

VHDL Modeling for Synthesis

Sequential Logic Circuits

VHDL “Process” Construct

- ▶ Allows conventional programming language methods to describe circuit behavior
- ▶ Supported language constructs (“sequential statements”) – **only allowed within a process:**
 - ▶ variable assignment
 - ▶ if-then-else (elsif)
 - ▶ case statement
 - ▶ while (condition) loop
 - ▶ for (range) loop



Process Format

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

- ▶ Process statements executed once at start of simulation
- ▶ Process halts at “end” until an event occurs on a signal in the “sensitivity list”



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
```

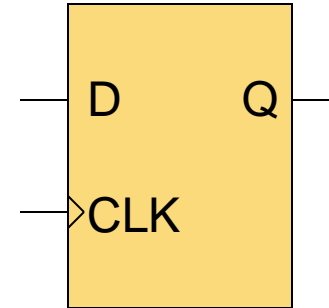
```
    if (clk'event and clk='1') then
```

```
      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

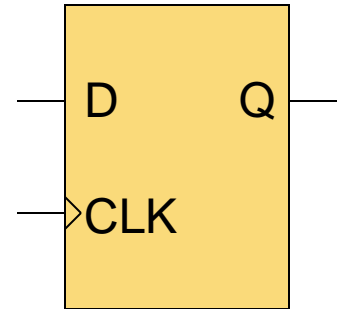
```
process (clk)
begin
    if rising_edge(clk) then -- std_logic_1164 function
        Q <= D ;
    end if;
end process;
```

Leonardo also recognizes 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;
```

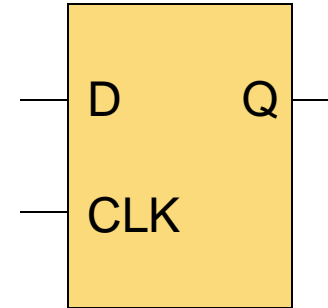


- ▶ Other “wait” formats: `wait until (clk'event and clk='1')`
`wait for 20 ns;`
 - ▶ This format does not allow for asynchronous controls
 - ▶ **Process executes endlessly if no sensitivity list or wait statement!**
-



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)



Defining a “register” for an RTL 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
```

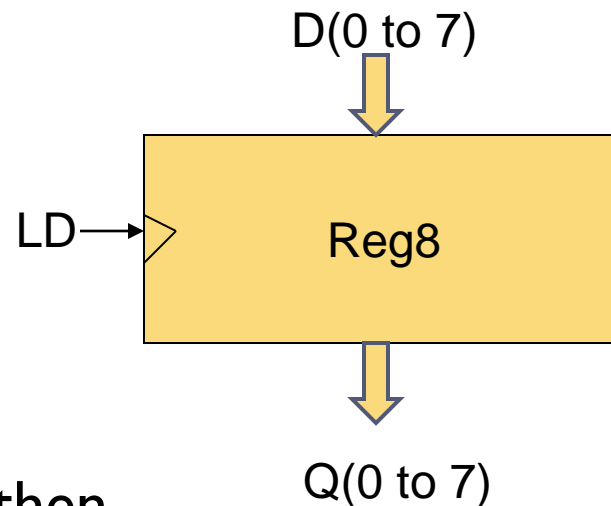
```
    if (LD'event and LD='1') then
```

```
        Q <= D after 1 ns;
```

```
    end if;
```

```
end process;
```

```
end;
```



D and Q could be any abstract data type



Basic format for synchronous and asynchronous inputs

```
process (clock, asynchronously_used_signals )
begin
    if (boolean_expression) then
        asynchronous signal_assignments
    elsif (boolean_expression) then
        asynchronous signal assignments
    elsif (clock'event and clock = constant) then
        synchronous signal_assignments
    end if ;
end process;
```



Synchronous vs. Asynchronous Flip-Flop Inputs

entity DFF is

```
port (D,CLK: in bit;  
      PRE,CLR: in bit;  
      Q: out bit);
```

end DFF;

architecture behave of DFF is

```
begin
```

```
  process(clk,PRE,CLR)
```

```
  begin
```

```
    if (CLR='0') then
```

```
      Q <= '0' after 1 ns;
```

```
    elsif (PRE='0') then
```

```
      Q <= '1' after 1 ns;
```

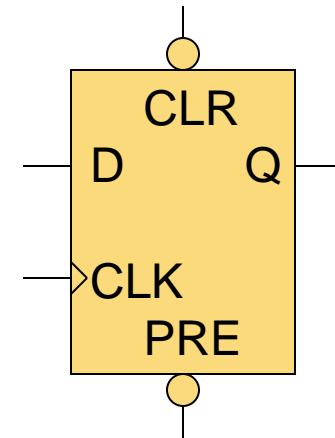
```
    elsif (clk'event and clk='1') then
```

```
      Q <= D after 1 ns;
```

```
    end if;
```

```
  end process;
```

```
end;
```



-- async CLR has precedence

-- then async PRE has precedence

-- sync operation only if CLR=PRE='1'



Example: register with asynchronous reset and preset

```
process (clock, clear, preset)
begin
```

```
  if (clear = '0') then
```

```
    q <= '0';           -- reset has precedence
```

```
  elsif (preset = '1') then
```

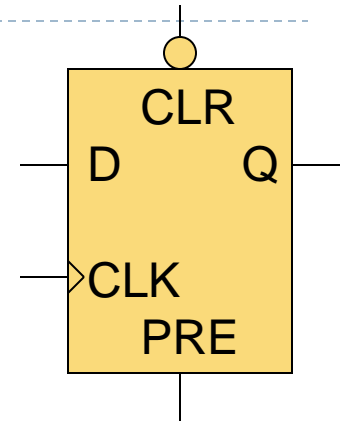
```
    q <= '1';           -- async preset
```

```
  elsif (clock'event and clock = '0') then
```

```
    q <= d;             -- synchronous load
```

```
  end if ;
```

```
end process;
```



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 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; -- “to_bitvector” must be supplied
```

```
    Dout <= to_bitvector(count);
```

```
end process;
```



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

```
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)



Predefined flip flops & latches

- ▶ Can instantiate pre-defined flip flops and latches from a library
- ▶ “Exemplar” library/package contains additional flip flops & latches with various configurations of asynchronous & synchronous inputs



Example: shift register (12.6.8)

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.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); end if;
  end process;
end Synthesis_1;
```



Eliminate “buffer” for synthesis

```
library IEEE; use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD.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
    process (PO_t) begin PO <= PO_t; end process; --drive external output
    process (clk, res) begin -- asynchronous reset plus clock
        if (res = '0') then
            PO_t <= (others => '0'); -- reset
        elsif (rising_edge(clk)) then
            PO_t <= SI & PO_t(3 downto 1); -- shift
        end if;
    end process;
end Synthesis_1;
```

Circuits with counters

```
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

```
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
```



Memory Synthesis (12.8)

- ▶ **Approaches:**
 - ▶ Random logic using flip-flops or latches
 - ▶ Register files in datapaths
 - ▶ RAM standard components
 - ▶ RAM compilers



Modeling RAM modules

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package RAM_package is
    constant numOut : INTEGER := 8;
    constant wordDepth: INTEGER := 8;
    constant numAddr : INTEGER := 3;
    subtype MEMV is STD_LOGIC_VECTOR(numOut-1 downto 0);
    type MEM is array (wordDepth-1 downto 0) of MEMV;
end RAM_package;
```



Modeling RAM modules

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
use work.RAM_package.all;
entity RAM_1 is
    port (signal A : in STD_LOGIC_VECTOR(numAddr-1 downto 0);
          signal CEB, WEB, OEB : in STD_LOGIC;
          signal INN : in MEMV;
          signal OUTT : out MEMV);
end RAM_1;
architecture Synthesis_1 of RAM_1 is
    signal i_bus : MEMV; -- RAM internal data latch
    signal mem : MEM; -- RAM data
begin
```



Modeling RAM modules

```
process begin
  wait until CEB = '0';  -- chip enable
  if WEB = '1' then     -- read if write not enabled
    i_bus <= mem(TO_INTEGER(UNSIGNED(A)));
  elsif WEB = '0' then  -- write enable
    mem(TO_INTEGER(UNSIGNED(A))) <= INN;
    i_bus <= INN;
  else i_bus <= (others => 'X'); end if;
end process;
```

```
process(OEB, int_bus) begin  -- control output drivers:
  case (OEB) is
    when '0'    => OUTT <= i_bus;
    when '1'    => OUTT <= (others => 'Z');
    when others => OUTT <= (others => 'X');
  end case;
end process;
```



64K x 8 Memory Example

```
library ieee;
use ieee.std_logic_1164.all;
use work.qsim_logic.all;      -- package with to_integer() func

entity memory8 is
  port (dbus: inout std_logic_vector(0 to 7);
        abus: in    std_logic_vector(0 to 15);
        ce:  in bit;      -- active low chip enable
        oe:  in bit;      -- active low output enable
        we:  in bit);     -- active low write enable
end memory8;
```



64K x 8 Memory Example

architecture reglevel of memory8 is

begin

process (ce,oe,we,abus,dbus)

type mem is array(natural range <>) of std_logic_vector(0 to 7);

variable M: mem(0 to 65535);

begin

if (ce='0') and (oe='0') then

-- read enabled

dbus <= M(to_integer(abus));

-- drive the bus

elsif (ce='0') and (we='0') then

-- write enabled

dbus <= "ZZZZZZZZ";

-- disable drivers

M(to_integer(abus)) := dbus;

-- write to M

else

dbus <= "ZZZZZZZZ";

--disable drivers

end if;

end process;

end;

