**Digital Design with Synthesizable VHDL**

Prof. Stephen A. Edwards  
sedwards@cs.columbia.edu

Columbia University  
Spring 2008

---

**Why HDLs?**

1970s: SPICE transistor-level netlists

An XOR built from four NAND gates

```
MODEL P PMOS
MODEL N NMOS
SUBCKT NAND A B Y Vdd Vss
M1 Y A Vdd Vdd PM
M2 Y B Vdd Vdd PM
M3 Y A X Vss NM
M4 X B Vss Vss N
NAND
```

1980s: Graphical schematic capture programs

---

**Two Separate but Equal Languages**

Verilog and VHDL

Verilog: More succinct, less flexible, really messy

VHDL: Verbose, very (too?) flexible, fairly messy

Part of languages people actually use identical. Every synthesis system supports both.

---

**Basic Lexical Rules of VHDL**

- Free-form: space only separates tokens.
- Case-insensitive: "VHDL," "vHdL," and "vhdl" are equivalent.
- Comments: from "--" to the end of the line.
- Identifiers: `[a-zA-Z](_?[a-zA-Z0-9]*)`

Examples: `X X_or_Y ADDR addr`

Illegal: `14M CLK__4 FOO_`

---

**Literals in VHDL**

- Decimal integers*: 1 42 153_1203
- Based integers*: `2#1_0010# 16#F001D#`
- Characters: `'0' '1' 'X`
- Strings: "101011" "XXXXXX"
- Bit string literals: `B"1001_0101" X"95"`

*Underscores added for readability are ignored

---

**Bits**

<table>
<thead>
<tr>
<th>Logical</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Voltage</td>
<td>1.65–3.3V</td>
<td>0–1.65V</td>
</tr>
<tr>
<td>Timing Diagram</td>
<td>__</td>
<td>__</td>
</tr>
</tbody>
</table>

In VHDL, zeros and ones on wires are members of an enumerated type. *They are not Boolean.*

---

**Combinational Logic in a Dataflow Style**
The std_logic_1164 package

```vhdl
package std_logic_1164 is
    type std_logic is
        ('U', -- Uninitialized
         'X', -- Forcing Unknown
         '0', -- Forcing 0
         '1', -- Forcing 1
         'Z', -- High Impedance
         'W', -- Weak Unknown
         '1', -- Weak 1
         '0' -- Weak 0
    end type;

    subtype std_logic is resolved std_ulogic;
end;
```

Boolean Operators

The basic ones in VHDL:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a and b</th>
<th>a or b</th>
<th>not a</th>
</tr>
</thead>
<tbody>
<tr>
<td>'0'</td>
<td>'0'</td>
<td>'0'</td>
<td>'1'</td>
<td>'1'</td>
</tr>
<tr>
<td>'0'</td>
<td>'1'</td>
<td>'0'</td>
<td>'1'</td>
<td>'0'</td>
</tr>
<tr>
<td>'1'</td>
<td>'0'</td>
<td>'0'</td>
<td>'1'</td>
<td>'1'</td>
</tr>
<tr>
<td>'1'</td>
<td>'1'</td>
<td>'1'</td>
<td>'0'</td>
<td>'0'</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a nand b</th>
<th>a nor b</th>
<th>a xor b</th>
</tr>
</thead>
<tbody>
<tr>
<td>'0'</td>
<td>'0'</td>
<td>'1'</td>
<td>'1'</td>
<td>'0'</td>
</tr>
<tr>
<td>'0'</td>
<td>'1'</td>
<td>'1'</td>
<td>'0'</td>
<td>'1'</td>
</tr>
<tr>
<td>'1'</td>
<td>'0'</td>
<td>'0'</td>
<td>'1'</td>
<td>'1'</td>
</tr>
<tr>
<td>'1'</td>
<td>'1'</td>
<td>'0'</td>
<td>'0'</td>
<td>'0'</td>
</tr>
</tbody>
</table>

Rules of Boolean Algebra (1)

-- Precedence
not a or b and c = (not a) or (b and c)

-- Basic relationships
not not a = a
a and '1' = a
a and '0' = '0'
a or '1' = '1'
a or '0' = a
a and a = a
a or a = a
a and not a = '0'
a or a = '1'
a and not a = '1'
a nand b = not (a and b)
a nor b = not (a or b)
a xor '0' = a
a xor '1' = not a
a xor b = (not a and b) or (a and not b)

Rules of Boolean Algebra (2)

A Full Adder: Truth Table

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>carry</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Simplifying Using Boolean Rules

carry <= (not a and b and c) or (a and not b and c) or (a and b and not c) or (a and b and c);

sum <= (not a and b and c) or (not a and b and c) or (a and not b and c) or (a and b and c);

-- De Morgan's Law
not (a or b) = not a and not b
not (a and b) = not a or not b

A Full Adder In VHDL

```vhdl
library ieee; -- always needed
use ieee.std_logic_1164.all; -- std_logic, et al.

entity full_adder is -- the interface
    port(a, b, c : in std_logic;
        sum, carry : out std_logic);
end entity full_adder;

architecture imp of full_adder is -- the implementation
begin
    sum <= (a xor b) xor c; -- combinational logic
    carry <= (a and b) or (a and c) or (b and c);
end imp;
```

Structure of a VHDL Module

...After Logic Synthesis

```vhdl
constant carry~1 : std_logic := 0;
constant carry~2 : std_logic := 0;
constant carry~3 : std_logic := 0;
constant carry~4 : std_logic := 0;
```

A Full Adder in VHDL

```vhdl
entity full_adder is
    port(a, b, c : in std_logic;
        sum, carry : out std_logic);
end;

architecture dataflow of full_adder is
begin
    sum <= a xor b xor carry~1;
    carry <= (a and b) or (a and c) or (b and c);
end;
```

Dataflow Expression

```
X <= '1' when Y = '1' and X = "110" else '0'
```
VHDL: Hex-to-seven-segment Decoder

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all; -- Provides the unsigned type

dec1_8

entity hex7seg is
port ( input : in unsigned(3 downto 0); -- A number
      output : out std_logic_vector(6 downto 0)); -- Just bits
endhex7seg;

architecture combinational of hex7seg is
begin
with input select
  "0111111" => '0',
  "0000110" when '1',
  "1011011" when '2',
  "1001111" when '3',
  "1110111" when '4',
  "0110001" when '5',
  "1111001" when '6',
  "1100001" when '7',
  "XXXXXX" when others;
end with;

end hex7seg;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity priority is
port (sel : in std_logic_vector(7 downto 0);
code : out unsigned(2 downto 0));
end priority;

architecture imp of priority is
begin
  code <= "000" when sel(0) = '1' else
          "001" when sel(1) = '1' else
          "010" when sel(2) = '1' else
          "011" when sel(3) = '1' else
          "100" when sel(4) = '1' else
          "101" when sel(5) = '1' else
          "110" when sel(6) = '1' else
          "111";
end imp;

begin
  GE <= '1' when A >= B else '0';

end rtl;

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

entity comparator is
port (A, B : in unsigned(1 downto 0);
GE : out std_logic);
end comparator;

architecture imp of comparator is
begin
  GE <= '1' when A => B else '0';
end imp;

begin
  data <= d_out when oe = '1' else
          tmp when oe = '0';

end rtl;

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

entity tri_demo is
port (addr : inout unsigned(15 downto 0);
      d0, d1 : out unsigned(7 downto 0);
      oe : out std_logic);
end tri_demo;

architecture rtl of tri_demo is
begin
  tri0 <= tri1(0);
end rtl;

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

entity add2 is
port (A, B, C : in std_logic;
      SUM, COUT : out std_logic);
end add2;

architecture imp of add2 is
component full_adder
port (a, b, c : in std_logic;
      sum, carry : out std_logic);
end component;

signal carry : std_logic;

begin
  bit0 : full_adder port map (A(0), B(0), '0', C(0), carry);
  bit1 : full_adder port map (A(1), B(1), carry, C(1), Z);
end imp;

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

entity alu is
port (A, B, C : in unsigned(7 downto 0);
      ADD : in std_logic;
      RES : out unsigned(7 downto 0));
end alu;

architecture imp of alu is
begin
  RES <= A + B when ADD = '1' else A - B;
end imp;

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

entity adder is
port (A, B : in unsigned(7 downto 0);
      SUM : out unsigned(7 downto 0);
      ADD : in std_logic;
      CO : out std_logic);
end adder;

architecture imp of adder is
begin
  tmp <= A + B + ("0" & ci);
  SUM <= tmp(7 downto 0);
  CO <= tmp(8);
end imp;

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

entity tri_demo is
port (addr : inout unsigned(15 downto 0);
      d0, d1 : out unsigned(7 downto 0);
      oe : out std_logic);
end tri_demo;

architecture rtl of tri_demo is
begin
  tri0 <= tri1(0);
end rtl;

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

entity comparator is
port (A, B : in unsigned(1 downto 0);
      GE : out std_logic);
end comparator;

architecture imp of comparator is
begin
  GE <= '1' when A => B else '0';
end imp;

begin
  data <= d_out when oe = '1' else
          tmp when oe = '0';

end rtl;
Hierarchical: port map by-name style

Direct Instantiation (no component)

Generate: Ripple-carry adder

Processes

A 4-to-1 mux in the procedural style

An Address Decoder

Summary of Procedural Modeling

null

Signal <= expr;

variable := expr;

if expr then stmts
  (elsif expr then stmts)
  (else stmts)
end if;

case expr is
  (when choices => stmts)
end case;

Note: when...else and with...select not allowed
### Sequential Logic

**Basic D Flip-Flop**

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity flipflop_enable is
port (Clk, Reset, D, EN : in std_logic;
     Q : out std_logic);
end flipflop_enable;
architecture imp of flipflop_enable is
begin
    process (Clk)
    begin
        if rising_edge(Clk) then
            if EN = '1' then
                Q <= D;
            else
                Q <= '0';
            end if;
        end if;
    end process;
end imp;
```

**Flip-Flop with Latch Enable**

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity flipflop_enable is
port (Clk, Reset, D, EN : in std_logic;
     Q : out std_logic);
end flipflop_enable;
architecture imp of flipflop_enable is
begin
    process (Clk)
    begin
        if rising_edge(Clk) then
            if EN = '1' then
                Q <= D;
            else
                Q <= '0';
            end if;
        end if;
    end process;
end imp;
```

### Flip-Flop with Synchronous Reset

**Eight-bit serial in/out shift register**

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity shifter is
port ( Clk, SI : in std_logic;
       Q : out std_logic);
end shifter;
architecture impl of shifter is
begin
    Q <= SI;
end impl;
```

**Flip-Flop with Latch Enable**

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity flipflop_enable is
port (Clk, Reset, D, EN : in std_logic;
     Q : out std_logic);
end flipflop_enable;
architecture imp of flipflop_enable is
begin
    process (Clk)
    begin
        if rising_edge(Clk) then
            if EN = '1' then
                Q <= D;
            else
                Q <= '0';
            end if;
        end if;
    end process;
end imp;
```

### Variables and Signals

**Synchronous RAM**

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity ram_32_4 is
port ( Clk, en : in std_logic;
       addr : in unsigned(3 downto 0);
       di : in unsigned(3 downto 0);
       do : out unsigned(3 downto 0));
end ram_32_4;
narrow architecture impl of ram_32_4 is
begin
    process (Clk)
    begin
        if rising_edge(Clk) then
            if en = '1' then
                RAM(TO_INTEGER(addr)) <= di;do <= di;
            end if;
        end if;
    end process;
end imp;
```

**A small ROM**

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity rom_32_4 is
port ( Clk, en : in std_logic;
       addr : in unsigned(3 downto 0);
       di : in unsigned(3 downto 0);
       do : out unsigned(3 downto 0));
end rom_32_4;
narrow architecture impl of rom_32_4 is
begin
    constant ROM : rom_type :=
        (X"1", X"2", X"3", X"4", X"5", X"6", X"7", X"8",
         X"9", X"A", X"B", X"C", X"D", X"E", X"F", X"11");
    process (Clk)
    begin
        if rising_edge(Clk) then
            if en = '1' then
                RAM(TO_INTEGER(addr)) <= ROM(TO_INTEGER(addr));
            end if;
        end if;
    end process;
end imp;
```

**Eight-bit serial in/out shift register**

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity shifter is
port ( Clk, SI : in std_logic;
       Q : out std_logic);
end shifter;
architecture impl of shifter is
begin
    Q <= SI;
end impl;
```

**Synchronous RAM**

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity ram_32_4 is
port ( Clk, en : in std_logic;
       addr : in unsigned(3 downto 0);
       di : in unsigned(3 downto 0);
       do : out unsigned(3 downto 0));
end ram_32_4;
narrow architecture impl of ram_32_4 is
begin
    process (Clk)
    begin
        if rising_edge(Clk) then
            if en = '1' then
                RAM(TO_INTEGER(addr)) <= di;do <= di;
            end if;
        end if;
    end process;
end imp;
```

**A small ROM**

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity rom_32_4 is
port ( Clk, en : in std_logic;
       addr : in unsigned(3 downto 0);
       di : in unsigned(3 downto 0);
       do : out unsigned(3 downto 0));
end rom_32_4;
narrow architecture impl of rom_32_4 is
begin
    constant ROM : rom_type :=
        (X"1", X"2", X"3", X"4", X"5", X"6", X"7", X"8",
         X"9", X"A", X"B", X"C", X"D", X"E", X"F", X"11");
    process (Clk)
    begin
        if rising_edge(Clk) then
            if en = '1' then
                RAM(TO_INTEGER(addr)) <= ROM(TO_INTEGER(addr));
            end if;
        end if;
    end process;
end imp;
```

**Eight-bit serial in/out shift register**

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity shifter is
port ( Clk, SI : in std_logic;
       Q : out std_logic);
end shifter;
architecture impl of shifter is
begin
    Q <= SI;
end impl;
```
**Variables vs. Signals**

- **Property** | **Variables** | **Signals**
  - Scope          | Local to process | Visible throughout architecture
  - Assignment     | Felt immediately (e.g., in next statement) | Only visible after clock rises (i.e., process terminates)

Lesson: use variables to hold temporary results and state to be hidden within a process. Otherwise, use signals.

---

**Combinational process**

Digital Design with Synthesizable VHDL – p. 63

**Sequential process**

---

**The Traffic Light Controller**

- This controls a traffic light at the intersection of a busy highway and a farm road. Normally, the highway light is green but if a sensor detects a car on the farm road, the highway light turns yellow then red. The farm road light then turns green until there are no cars or after a long timeout. Then, the farm road light turns yellow then red, and the highway light returns to green. The inputs to the machine are the car sensor, a short timeout signal, and a long timeout signal. The outputs are a timer start signal and the colors of the highway and farm road lights.


---

**Rocket Science: FSMs**

This is a **Mealy FSM**: outputs may depend directly on inputs.

---

**Circuit Design**

**Inputs**

**Outputs**

**Combinational Logic**

**State**

**Clock**

**Next State**

**Present State**

This is a Moore FSM: outputs come from state bits.

---

**Moore FSMs**

**Circuit Design**

**Inputs**

**Combinational Logic**

**Present State**

**Next State**

**State**

**Clock**

**Outputs**

**Coding Mealy State Machines**

**Library**

library ieee; use ieee.std_logic_1164.all;

**Entity**

entity mealy is
  port (clk : in std_logic; state, i1, i2, ... : in std_logic);
  signal next_state : states;
end entity mealy;

**Architecture**

architecture mealy of ... is
begin
  process (clk)
    begin
      if rising_edge(clk) then
        next_state <= state;
      end if;
  end process;

  process (reset, state, i1, i2, ... )
    variable state : states;
    begin
      if rising_edge(clk) then
        state <= next_state;
      else case state is
        when ZERO =>
          if i1 = '1' then
            state := ONE;
          else
            state := ZERO;
          end case;
          end if;
      end if;
    end process;
end architecture;
TLC in VHDL, continued

```vhdl
process (state, reset, cars, short, long)
begin
  if reset = '1' then
    start_timer <= '1'; next_state <= HG;
  else
    case state is
      when HG =>
        highway_yellow <= '0'; highway_red <= '0'; farm_yellow <= '0'; farm_red <= '1';
        if cars = '1' and long = '1' then
          start_timer <= '1'; next_state <= HY;
        else
          start_timer <= '0'; next_state <= HG;
        end if;
      when HY =>
        highway_yellow <= '1'; highway_red <= '0'; farm_yellow <= '0'; farm_red <= '1';
        if short = '1' then
          start_timer <= '1'; next_state <= FG;
        else
          start_timer <= '0'; next_state <= HY;
        end if;
      when FG =>
        highway_yellow <= '0'; highway_red <= '1'; farm_yellow <= '0'; farm_red <= '0';
        if cars = '0' or long = '1' then
          start_timer <= '1'; next_state <= FY;
        else
          start_timer <= '0'; next_state <= FG;
        end if;
      when FY =>
        highway_yellow <= '0'; highway_red <= '1'; farm_yellow <= '1'; farm_red <= '0';
        if short = '1' then
          start_timer <= '1'; next_state <= HG;
        else
          start_timer <= '0'; next_state <= FY;
        end if;
    end case;
  end if;
end process
end
```
II. Thou Shalt be Synchronous

- One global clock
- Flip-flops generate inputs to combinational logic, which computes inputs to flip-flops
- Exactly one value per signal per clock cycle
- Do not generate asynchronous reset signals; only use them if they are external
- Edge-triggered flip-flops only. Do not use level-sensitive logic
- Do not generate clock signals. Use multiplexers to create "load enable" signals on flip-flops.

III. Thou Shalt Be Sensitive

Combinational processes: list all process inputs

```vhdl
process (state, input)
begin
  case state is
    when S1 =>
      if input = '1' then
        output <= '0';
      else
        output <= '1';
      end if;
    when S2 =>
      output <= '1';
    end case;
end process;
```

Sequential processes: always include the clock. Include reset if asynchronous, and nothing else.

```vhdl
process (clk)
begin
  if rising_edge(clk) then
    if reset = '1' then
      Q <= '0';
    else
      Q <= '1';
    end if;
  end if;
end process;
```

Better to use an enumeration to encode states:

```vhdl
type states is (START, BKG, IDLE, ZAPWIN);
signal current, next: states;

process (current)
begin
  case current is
    when START => ...
    when BKG => ...
    when IDLE => ...
  end case;
end process;
```

Running this produces a helpful error:

```
Compiling vhdl file /home/cristi/cs4840/lab4/main.vhd in Library work.

$ ./main
/home/cristi/cs4840/lab4/main.vhd(39,14): error: unexpected character 'I

Digital Design with Synthesizable VHDL – p. 7
```

IV. Thou Shalt Assign All Outputs

"Default" values are convenient

```vhdl
process (state, input)
begin
  output <= '1';
end process;
```

```vhdl
process (state, input)
begin
  output <= '0';
end process;
```

V. Thou Shalt Enumerate States

Better to use an enumeration to encode states:
**VII: Thou Shalt Avoid Async**

Only use asynchronous reset when there is one global signal from outside.

--- OK for external Reset

```
process (Clk, Reset)
begin
  if Reset = '1' then
    Q <= '0';
  else
    if rising_edge(Clk) then
      Q <= '1';
    end if;
  end if;
end process;
```

--- Better

```
process (Clk)
begin
  if rising_edge(Clk) then
    if Reset = '1' then
      Q <= '0';
    else
      Q <= '1';
    end if;
  end if;
end process;
```

Never generate your own asynchronous reset. Generating a synchronous reset is fine.

**VIII: Thou Shalt Have One Version**

Never assume signals from the test bench that are not there on the board.

It is hard enough to make simulation match the design; do not make it any harder.

**IX: Thou Shalt Not Test For X Or Z**

Don't assign Boolean to `std_logic`.

The `wait` statement can delay for a certain amount of time, e.g., “wait 10ns;”

Only use it in test benches that are not meant to become hardware.

Do not use them in the design of your hardware.

**X: Thou Shalt Not Specify Delays**

```
library ieee;
use ieee.std_logic_1164.all;
entity dont_read_output is
  port ( a : in std_logic;
         x, y : out std_logic );
end dont_read_output;
architecture bad of dont_read_output is
begin
  x <= not a;
  y <= not x;
end bad;
architecture ok of dont_read_output is
begin
  x <= not a;
  y <= not x;
end ok;
```

**Pitfalls: Boolean vs. Std_logic**

Don't assign Boolean to `std_logic`.

```
signal a : std_logic;
signal b : unsigned(7 downto 0);
```

Example:

```
a <= b = x"7E"; -- BAD result is Boolean, not std_logic
a <= '1' when b = x"7E" else '0'; -- OK
```

**Pitfalls: Reading Output Port**

```
library ieee;
use ieee.std_logic_1164.all;
entity bad_port_map is
  component bar_port is
    port ( x : in unsigned(5 downto 0));
    component bar
      port map ( x => "000" & a);
  end component;
architecture bad of bar_port is
begin
  varbar : bar port map ( x => "000" & a);
end bad;
architecture ok of bar_port is
begin
  varbar : bar port map ( x => aa & a);
end ok;
```

**Pitfalls: Complex Port Map Args**

```
library ieee;
use ieee.std_logic_1164.all;
entity bad_port_map is
  component bar_port is
    port ( x : in unsigned(5 downto 0));
    component bar
      port map ( x => "000" & a);
  end component;
architecture bad of bar_port is
begin
  varbar : bar port map ( x => "000" & a);
end bad;
architecture ok of bar_port is
begin
  varbar : bar port map ( x => aa & a);
end ok;
```

**VI: Thou Shalt Not Have One Version**

Only use asynchronous reset when there is one global signal from outside.

This is legal VHDL, but the synthesized circuit won't behave like you expect.

**VIII: Thou Shalt Avoid Async**

Never assume signals from the test bench that are not there on the board.

It is hard enough to make simulation match the design; do not make it any harder.

**IX: Thou Shalt Not Test For X Or Z**

Don't assign Boolean to `std_logic`.

```
signal a : std_logic;
signal b : unsigned(7 downto 0);
```

Example:

```
a <= b = x"7E"; -- BAD result is Boolean, not std_logic
a <= '1' when b = x"7E" else '0'; -- OK
```

**X: Thou Shalt Not Specify Delays**

```
library ieee;
use ieee.std_logic_1164.all;
entity dont_read_output is
  port ( a : in std_logic;
         x, y : out std_logic );
end dont_read_output;
architecture bad of dont_read_output is
begin
  x <= not a;
  y <= not x;
end bad;
architecture ok of dont_read_output is
begin
  x <= not a;
  y <= not x;
end ok;
```

**Pitfalls: Boolean vs. Std_logic**

Don't assign Boolean to `std_logic`.

```
signal a : std_logic;
signal b : unsigned(7 downto 0);
```

Example:

```
a <= b = x"7E"; -- BAD result is Boolean, not std_logic
a <= '1' when b = x"7E" else '0'; -- OK
```

**Pitfalls: Reading Output Port**

```
library ieee;
use ieee.std_logic_1164.all;
entity bad_port_map is
  component bar_port is
    port ( x : in unsigned(5 downto 0));
    component bar
      port map ( x => "000" & a);
  end component;
architecture bad of bar_port is
begin
  varbar : bar port map ( x => "000" & a);
end bad;
architecture ok of bar_port is
begin
  varbar : bar port map ( x => aa & a);
end ok;
```

**Pitfalls: Complex Port Map Args**

```
library ieee;
use ieee.std_logic_1164.all;
entity bad_port_map is
  component bar_port is
    port ( x : in unsigned(5 downto 0));
    component bar
      port map ( x => "000" & a);
  end component;
architecture bad of bar_port is
begin
  varbar : bar port map ( x => "000" & a);
end bad;
architecture ok of bar_port is
begin
  varbar : bar port map ( x => aa & a);
end ok;
```
Pitfalls: Combinational Loops

You never really need them.

Drive every signal from exactly one process or concurrent assignment.

Don’t build SR latches. Use D flip-flops instead.

Pitfalls: Clock Gating

Dangerous, difficult to get right.

Use a single, global clock and latch enables to perform the same function.

Pitfalls: Multiple Clock Domains

If you must, vary the phase and drive clocks directly from flip-flops.

Testbenches

One of VHDL’s key points: can describe hardware and environment together.

-- Explicit delays are allowed
clk <= not clk after 50 ns;
process
begin
reset <= '0';
wait 10 ns; -- Explicit delay
reset <= '1';
wait for a = '1'; -- Delay for an event
assert b = '1' report "b_did_not_rise" severity failure;
assert c = '1' report "c=0" severity warning; -- or error or note
wait for 50 ns; -- Delay for some time
wait; -- Halt this process
end process;

Digital Design with Synthesizable VHDL – p. 91

Testbench Methodology

– Always put testbench in a separate .vhd file since it cannot be synthesized.
– Instantiate block under test and apply desired inputs (clocks, other stimulus)
– Use assert to check conditions
– Try to emulate hardware environment as closely as possible (no special inputs, etc.)

A Testbench

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity tlc_tb is -- A testbench usually has no ports
end tlc_tb;
architecture tb of tlc_tb is
signal clk : std_logic := '0'; -- Must initialize!
-- One signal per port is typical
signal reset, cars, short, long : std_logic;
signal farm_red, start_timer : std_logic;
begin
clk <= not clk after 34.92 ns; -- 14 MHz

A testbench continued

-- Apply stimulus and check the results
process
begin
cars <= '0'; short <= '0'; long <= '0'; reset <= '1';
wait for 100 ns;
assert start_timer = '1' report "No_timer" severity error;
reset <= '0';
wait for 100 ns;
assert farm_red = '1' report "Farm_not_red" severity error;
wait;
end process;

-- Instantiate the Unit Under Test
uut : entity work.tlc
port map ( clk => clk, reset => reset,
cars => cars, short => short,
long => long, farm_red => farm_red,
start_timer => start_timer);
end tb;

Digital Design with Synthesizable VHDL – p. 96