => t);
END structural;
ENTITY priority_resolver IS
  PORT (   r : IN  STD_LOGIC_VECTOR(5 DOWNTO 0);
            s : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
            clk : IN  STD_LOGIC;
            en : IN  STD_LOGIC;
            t : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END priority_resolver;

ARCHITECTURE structural OF priority_resolver IS

  SIGNAL  p : STD_LOGIC_VECTOR (3 DOWNTO 0);
  SIGNAL  q : STD_LOGIC_VECTOR (1 DOWNTO 0);
  SIGNAL  z : STD_LOGIC_VECTOR (3 DOWNTO 0);
  SIGNAL  ena : STD_LOGIC;

  -- continue on the next slide
COMPONENT mux2to1
  PORT (w0, w1, s : IN STD_LOGIC;
        f : OUT STD_LOGIC);
END COMPONENT;

COMPONENT priority
  PORT (w : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
        y : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
        z : OUT STD_LOGIC );
END COMPONENT;
COMPONENT dec2to4

    PORT( w : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
          En : IN STD_LOGIC;
          y  : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));

END COMPONENT;

COMPONENT regn

    GENERIC(N : INTEGER := 8 );

    PORT( D    : IN   STD_LOGIC_VECTOR(N-1 DOWNTO 0);
          En   : IN   STD_LOGIC;
          Clk  : IN   STD_LOGIC;
          Q    : OUT  STD_LOGIC_VECTOR(N-1 DOWNTO 0));

END COMPONENT ;
BEGIN
    u1: mux2to1 PORT MAP (w0 => r(0), w1 => r(1),
                         s => s(0), f => p(0));
    p(1) <= r(2);
    p(2) <= r(3);

    u2: mux2to1 PORT MAP (w0 => r(4), w1 => r(5),
                         s => s(1), f => p(3));

    u3: priority PORT MAP (w => p, y => q, z => ena);

    u4: dec2to4 PORT MAP (w => q, En => ena, y => z);

    u5: regne GENERIC MAP (N => 4)
        PORT MAP (D => z, En => En,
                  Clk => Clk, Q => t);
END structural;
Component Instantiation in VHDL-93

architecture structural of ex is begin
  U1 : ENTITY work.regn(behavioral)
      GENERIC MAP (N => 4)
      PORT MAP (D => z,
                 Resetn => reset,
                 Clock => clk,
                 Q => t);
  -- other concurrent statements
end architecture structural;

Generic map is optional.
architecture structural of ex is 
component regn IS
   GENERIC (N : INTEGER := 16) ;
   PORT( D : in STD_LOGIC_VECTOR(N-1 downto 0);
         Resetn, Clock : in STD_LOGIC;
         Q : out STD_LOGIC_VECTOR(N-1 downto 0));
end regn;
begin 
   U1: regn  GENERIC MAP (N => 8)
       PORT MAP( D => z,
                  Resetn => reset ,
                  Clock => clk,
                  Q => t);

   -- other concurrent statements 
end architecture structural;
A Word on Generics

→ Generics are typically integer values
→ In this class, the entity inputs and outputs should be std_logic or std_logic_vector.
→ But the generics can be integer.
→ Generics are given a default value
→ GENERIC ( N : INTEGER := 16 ) ;
→ This value can be overwritten when entity is instantiated as a component
→ Generics are very useful when instantiating an often-used component
→ Need a 32-bit register in one place, and 16-bit register in another
→ Can use the same generic code, just configure them differently