Modeling Behavior: Synthesis

vhdl model

entity my_ckt is
port(x, y: in bit;
     z: out bit)
end entity my_ckt;
architecture behavioral of my_ckt is
begin
--
-- process code here
--
end architecture behavioral;

- Inferring hardware from sequential code in processes
  - multiple alternative implementations
  - metrics for selecting the right implementation
- Coding style can “help” the compiler
Language Directed View

- Inferring combinational logic
  - every combination of inputs produces an output
  - every execution path performs an assignment to every signal

- Inferring sequential logic
  - a signal is not assigned in some execution path
    remember the value, i.e., storage
Simple Assignment Statements

architecture behavior of stover is
signal sig_s1, sig_s2: std_logic;
begin
proc1: process (x, y, z) is -- Process 1 using variables
variable var_s1, var_s2: std_logic;
begin
var_s1 := x and y;
var_s2 := var_s1 xor z;
res1 <= var_s1 and var_s2;
end process;

proc2: process (x, y, z) -- Process 2 using signals
begin
sig_s1 <= x and y;
sig_s2 <= sig_s1 xor z;
res2 <= sig_s1 and sig_s2;
end process;
end architecture behavior;

• simulation mismatch
• dependence analysis
Conditional Statements

```
architecture behavior of inflate is
begin
process (x, y, z, sel) is
variable s1, s2: std_logic;
begin
if (sel = '1') then
  s1 := x and z;
  s2 := s1 xor y;
  w <= s2 and s1;  -- w gets a value only conditionally
end if;
-- hence a latch is inferred
end process;
end architecture behavior;
```
Avoiding Latch Inference

- Ensure every path computes a value for every signal
  - presence of the else branch
  - nested structures
  - if-then-elsif
- Use initial values
  - Basic principle: ensure that every combination of input signal values leads to a computation of a value for every output signal

 combinatorial logic
Revisit the Example

architecture behavior of inflate is
begin
process (x, y, z, sel) is
variable s1, s2: std_logic;
begin
w <= '0'; -- output signal set to a default value to avoid latch inference
if (sel = '1') then
s1 := x and z; -- body generates combinatorial logic
s2 := s1 xor y;
w <= s2 and s1;
end if;
end process;
end architecture behavior;
Efficiency Considerations

• Now that we can control latch inferencing what about circuit size and speed?
• Move common operations (hardware) out of the branches
  • good programming practice in general
  • trade multiplexors for more expensive hardware
Efficiency Considerations

architecture behavior of inference is
begin
process (x, y, z, sel)
variable right: std_logic;
begin
if (sel = ‘1’) then
right:= y;
else
right:= z;
end if;
w <= x and right;
end process;
end behavior;

• Eliminated the latches
• Computation (logical AND) preceded by operand selection (mux)
Nested Constructs

architecture behavior of unsteadies
is
begin
process (x, y, z, sel1, sel2, sel3)
begin
  if (sel1 = '1') then
    re <= x and y;
  elsif (sel2 = '1') then
    re <= y xor z;
  elsif (sel3 = '1') then
    re <= x or y;
  end if;
end process;

• Latch inference due to the absence of the last “else”
• Latch inference due to placement in higher level blocks
• Priority ordering of the computations
Comparison Logic and Effects

architecture behavior of stover is
begin
process (x, y, z) is
begin
if (sel = "– 0") then -- the symbol “–” represents the don’t care
    w <= x nor y; -- for std_logic types
else
    w <= x nor z;
end if;
end process;
end architecture behavior;

• Comparison always returns false - the “then” branch is never taken
• Synthesis returns the combinational logic for the “else” branch
Case Statements

architecture behavior of case_st is
begin
process (x,y,z,sel) is
begin
case sel is
when 0 => res <= x and y;
when 1 => res <= y xor z;
when 2 => res <= x nand z;
when others => res <= x nor z;
end case;
end process;
end architecture behavior;

- Synthesis of a multiplexor
- Latch inference intuition applies
• Use of the “null” statement and latch inference
Loop Statements

- How much hardware should be generated?
- Most commonly supported is the *for loop*
  - number of iterations is known apriori
  - loop is unrolled and optimized as a sequence of sequential statements

```vhdl
for N in 3 downto 1 loop
    shift_reg (n) <= shift_reg (n-1);
end loop;
```

- Dependencies within and across iterations
  - cross iteration dependencies synthesize to long signal paths
Loop Statements: Example

architecture behavior of iteration is
signal shift_reg : std_logic_vector(3 downto 0);
begin
process (clk, reset, data) is
begin
  if (rising_edge(clk)) then
    if (reset = ‘1’) then res <= “0000”; 
    else
      for n in 3 downto 1 loop
        shift_reg(n) <= shift_reg(n-1);
      end loop;
      shift_reg(0) <= data;
      res <= shift_reg;
    end if;
  end if;
end process;
end architecture behavior;

• call to rising_edge(clk) produces edge triggered flip flops
• loop leads to sequence of dependent assignment statements
Loop Statements: Example
Miscellaneous Issues

- Use of sensitivity lists
- Pre-synthesis and post-synthesis simulation mismatches
- Optimizations involving variables
- Architecture level interpretation of inference rules
- Use of buffer and inout modes.

\[
\begin{align*}
process \ (x,y) \ is \\
\quad \quad \\
\quad \quad \\
\quad \quad \\
\quad \quad \\
\quad \quad \\
\end{align*}
\]
Inference using Signals vs. Variables

- Variable synthesized to a wire vs. signal synthesized to a latch
- Why is a latch inferred at all since all execution paths are covered?

library IEEE;
use IEEE.std_logic_1164.all;

entity sig_var is
port (sel : in std_logic;
  x, y, z: in std_logic;
  v, w: out std_logic);
end entity sig_var;

architecture behavior of sig_var is
signal sig_s1 : std_logic;
begin
process (x, y, z, sel)
variable var_s1: std_logic;

L1: if (sel = ‘1’) then
  sig_s1 <= x and z;
  v <= sig_s1 xor y;
end if;

L2: if (sel = ‘0’) then
  var_s1 := x and z;
  v <= var_s1 xor y;
end if;
end process;
end architecture behavior;
Inference using Signals vs. Variables

- current value (variable) vs. previous value (signal)
Inferring Storage for Variables

architecture behavior of sig_var is
begin
process
variable var_s1, var_s2 :std_logic;
begn
wait until (rising_edge(clk));
var_s1 := x nand var_s2;
var_s2 := var_s1 xor y;
res <= var_s1 xor var_s2;
end process;
end behavior;

• variables used before it is defined
• there exists an execution sequence(first) where use precedes definition
Latch vs. Flip Flop Inference

- Predicates in conditional expressions lead to latch inference
  
  \[ \text{if ( sel = '1') then...} \]

- Edge detection expressions lead to flip flop inference
  
  \[ \text{if (rising_edge(clk)) then...} \]
  \[ \text{if (clk'event and clk = '0') then..} \]
  \[ \text{if (clk'lastvalue = '0' and clk = '1' and clk'event) ..} \]

- Semantics must be consistent with available parts
  - latches vs. flip flops
  - trigger condition

**Simulation semantics only**
Example: Counter

define the entity and its ports:

```vhdl
entity counter is
port (clk, reset : in std_logic;
res : out unsigned (3 downto 0));
end entity counter;
```

define the behavior architecture:

```vhdl
architecture behavior of counter is
begin
process (clk, reset)
variable var_count : unsigned (3 downto 0);
begin
if (rising_edge (clk)) then
if (reset = ‘1’) then res <= “0000”;
else
var_count := var_count + 1;
end if;
res <= var_count;
end if;
end process;
end architecture behavior;
```
Example: Counter
Wait Statements

- wait statements imply synchronous logic
- only one wait statement permitted in a process and it must be the first statement in the process
- all signals in the body of the wait cause storage to be inferred
- use if-then-elsif-endif constructs to trade-off combinational vs. sequential circuit inference
Controlling Inference: Example

architecture behavioral of edge is
begin
process
begin
wait until (rising_edge(clk));
if reset = '1' then
Aout <= 1;
else
Aout <= 3;
end if;
Bout <= 0;
end process;
end architecture behavioral;

architecture behavioral of edge is
begin
process (reset, clk) is
begin
if reset = '1' then
Aout <= 1;
else
Aout <= 3;
end if;
if (rising_edge(clk)) then
Bout <= 0;
end if;
end process;
end architecture behavioral;

• control inference by use of the wait statement
• storage inference is a function of code structure
Variables Controlled by a Wait Statement

library IEEE;
use IEEE.std_logic_1164.all;
entity sigvar is
port(y,z, x, clk : in std_logic;
    res: out std_logic);
end entity sigvar;

architecture behavioral of sigvar is
begin
    process is
        variable a_var, b_var : std_logic;
        begin
            wait until (rising_edge(clk));
            a_var := x or y;
            b_var := a_var nor z;
            res <= b_var xor y;
        end process;
    end architecture behavioral;

• variable dependencies collapsed
• if variables used before they are defined then storage is inferred
• signal dependencies are not collapsed, but rather storage is inferred for each signal
Synthesis of State Machines

- Combinatorial component and sequential component
- Asynchronous and synchronous components
- Control inference through use of wait and edge detection expressions
State Encodings

• State encodings

<table>
<thead>
<tr>
<th>State</th>
<th>Sequential</th>
<th>Gray Code</th>
<th>One Hot</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>000</td>
<td>00000001</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>001</td>
<td>00000010</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>011</td>
<td>00000100</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>010</td>
<td>00001000</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>110</td>
<td>00010000</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>111</td>
<td>00100000</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>101</td>
<td>01000000</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>100</td>
<td>10000000</td>
</tr>
</tbody>
</table>

• Source-level constructs for user supplied encodings
• Goal: optimize area or speed
Controlling Inference: Example

architecture behavioral of state_machine is
  type statetype is (state0, state1);
  signal state, next_state : statetype;
begin
  process is
    begin
      wait until (rising_edge(clk)); -- rising edge
      if reset = '1' then
        res <= '0'; -- check for reset
        state <= statetype'left; -- initialize state
      else
        case state is -- switch on current state
          when state0 => -- set outputs and next state
            if x = '0' then
              state <= state1;
              res <= '1';
            else
              state <= state0;
              res <= '0';
            end if;
          when state1 =>
            if x = '1' then
              state <= state0;
              res <= '0';
            else
              state <= state1;
              res <= '1';
            end if;
        end case;
      end if;
    end process;
  end architecture behavioral;

  Flips flops inferred for all signals in the body of the process
Controlling Inference: Example

res

state
Controlling Inference: Example

architecture behavioral of state_machine is

    type statetype is (state0, state1);
    signal state, next_state : statetype;

begin

    process (state, x) is
    begin
        case state is -- switch on the current state
            when state0 => -- set output and next state
                if x = '0' then
                    next_state <= state1;
                    res <= '1';
                else
                    next_state <= state0;
                    res <= '0';
                end if;
            when state1 =>
                if x = '1' then
                    next_state <= state0;
                    res <= '0';
            end case;

        end process;

    clk_process: process is
    begin
        wait until (rising_edge(clk)); -- wait until the rising edge
        if reset = '1' then -- check for reset and initialize state
            state <= statetype'left;
        else
            state <= next_state;
        end if;
    end process clk_process;

end architecture;

• Flip flops for the signal “res” are avoided
Implementing Global Set/Reset in VHDL

- Use special on-chip circuitry
- GSR inferencing optimization
- Instantiate the following element to explicitly use this facility

![Diagram of GSR/Reset element]

- Connect your reset signal to this global net
- Note
  - both set/reset for a flip flop not supported
  - improves “routability” of designs
  - active low reset?
library IEEE;
use IEEE.std_logic_1164.all;

entity sig_var is
port ( clk, reset, x, y, z: in std_logic;
      w : out std_logic);
end sig_var;

architecture behavior of sig_var is
component STARTUP is
port (GSR : in std_logic);
end component STARTUP;
signal s1, s2 : std_logic;
begin
U1: STARTUP port map(GSR => reset);

process
begin
wait until (rising_edge(clk));
if (reset = '1') then
  w <= '0';
s1 <= '0';
s2 <= '0';
else
  L1: s1 <= x xor y;
  L2: s2 <= s1 or z;
  L3: w <= s1 nor s2;
end if;
end process;
end behavior;

• Active low reset?
Simulation vs. Synthesis

- Design flow transforms a design to more detailed levels
- Simulation at each level to validated design
- Need to understand sources of mismatch between levels
- Performance issues differ at each level
Simulation vs. Synthesis

• Obvious source: design error
• Incomplete sensitivity lists
  ```
  process (sel) is
  begin
  if (sel = '1' and En = '0') then
  A <= 1;
  else
  A <= '0';
  end if;
  end process;
  ```

• Use of Signals in a process
  ```
  process (x, y, z)
  begin
  L1: s1 <= x xor y;
  L2: s2 <= s1 or z;
  L3: w <= s1 nor s2;
  end process;
  ```
Simulation vs. Synthesis

- Delay statements
- Speed
  - overhead of maintaining and updating signal drivers
  - use of processes and variables speed up simulation
library IEEE;
use IEEE.std_logic_1164.all;
entity sig_var is
port ( clk, reset, x, y, z: in std_logic;
w : out std_logic);
end sig_var;
architecture behavior of sig_var is
begin
--U1: STARTUP port map(GSR => reset);
process
begin
wait until (rising_edge(clk));
if (reset = '1') then
  w <= '0';
s1 <= '0';
s2 <= '0';
else
  L1: s1 <= x xor y;
  L2: s2 <= s1 or z;
  L3: w <= s1 nor s2;
end if;
end process;
end behavior;