VHDL Modeling for Synthesis

Sequential Logic Circuits
VHDL “Process” Construct

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

- Process statements are executed \textit{in sequence}
- Process statements are executed once at start of simulation
- Process halts at “end” until an event occurs on a signal in the “sensitivity list”
- Allows conventional programming language structures to describe circuit behavior
Modeling sequential behavior

-- Edge-triggered flip flop/register
entity DFF is
  port (D, CLK: in bit;
       Q: out bit);
end DFF;
architecture behave of DFF is
begin
  process(clk)   -- "process sensitivity list"
  begin
    if (clk'event and clk='1') then
      Q <= D after 1 ns;
    end if;
  end process;
end;

- Process statements executed sequentially (sequential statements)
- clk'event is an attribute of signal clk which is TRUE if an event has occurred on clk at the current simulation time
Edge-triggered flip-flop

Alternative methods for specifying clock transition

process (clk)
begin
  if rising_edge(clk) then -- std_logic_1164 function
    Q <= D;
  end if;
end process;

Synthesis tool may also recognize: not clk’stable
as equivalent to: clk’event
Alternative to sensitivity list

process -- no “sensitivity list”
begin
  wait on clk; -- suspend process until event on clk
  if (clk='1') then
    Q <= D after 1 ns;
  end if;
end process;

- Other “wait” formats: wait until (clk’event and clk='1')
  wait for 20 ns;

- This format does not allow for asynchronous controls

- Process executes endlessly if no sensitivity list or wait statement!
Level-Sensitive D latch vs. D flip-flop

entity Dlatch is
  port (D, CLK: in bit;
       Q: out bit);
end Dlatch;
architecture behave of Dlatch is begin
  process (D, clk)
  begin
    if (clk='1') then
      Q <= D after 1 ns;
    end if;
  end process;
end;

Latch, Q changes whenever the latch is enabled by CLK='1' (rather than edge-triggered)
RTL “register” model (not gate-level)

entity Reg8 is
  port (D: in bit_vector(0 to 7);
        Q: out bit_vector(0 to 7);
        LD: in bit);
end Reg8;

architecture behave of Reg8 is begin
  process(LD)
  begin
    if (LD'event and LD='1') then
      Q <= D after 1 ns;
    end if;
  end process;
end;

D and Q can be any abstract data type
Basic format for synchronous and asynchronous inputs

process (clock, asynchronous_signals )
begin
  if (boolean_expression) then
    asynchronous signal_assignments
  elsif (boolean_expression) then
    asynchronous signal assignments
  elsif (clock'event and clock = contstant) then
    synchronous signal_assignments
  end if;
end if;
end process;
### Synchronous vs. Asynchronous Flip-Flop Inputs

**Entity DFF**

```vhdl
entity DFF is
    port (D, CLK: in bit;
          PRE, CLR: in bit;
          Q: out bit);
end DFF;
```

**Architecture behave of DFF**

```vhdl
architecture behave of DFF is
begin
    process(clk, PRE, CLR)
    begin
        if (CLR='0') then  -- async CLR has precedence
            Q <= '0' after 1 ns;
        elsif (PRE='0') then  -- then async PRE has precedence
            Q <= '1' after 1 ns;
        elsif (clk'event and clk='1') then
            Q <= D after 1 ns;  -- sync operation only if CLR=PRE='1'
        end if;
    end process;
end;
```

<table>
<thead>
<tr>
<th>CLR</th>
<th>D</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Detecting errors with “assert”

process (clock, clear, preset )
begin
    assert ((clear = '1') or (preset = '0'))
    report “Error” clear and present both active”;
    if (clear = '0') then
        q <= '0';                   -- reset has precedence
    elsif (preset = '1') then
        q <= '1';                   -- async preset
    elsif (clock'event and clock = '0') then
        q <= d ;                    -- synchronous load
    end if ;
end if ;
end process;
Synchronous reset/set

--Reset function triggered by clock edge

process (clk)
begin
    if (clk’event and clk = ‘1’) then
        if reset = ‘1’ then – reset has precedence over load
            Q <= ‘0’ ;
        else
            Q <= D ;
        end if;
    end if;
end process;
DFF with clock enable

process (clk)
begin
    if (clk’event and clk = ‘1’) then
        if enable = ‘1’ then
            Q <= D;
        end if;
    end if;
end process;

-- “enable” effectively enables/disables clocking
Using a “variable” to describe sequential behavior within a process

cnt: process(clk)

variable count: integer;  -- internal counter state
begin                                  -- valid only in a process
  if clk='1' and clk'event then
    if ld='1' then        -- “to_integer” must be supplied
      count := to_integer(Din);
    elsif cnt='1' then
      count := count + 1;
    end if;
  end if;
  Dout <= to_bitvector(count);
end process;
Circuits with counters

```vhdl
process begin
  wait until clk'event and clk='1' ;
  if (count = input_signal) then
    count <= 0 ;
  else
    count <= count + 1 ;
  end if ;
end process ;
-- counter and full-sized comparator generated to check count
```
Decrementer and comparator

```vhdl
process begin
    wait until clk'event and clk='1' ;
    if (count = 0) then
        count <= input_signal ;
    else
        count <= count - 1 ;
    end if ;
end process ;
-- comparison to 0 easier than a non-zero value
```
FFs generated from variables: 3-bit shift register example

-- External input/output din/dout
process (clk)
    variable a, b: bit;
begin
    if (clk'event and clk = '1') then
        dout <= b;
        b := a;
        a := din;
    end if;
end process;

-- note: a,b used before being assigned new values
3-bit shift register example

Unexpected resulting structure

```vhdl
process (clk)
    variable a,b: bit;
begin
    if (clk'event and clk = '1') then
        a := din;
        b := a;
        dout <= b;
    end if;
end process;
```

-- a,b changed before used so values are not stored
-- they become “wires”.
(Only one flip flop from din -> dout)
Example: shift register

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity SIPO_1 is
  port (Clk : in  STD_LOGIC;
        SI : in STD_LOGIC;  -- serial in
        PO : buffer STD_LOGIC_VECTOR(3 downto 0)); -- parallel out
end SIPO_1;

architecture Synthesis_1 of SIPO_1 is
begin
  process (Clk) begin
    if (Clk = '1') then
      PO <= SI & PO(3 downto 1); --shift PO right
    end if;
  end process;
end Synthesis_1;

--Synthesis tools often do not permit use of “buffer”
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity SIPO_R is
  port (clk : in STD_LOGIC; res : in STD_LOGIC;
       SI : in STD_LOGIC; PO : out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture Synthesis_1 of SIPO_R is
  signal PO_t : STD_LOGIC_VECTOR(3 downto 0); -- internal state
begin
  PO <= PO_t; -- drive external output
  process (clk, res) begin -- asynchronous reset plus clock
    if (res = '0') then
      PO_t <= (others => '0'); -- reset internal state
    elsif (rising_edge(clk)) then
      PO_t <= SI & PO_t(3 downto 1); -- shift internal state
    end if;
  end process;
end;

architecture Synthesis_1 of SIPO_R is
  signal PO_t : STD_LOGIC_VECTOR(3 downto 0); -- internal state
begin
  PO <= PO_t; -- drive external output
  process (clk, res) begin -- asynchronous reset plus clock
    if (res = '0') then
      PO_t <= (others => '0'); -- reset internal state
    elsif (rising_edge(clk)) then
      PO_t <= SI & PO_t(3 downto 1); -- shift internal state
    end if;
  end process;
end Synthesis_1;