Lecture 5: Sequential and Combinatorial Logic, Statements, Generics

TIE-50206 Logic Synthesis
Arto Perttula
Tampere University of Technology
Fall 2016
Acknowledgements

• Prof. Pong P. Chu provided "official" slides for the book which is gratefully acknowledged
  – See also: http://academic.csuohio.edu/chu_p/
• Most slides were originally made by Ari Kulmala
  – and other previous lecturers (Teemu Pitkänen, Konsta Punkka, Mikko Alho, Erno Salminen…)
Contents

• Concurrent and sequential statements of VHDL
• Sequential logic from VHDL
  – Reset
• Common pitfalls
Concurrent vs. Sequential Statement in HDL

- Most programming languages are sequential but digital logic operates as parallel
- HW designers need a bit different frame of mind to take parallelism into account
- VHDL is a parallel language but some things are better captured with sequential description
- Hence, there are 2 types of statements
  1. Concurrent
  2. Sequential
Category 1: Concurrent Statements

• Define interconnected blocks and processes that jointly define the behaviour or structure of a design – inside the architecture

• Are executed in parallel and asynchronously with respect each others
  – PROCESS
  – COMPONENT INSTANTIATION
  – GENERATE
  – CONCURRENT SIGNAL ASSIGNMENT
  – CONDITIONAL SIGNAL ASSIGNMENT
  – SELECTED SIGNAL ASSIGNMENT
  – CONCURRENT ASSERTION
  – CONCURRENT PROCEDURE CALL
  – BLOCK

• All concurrent statements can be labeled
Category 2: Sequential Statements

- Executed in order in which they appear
- Can be used inside processes, procedures and functions
  - SIGNAL ASSIGNMENT
  - VARIABLE ASSIGNMENT
  - IF
  - CASE
  - ASSERTION
  - LOOP
  - NEXT
  - EXIT
  - PROCEDURE CALL
  - RETURN
  - NULL
  - WAIT
Detecting Concurrent And Sequential Code

• Concurrent:
  1. sync_proc
  2. comb_proc
  3. z <= ...;
  4. m_out <= ...

• Sequential
  1. Sync_proc internals
  2. Comb_proc internals

begin -- rtl
sync_proc: process (clk, rst_n)
  variable k_v : integer;
begin -- process sync_proc
  if rst_n = '0'
    x_r <= 0;
    y_r <= 0;
  elsif clk'event and clk = '1' then
    k_v := to_integer(unsigned(a_in));
    x_r <= to_integer(unsigned(a_in));
    y_r <= k_v + x_r;
  end if;
end process sync_proc;
comb_proc: process (a_in, c_in)
  variable inter_v : std_logic;
begin -- process comb_proc
  inter_v := a_in(0) and c_in(0);
  if inter_v = '1' then
    res <= a_in xor c_in;
  else
    res <= a_in;
  end if;
end process comb_proc;
z <= y_r;
m_out <= res when b_in(0) = '1' else std_logic_vector(to_unsigned(z, data_width_g));
end rtl;

Quiz: which signals implement a register?
A: x_r, y_r
Sequential Statements: IF

- Probably the most important sequential statement
- Like in conventional programming languages
  - Priority encoded (if tested first, then elsif, then else)

```plaintext
IF condition THEN
    sequence of statements
[ ELSEIF condition2
    sequence of statements ]
[ ELSE
    sequence of statements ]
END IF;
```

- Example of incrementing and clipping values within allowed range:

```plaintext
IF a > upper_limit_c THEN
    a <= upper_limit_c;
ELSIF a < lower_limit_c
    a <= lower_limit_c;
ELSE
    a <= a+1;
END IF;
```

Inside processes, procedures and functions only
Example: Combinatorial Mux Using IF

--
-- Synthesis example: Multiplexer using IF statement
--
ENTITY ifmultiplexer IS
  port (a, b, sel : IN STD_LOGIC;
       z : OUT STD_LOGIC);
END ifmultiplexer;

ARCHITECTURE syn OF ifmultiplexer IS
BEGIN -- Syn
  PROCESS (a, b, sel)
  BEGIN -- PROCESS
    IF (sel = '1') THEN
      z <= b;
    ELSE
      z <= a;
    END IF;
  END PROCESS;
END syn;

Arto Perttula  7.11.2016
Clear Control Structure Is Essential

if foo='0' then
    ...
end if;
...
stuff();
if foo='1' then
    ...
    if...
        a <='0';
        b <='1';
    else
        b <='0';
        a <='1';
    end if;

if foo='0' then
    ...
else
    ...
    -- Branches are
    -- mutually exclusive.
    -- Easier to analyze.
end if;
stuff(); -- Better separated

if...
    a <='0';
    b <='1';
else
    b <='0';
    a <='1';
end if;

if...
    a <='0';
    b <='1';
else
    a <='1';
    b <='0';
    -- Assignments in fixed
    -- order
end if;
Clear Control Structure (2)

if val > 10 then
    ...
    ...
    ... -- lots of code
    ...
    ...
else
    -- only few code lines
end if;

if foo = '1' then
    ...
    -- WOW! Something strange happened...
else
    -- Normal case
end if;

if val < buf_limit_c then
    -- Simple things first
    -- Constants vs. magic num
    -- Be careful with ±1
else
    ...
    ...
    ... -- lots of code
    ...
    ...
    ...
end if;

if foo = '0' then
    -- Normal case
else
    -- Error handling
    -- Better commenting
    ...
end if;
if foo='1' then
a <='1'; b<='1'; else
if bar=... then a<='0';... end if;
b <='1';
end if;

if ... then
  if full='1' then...
  else ...
else
  if full='0' then...
  else ...

b <='1'; -- moved here

if foo='1' then
  a <='1'; -- indent
  -- 1 statement/line
  -- Blank lines
else
  if bar=... then
    a<=’0’;
    ...
  end if; -- bar=
end if; -- foo=1

if ... then
  if full = '1' then...
  else ...
else
  if full = '1' then...
  else Consistency
  else ...

Arto Perttula 7.11.2016
if not(cnt_r /=10) then
...
else
    if (to_integer(d and "001")=1 then
...)

elsif (not e) and f) and h then
...
else
    if i nor j then
...)

else
...
Statements: CASE

• Alternative to if-clause
• All possible choices must be considered or the last choice must contain others clause

```
CASE expression IS
    WHEN choice1 =>
        statements
    WHEN choice2 =>
        statements
    WHEN others =>
        statements
END CASE;
```

• Example problem with DesignCompiler synthesis tool (from Synopsys):
  – "Error: All possible string values of selector type not covered by choices."

```
signal sel : std_logic;
...
CASE sel IS
    WHEN '0' => result <= a;
    WHEN '1' => result <= b;
END CASE;
```
Statements: CASE (2)

• Example:

```vhdl
CASE state IS
  WHEN "000" =>
    output <= 0;
  WHEN "001" =>
    output <= 1;
  WHEN "010" =>
    output <= 2;
  WHEN "011" =>
    output <= 3;
  WHEN OTHERS =>
    output <= 4;
END CASE;
```

• Use when others => with case
  
  Otherwise, some tools want you to specify also what happens with other std_logic values, e.g., "X11", "1Z1", "WHH", "UUU"
Statements: NULL

- Performs no action
- Is used to explicitly show that no action is to be performed when certain conditions are true
- Example
  ```
  CASE state IS
    WHEN "000" =>
      output <= 1;
    WHEN "001" =>
      output <= 2;
    WHEN "010" =>
      output <= 3;
    WHEN OTHERS =>
      NULL;
  END CASE;
  ```
  NOTE: be careful with NULL statement. In asynchronous/combinatorial processes it may generate latches (prob. unintentional)
Statements: Assertion

• Checks a specific condition and gives message
• Great help in simulation (not synthesizable)
• Also as checkers inside the module
  – E.g., if we assume that always input a < 10, we put an assertion
  – Use assertions to verify the initial assumptions (e.g., generic value range, input value range, known properties and relations)!

• General form:
  ```
  ASSERT condition [REPORT s_expr] [SEVERITY sl_expr]
  ```

• Examples:
  ```
  ASSERT word_count_g < 10
  REPORT “Parameter word count too big (should be <10)”
  SEVERITY warning;
  ```

• Checks by TB done only during verification but assertions are always there in every simulation
• Assertions can be inside or outside a process
Statements: WAIT

- Stops execution of a process or procedure
- General form:
  
  ```
  WAIT [on sensitivity_list] [UNTIL b_expr] [FOR t_expr]
  ```
- Examples:
  
  ```
  WAIT ON a;
  WAIT UNTIL a = 'l';
  WAIT FOR 10 ns;
  WAIT ON a UNTIL b = 'l' FOR 100 ns;
  WAIT UNTIL Clk’EVENT AND Clk = 'l';
  ```
- Another way to do clock-triggered process (not recommended)
- Reserve usage for test benches only
Statements: LOOP

• A way to represent iteration
  – Good for generic and portable code
• 3 different forms possible
  – Prefer for loop
• General form:
  
  ```
  [label:] [iteration_scheme] LOOP
  {sequential_statement}
  END LOOP [label];
  ```

• Examples:

  ```
  FOR i IN 10 DOWNTO 0 LOOP
    a(i) := b(i);
  END LOOP;

  WHILE i < 10 LOOP
    a(i) := a(i) + 1;
    i := i+1;
  END loop;

  LOOP
    clock <= not clock;
    wait for ClockPeriod/2;
    IF error = '1' THEN
      EXIT;
    END IF;
  END LOOP;
  ```

  Preferred loops
  • The bounds are easily seen

  Also in concurrent part

  Inside processes, procedures and functions

  example of clock generation in simulation test benches
Statements: EXIT

• **exit** statement is used to jump out from loops during execution and complete current loop
  – Problem: the exit points in arbitrary locations make the code very hard to comprehend later
  – Note! Hardware implemented for worst case anyway
  – **You cannot have any speed advantage with early exits or such**

• Example:

```plaintext
outer_loop:FOR j IN 0 TO data_width_g-1 LOOP
  inner_loop:FOR i IN 0 TO n_cpu_c-1 LOOP
    IF exit_cond = true THEN
      EXIT inner_loop;
    ELSE
      counter :=counter +1;
    END IF;
  END LOOP inner_loop;
END LOOP outer_loop;
```

Arto Perttula  7.11.2016
Statements: NEXT

- `next` statement is used to stop execution of statements in the loop for this iteration and go to the next iteration
  - Same obfuscation problem as with `exit`

- Example:

```plaintext
outer_loop:FOR j IN 0 TO data_width_g-1 LOOP
  inner_loop:FOR i IN 0 TO n_cpu_c-1 LOOP
    IF next_cond = true THEN
      NEXT inner_loop;
    ELSE
      counter := counter +1;
    END IF;
    END LOOP inner_loop;
  END LOOP outer_loop;
```
Subprograms

- Variables inside subprograms are local
- Variables are valid only during subprogram execution
- Contain sequential statements
- Produce combinatorial logic

Functions:
- Can return one argument (can be compound type)
- All parameters are input parameters

FUNCTION n_one_bits(vec : BIT_VECTOR) RETURN INTEGER IS
  VARIABLE tmp : INTEGER;
BEGIN
  tmp := 0;
  FOR i IN vec’RANGE LOOP
    IF vec(i) = ’1’ THEN
      tmp := tmp+1;
    ELSE
      tmp := tmp;
    END IF;
  END LOOP;
  RETURN tmp;
END n_one_bits;
Subprograms (2)

- **Procedures:**
  - Can contain several input, output and inout parameters

```vhdl
PROCEDURE n_one_bits (SIGNAL vec : IN BIT_VECTOR;
                         VARIABLE int : OUT INTEGER) IS
BEGIN
  int := 0;
  FOR i IN vec'range LOOP
    IF vec(i) = '1' THEN
      int := int+1;
    ELSE
      int := int;
    END IF;
  END LOOP;
END n_one_bits;
```
Procedures And Functions: Call And Return

- Procedure and function call statements:
  ```procedure_name([actual parameters]);```
- Parameters can be positional or named association
  ```myproc1(formal1 => '1',
            formal2 => "111",
            formal3 => 1
        );
myprocedure3; --No parameters```

  -- Similar to component instantiation and also positional assignment:
  ```myproc1(my_signal,my_variable,1);```  
  ```return statement completes execution of innermost procedure or function```

- Examples:
  ```RETURN("1111");```  
  ```RETURN;```
SEQUENTIAL LOGIC
Recap: Combinatorial vs. Sequential Circuit

- **Combinatorial circuit:**
  - No internal state
  - Output is a function of inputs only
  - No latches/FFs
  - Described with processes or concurrent statements

- **Sequential circuit:**
  - With internal state
  - Output is a function of inputs and internal state
  - Synchronous design style with DFFs
    - Includes also combinatorial logic
  - Described with processes
Sequential Statements vs. Logic

• Sequential VHDL statements do not necessarily represent sequential (synchronous) digital logic circuits
• They can describe both combinatorial and synchronous logic circuits
• Modeling combinatorial logic with sequential statements:
  – Sensitivity list of process statements must contain all inputs used in VHDL statements
  – Conditional and selected signal assignments (also if and case statements) must cover all possible branches (to avoid inferring unintentional latches)
Sequential Statements vs. Sequential Logic

• The two basic types of synchronous elements are
  1. D-type latch (level sensitive memory cell)
  2. D-type flip-flop (edge-triggered memory cell)
• The main disadvantage of latches (instead of flip-flops) is that static timing analysis (STA) of synthesized circuits can be very complex
  – Do not use latches! => Inferred latches indicate very likely a bug
  – They also complicate manufacturing tests
Sequential Statements vs. Sequential Logic (2)

• Flip-flops are inferred by signal assignment in a process that detects some signal’s edge
  – Note! Assigning in reset branch creates DFF as well, even if the clk’event branch does not touch that signal
• Example of rising edge detection:

  IF (clk’EVENT AND clk='1') THEN
  statements...
  END IF;

input ports, signals, generics, constants

statements

D Q

assigned signals and output ports

clk
Designing Synchronous Circuits

- All *signals and ports* that are assigned a value in process containing
  
  `[ELS]IF (clk'EVENT AND clk='1') THEN
  are implemented as registers!
  – Note that a register consists of several flip-flops

- This is because these assignments take place only at clock edge
  – This is exactly how the flip-flops work: they load the input value on clock edge

- Explicit usage of flip-flop component is not recommended

- Many signal types can infer registers. Integers become registers (32b) as well as `std_logic_vectors` and own defined types.
  – A flip-flop is instantiated for each bit
  – Integer ranges should be defined
Designing Synchronous Circuits (2)

- Remember the concept of RTL design
- With VHDL synchronous design style, this is actually just what you do
  - Define what happens before a register and where that data goes

```
ELSIF clk'event AND clk = '1' THEN  -- rising clock edge
    result_r <= (a_in XOR b_in) + c_in;
END IF;
```

Do this operation and move the result to result register

The code shows that result_r is a register, postfix _r is meant for human reader

Inputs may come from registers or other comb logic, but example code does not show where
Example: model a single rising-edge triggered D flip-flop

```
library ieee;
use ieee.std_logic_1164.all;
entity dffr is
  port(
    clk: in std_logic;
    reset: in std_logic;
    d: in std_logic;
    q: out std_logic
  );
end dffr;
architecture arch of dffr is
begin
  process (clk,reset)
  begin
    if (reset='1') then
      q <= '0';
    elsif (clk'event and clk='1') then
      q <= d;
    end if;
  end process;
end arch;
```

<table>
<thead>
<tr>
<th>reset</th>
<th>clk</th>
<th>q*</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>q</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>q</td>
</tr>
<tr>
<td>0</td>
<td>f</td>
<td>d</td>
</tr>
</tbody>
</table>

- If reset is active
- Process activated when there is a change in value of reset or clk
- If reset was not active AND we have a rising edge in signal 'clk'

Reminder: No code before if or after end if
Example 2: Multiply-Accumulate

Step 1: Define the Entity

• Entity is the same for all architecture variants
  – However, clk and rst_n are ignored in combinatorial

entity mac is
  generic (  
    data_width_g : integer := 4);
end mac;

port (  
  clk : in std_logic;
  rst_n : in std_logic;
  a_in : in std_logic_vector(data_width_g-1 downto 0);
  b_in : in std_logic_vector(data_width_g-1 downto 0);
  c_in : in std_logic_vector(data_width_g-1 downto 0);
  mac_out : out std_logic_vector(data_width_g*2-1 downto 0)
) ;
end mac;

Data width is parameterized, we’ll return to this...

Default range is 4-1 downto 0 = 4 bits

Double width due to multiplication
(in numeric_std, two 4-bit number addition results in 4-bit result, not 5-bit.)

Default: 4*2-1 = 7, 7 downto 0 = 8 bits
MAC: 3 Possible Architectures

1) Combinatorial circuit:

\[
\begin{align*}
A_{in} & \rightarrow * \\
B_{in} & \rightarrow + \\
C_{in} & \rightarrow Mac_{out}
\end{align*}
\]

2) Synchronous circuit:

\[
\begin{align*}
A_{in} & \rightarrow * \\
B_{in} & \rightarrow + \\
C_{in} & \rightarrow D\uffer Q \\
& \rightarrow Mac_{out}
\end{align*}
\]

3) Pipelined synchronous circuit:

\[
\begin{align*}
A_{in} & \rightarrow * \\
B_{in} & \rightarrow D\uffer Q \\
C_{in} & \rightarrow D\uffer Q \\
& \rightarrow + \\
& \rightarrow D\uffer Q \\
& \rightarrow Mac_{out}
\end{align*}
\]
MAC: Combinatorial Architecture

• Combinatorial circuit has no flip-flops

architecture rtl of mac is
begin   -- rtl

mac_comb: process (a_in, b_in, c_in)
  variable result_v : unsigned(data_width_g*2-1 downto 0);
begin   -- process mac_comb
  -- note that result_v does not depend on its previous value
  -- => this is safe usage of variable to beautify the code
  result_v := unsigned(a_in)*unsigned(b_in)+unsigned(c_in);
  mac_out <= std_logic_vector(result_v);

end process mac_comb;

end rtl;

Multiplication precedes addition, but parentheses would make that more obvious.

Here we don’t care about overflow in the sum.
MAC: Synchronous Architecture

- Synchronous circuit with single output flip-flop

architecture rtl of mac is
begin -- rtl

mac_sync : process (clk, rst_n)
    variable result_v : unsigned(data_width_g*2-1 downto 0);
begin -- process mac_comb
    if rst_n = '0' then -- asynchronous reset (active low)
        mac_out <= (others => '0');
    elsif clk'event and clk = '1' then -- rising clock edge
        result_v := unsigned(a_in)*unsigned(b_in)+unsigned(c_in);
        mac_out <= std_logic_vector(result_v);
    end if;
end process mac_sync;
end rtl;

A register is generated for mac_out since:
- It is a signal (port out) and
- It is assigned a value within the clock region

Result_v is just a wire and not implemented as register since:
- it is a variable that does not depend on its previous value
MAC: Pipelined

- Pipelined version has two register stages

```
architecture rtl of mac is
  signal mul_r : unsigned(data_width_g*2-1 downto 0); -- internal signal (reg)
  signal c_r : unsigned(data_width_g-1 downto 0); -- internal signal (delay reg)

begin -- rtl

  mac_pipe : process (clk, rst_n)
  begin -- process mac_comb
    if rst_n = '0' then -- asynchronous reset (active low)
      mac_out <= (others => '0');
    elsif clk'event and clk = '1' then -- rising clock edge
      c_r <= unsigned(c_in);
      mul_r <= unsigned(a_in)*unsigned(b_in);
      mac_out <= std_logic_vector(mul_r + c_r);
    end if;
  end process mac_pipe;

end rtl;
```

 Registers implemented for each signal. Order of assignments does not matter here

The mul_r is updated in previous statement. However, signal values do not change until the next clock edge with the clock region (unlike variables). Therefore, mac_out functions correctly as it uses mul_r(t) while the preceding statement produced mul_r(t+1).
MAC: Second Pipelined Version

4) Pipelined synchronous circuit without outregister:

```
architecture rtl of mac is
  signal mul_r : unsigned(data_width_g*2-1 downto 0);
  signal c_r : unsigned(data_width_g-1 downto 0);
begin
  mac_pipe : process (clk, rst_n)
  begin
    if rst_n = '0' then
      -- Note that intermediate pipeline registers do not have to be nullified under reset
      c_r <= (others => '0');
      mul_r <= (others => '0');
    elsif clk'event and clk = '1'
    c_r <= unsigned(c_in);
    mul_r <= unsigned(a_in)*unsigned(b_in);
    end if;
  end process mac_pipe;
  mac_out <= std_logic_vector(mul_r + c_r);
end rtl;
```

Quiz: why would this made more sense than v3?
A: Smaller area. Delay of ADD doesn’t probably increase the critical path.

Mac_out moved out of the clock region causes that no register generated for it but just a wire coming from an adder. It must NOT be reset.
System Reset

• Brings the system into known state
  – At-start-up
  – After crash
• Known state = deterministic values for flip-flops in the system
  – Concerns sequential logic (sequential processes)
• Memory (SRAM, DRAM) contents usually remain in reset
  – But you cannot count on that!
  – Contents are lost when power is shutdown
  – Will be initialized/reset explicitly location by location (often with SW)
• Flip-flop’s value may be set
  1. Asynchronously via special input pin – Asynchronous reset
  2. Synchronously via D input – Synchronous reset
  3. Synchronously via D input – Normal operation
DFF with Asynchronous Reset

- Common way
  1. Reset
     - Drives the **set or clear inputs** of a flip-flop
     - The other always inactive
     - Output Q updated even if no clock signal present
     - Avoid logic gates in reset signal if possible
  2. Normal operation
     - Value of D appears on Q after each rising edge of clock
     - D driven by some combinatorial logic (or input pins)
     - Clock driven by input pin, PLL, or other special clock generation logic. Not by your own logic!

- Thou shalt not mess with reset or clock signals!
async_rst : process (clk, rst_n)
begin
-- No signal assignments here
if rst_n = '0' then
    -- Assign bootup values for DFFs in this branch.
    -- Use constant values only! Do not read signals or
    -- input ports! Avoid comb. logic in reset signal.
elsif clk'event and clk = '1' then
    -- Assign the values of normal operation in this branch.
    -- No other conditions to elsif. Otherwise you’ll get
    -- comb. logic into the clock tree which is disastrous.
    -- Use nested if-statements instead.
end if;
-- No else of elsif branches here
-- No signal assignments here
end process async_rst;
DFF with Synchronous Reset

• Not so common way but OK
• Uses simpler DFFs that have no asynchronous inputs
• Now, also the reset needs valid clock signal
  – PLL may take milliseconds before locking and providing clean clock

1. Reset
  – Sets/clears the D input
  – Minor increase to critical path
  – Routing the reset signal throughout the chip is easier than in asynchronous case

2. Normal operation as previously
DFF with Synchronous Reset (2)

sync_rst : process (clk)
-- use exactly this sensitivity list
begin
  -- No signal assignments here
  if clk'event and clk = '1' then
    if sync_rst_n = '0' then
      -- Assign bootup values for DFFs in this branch.
      -- No other conditions here.
      -- Assign constant reset values only.
    else
      -- Assign the values of normal operation in this branch.
      end if;
      -- No elsif branches here.
    end if;
    -- No elsif branches here
  end if;
-- No else of elsif branches here
-- No signal assignments here
end process sync_rst;
Timing Example: Asynchronous Reset + Synchronous Clear

- DFF output changes only on clock edge when reset is not active
- Reset signal set output value to zero immediately
- Synchronous clear signal set output to zero on clock edge

```
reset : process (clk, rst_n)
begin
  -- process reset
  if rst_n = '0' then
    Q1 <= (others => '0');
    Q2 <= D;
  elsif clk'event and clk = '1' then
    Q1 <= D;
    Q2 <= (others => '0');
  end if;
end process reset;
```

Note! This timing in simulation.
In real HW, Q2 does not care about reset and just stores D on every clk edge (also values 0 and 1).

Undefined signal value on reset.
The True Devil – Never Do!

ENTITY bad_counter IS
  PORT (  
    reset, clk, inc : IN STD_LOGIC;  
    cnt : BUFFER INTEGER RANGE 0 TO 4);  
END bad_counter;
ARCHITECTURE example OF bad_counter IS
BEGIN -- Example
  PROCESS (clk, reset, inc, cnt)
  BEGIN -- PROCESS
    IF reset = '0' THEN -- asynchronous reset (active low)
      cnt <= 0;
    ELSIF inc = '1' THEN
      cnt <= cnt+1;
    ELSIF clk'EVENT AND clk = '1' THEN -- rising clock edge
      cnt <= cnt-1;
    END IF;
  END PROCESS;
END example;

What is wrong?
Generates a pseudo-random machine.
Few Further Notes about Reset

- Terms "Asynchronous/Synchronous Reset" refers how the reset signal is connected to the DFFs
- All flip-flops must leave the reset state simultaneously
- Hence in both cases, the reset signal must be synchronous to the used clocks!
  - Reset from external pin is specifically synchronized first!
  - Otherwise, some DFFs may start little earlier (one cycle) than others ➔ Medium-scale catastrophe
  - Synchronization logic covered later
- In large chip, the reset operation may be complex sequence
  - Detect stable voltage and clock first, reset blocks in certain order…
Few Further Notes about Reset (2)

- Reset signal has similar setup and hold time constraints as D input to avoid metastability problems
- Synchronous reset via D input is good for synchronizers at chip’s inputs
  - The incoming values propagate through the synchronizer during reset
  - However, asynchronous reset needs constant values. Hence, synchronized inputs are reset differently depending on their behavior. Hard to maintain such logic.
- Both styles are used and there is no clear winner
  - But you can, of course, have endless and heated discussion on their benefits 😊
GENERICS AND GENERATE STATEMENTS
Generics

- Pass *instance-specific* information to an entity
- Ability to parameterize models using generics
- The values of generic parameters must be computable at design time
  - Dynamic changes are not possible
- Use generics instead of hard-coded values in interface!
  - Not all mac units are 32-bit wide, some are 16-bit and some 64-bit

```vhdl
entity mac is
  generic (data_width_g : integer := 4);

  port (clk : in std_logic;
       rst_n : in std_logic;
       a_in : in std_logic_vector(data_width_g-1 downto 0);
       b_in : in std_logic_vector(data_width_g-1 downto 0);
       c_in : in std_logic_vector(data_width_g-1 downto 0);
       mac_out : out std_logic_vector(data_width_g*2-1 downto 0)
      );
end mac;
```
Generics (2)

architecture testbench of tb_mac is

component mac
  generic (
    data_width_g : integer := 4);
  port (
    clk : in std_logic;
    rst_n : in std_logic;
    a_in : in std_logic_vector(data_width_g-1 downto 0);
    b_in : in std_logic_vector(data_width_g-1 downto 0);
    c_in : in std_logic_vector(data_width_g-1 downto 0);
    mac_out : out std_logic_vector(data_width_g*2-1 downto 0);
  );
end component;

-- values to assign for component generics
constant data_width_c : integer := 8;

-- Internal signals to/from DUT
signal clk : std_logic;
signal rst_n : std_logic;
signal a_to_mac : std_logic_vector(data_width_c-1 downto 0);
signal b_to_mac : std_logic_vector(data_width_c-1 downto 0);
signal c_to_mac : std_logic_vector(data_width_c-1 downto 0);
signal mac_result : std_logic_vector(data_width_c*2-1 downto 0);

begin -- testbench
  -- component instantiation
  DUT: mac
  generic map (data_width_g => data_width_c)
  port map (clk => clk,
             rst_n => rst_n,
             a_in => a_to_mac,
             b_in => b_to_mac,
             c_in => c_to_mac,
             mac_out => mac_result);
  ...
end testbench;

Note:

- By using generic, this same code for mac can be used for any data_width_g and it works without modifications
- If generic value is not specified in the instantiation, the default is used (in this case 4, as shown in previous slide’s entity declaration). Danger danger.
Generic Use Case Examples (3)

1. Widths of port and internal signals
2. Sizes of register files and memories
3. Addresses
   - E.g., own address, shared region in memory…
4. Enables for features
   - Whether or not instantiate some sub-module
   - Encrypt/decrypt/both, little/big-endian
5. File paths in test benches
   • However, large number of generics makes verification harder
     - How to test all (at least most) combinations?
     - How to detect and prevent illegal combinations?
     - New users might also get confused
   • Default values simplify instantiation, BUT
     - Responsible integrator always sets ALL generics explicitly
Summary of Generics

• They are great! Use them!
Statements: GENERATE

- **Very important for hierarchical design**
- **generate** statement makes possible to replicate (FOR GENERATE)
  - Component instantiations
  - Concurrent statements
- Also conditional replication/instantiation is possible (IF GENERATE)
  - Whether a component is instantiated or not
  - Whether concurrent statement/process is implemented or not
  - Note that there is no ELSE GENERATE
- Example: DMA – direct memory access
  - Parametrized amount of DMA channels
  - Real code example from a research project
Example Background: Idea of DMA Component

- DMA copies data to/from dual-port memory from/to network
  - Much faster than CPU-controlled communication
- CPU sets: src addr, dst addr, data amount, and command
- Example TX: Send 100 bytes starting from 0x200 in DPRAM to network addr 0x5000
- Example RX: If there’s data coming from network with addr 0xF000, put it to addr 0x164 in DPRAM, Interrupt CPU when 50 bytes have been received.
- There can be multiple RX transfers (=channels) pending – use FOR GENERATE
Example of For-Generate Component

-- Component
cannels : for i in 0 to n_chans_g-1 generate

i_rx_chan : n2h2_rx_chan
generic map (  
data_width_g => data_width_g,
id_g => i,
...  
addr_cmp_hi_g => hibi_addr_cmp_hi_g)
port map (  
clk => clk,
rst_n => rst_n,
avalon_addr_in => mem_addr_r(i),
...  
irq_out => irq_chan_r(i)
);

end generate channels;

Parameters:

<table>
<thead>
<tr>
<th>Channel 0</th>
<th>Channel 1</th>
<th>Channel n_chans_g-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mem_addr_r(0)</td>
<td>...</td>
<td>Mem_addr_r(n_chans_g-1)</td>
</tr>
<tr>
<td>Irq_chan_r(0)</td>
<td>...</td>
<td>Irq_chan_r(n_chans_g-1)</td>
</tr>
</tbody>
</table>

HW:

Ports:

- Hardware parameterized with generics, same "base" for all
- In example, each channel has
  - Different identifier id_g
  - Same signal widths data_width_g and addr_cmp_hi_g
- I/Os can be mapped to different signals
- Outputs must be mapped to different signals!
- In example, each channel has
  - Different mem_addr_r and irq_chan_r (std_logic_vector position in 2D-array)
  - Same clk and rst_n
• Note that the generic parameters can be result of a function, from a table etc.
  – Very powerful
  – Statically computed values, no dynamic parameters

```
i_hibi_nios : hibi_wrapper_r4
  generic map (
    id_g => get_prior(i),
    base_id_g => 2**id_width_c-1,
    id_width_g => id_width_c,
    addr_g => get_addr(i),
    rel_bus_freq_g => determine_clk_freq(true, fifo_sel_g, bus_freq_c),
    rel_agent_freq_g => determine_clk_freq(false, fifo_sel_g, cpu_freq_c),
    prior_g => get_prior(i),
    arb_type_g => arb_type_c(i),
    fifo_sel_g => fifo_sel_g,
    inv_addr_en_g => 0,
    -- ...
  )
```

-- purpose: To determine proper relative frequencies
-- depending on the syncmode

```
function determine_clk_freq (
  constant device_is_bus : boolean;
  constant syncmode : integer range 0 to 3;
  constant frequency : integer)
return integer is
begin
  --syn
  case syncmode is
    when 1 =>
      --asyn fast
      if device_is_bus then
        return bus_freq_c;
      else
        return hibi_sync_freq_c;
      end if;
    when others =>
      --syn
      return frequency;
  end case;
  return 0;
end determine_clk_freq;
```

a) Generic value determined with function
   • Get_prior(i)
   • Determine_clk_freq(..)

b) Generic value read from a constant table
   • Arb_type_c(i)
TIPS AND PITFALLS
Get Rid of Red Signals

• Red signals in wave form indicate serious problems!
• Your first task is to remove all red signals!
• There are 2 reasons
  1. Signal does not have any value – undefined 'U'
     – Any logic reading that signal will malfunction
     – Likely caused by
       a) Missing initialization during reset
       b) Failed component instantiation during simulation bootup
         • "Component DUV is not bound"
         • This is only warning in ModelSim
         • This may go unnoticed if there are many messages and designer is not careful
     – Note that some red signals are derived from those failing due to above reasons
Get Rid of Red Signals (2)

2. Signal has conflicting values – result is ’X’
   – It is driven in 2+ places in code that will be executed in parallel
     a) Two processes (including reset part of synchronous process)
     b) In process and concurrent assignment (=outside all processes)
     c) Multiple concurrent assignments
     d) Port of subcomponent and signal in higher level component
   – Again, some red signals are derived from those failing due to above reasons

• Reminder: Your first task is to remove all red signals!
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb_conflict is
end tb_conflict;

architecture behavioral of tb_conflict is

constant period_c : time := 10 ns;

signal clk : std_logic := '0';
signal rst_n : std_logic;
signal a_tb_duv : std_logic;
signal test0_duv_tb : std_logic;
signal test1_duv_tb : std_logic;
signal counter_r : unsigned ( 10 downto 0);

begin -- behavioral

DUV : entity work.var1
port map (
clk => clk,
rst_n => rst_n,
in => a_tb_duv,
test0_out => test0_duv_tb,
test1_out => test1_duv_tb);

clk <= not clk after period_c/2;
rst_n <= '0', '1' after 5 period_c;
test0_duv_tb <= '0'; -- aargh! Conflict with DUV's output

counter_r <= (others => '0') after 50 * period_c;

sync: process (clk, rst_n)
begin
-- process conflicting reset
if rst_n = '0' then

test1_duv_tb <= '0'; -- aargh! Conflict with DUV's output

-- Other reset values here
counter_r <= (others => '0');

elsif clk'event and clk = '1' then

-- Regular sync. logic here
counter_r <= counter_r+1;

if to_integer(counter_r) = 20 then

test1_duv_tb <= '1'; -- aargh!

elsif to_integer(counter_r) = 30 then

test1_duv_tb <= 'Z'; -- suspicious but works

elsif to_integer(counter_r) = 35 then

test1_duv_tb <= 'H'; -- suspicious but works

elsif to_integer(counter_r) = 40 then

test1_duv_tb <= 'L'; -- suspicious but works

end if;

end if;

end process sync;

counter_r <= (others => 'Z'), (others => '0') after 50 * period_c;

end behavioral;

gen_stimulus: process
begin -- process

wait for period_c/2; -- align to rising edge
wait for period_c/3; -- wait "a little"
wait for 10 * period_c; -- aargh! a_tb_duv not given any value

a_tb_duv <= '0';
wait for 10 * period_c;
a_tb_duv <= '1';
wait for 10 * period_c;
a_tb_duv <= '0';
wait for 10 * period_c;
a_tb_duv <= '0';
wait for 10 * period_c;
a_tb_duv <= '1';
wait for 10 * period_c;
a_tb_duv <= '0';
wait for 10 * period_c;
a_tb_duv <= '0';
wait for 10 * period_c;
a_tb_duv <= '1';
wait for 10 * period_c;
a_tb_duv <= '0';
wait for 10 * period_c;
a_tb_duv <= '0';
wait for 10 * period_c;
a_tb_duv <= '1';
wait for 10 * period_c;
a_tb_duv <= '0';
wait for 10 * period_c;
a_tb_duv <= '0';
wait for 10 * period_c;

assert false report "Simulation ended successfully" severity failure;
wait;
end process gen_stimulus;
end behavioral;
Get Rid of Red Signals (3)

Undefined value 'U' at input ruins both outputs

Conflict between '1' driven by DUV and '0' driven by TB. Short-circuit between GND and VDD. Zap!

Values 'Z', 'H' or 'L' driven by TB do not cause conflicts.

Conflict between values given in process and in concurrent assignment.
architecture rtl of ac is
  signal x_r : integer;
  signal y_r : integer;
  signal z : integer;
begin -- rtl

  value: process (clk, rst_n)
    variable k_v : integer;
    begin -- process value
      if rst_n = '0' then
        x_r <= 0;
        y_r <= 0;
      elsif clk'event and clk = '1' then
        k_v := to_integer(unsigned(a_in));
        x_r <= to_integer(unsigned(a_in));
        y_r <= k_v + x_r;
      end if;
    end process value;

  z <= y_r;
end rtl;

Process is triggered at clock edge.
Variable is updated instantly at the clock edge.
Signal is updated just after the clock edge

Simulator view:

Note: difference in variable and signal timing has been exaggerated in fig. In simulator, they look the same

Note: variables are not necessarily visible at all in older simulator’s wave window! Debugging gets complicated…

y_r(t) = k_v(t) + x_r(t), i.e. a_in(t)+a_in(t-1)

z <= y_r is a continuous assignment, happens instantly (not within the clock region)
architecture rtl of ac is
    signal x_r : integer;
signal y_r : integer;
signal z : integer;
begin  -- rtl

value: process (clk, rst_n)
    variable k_v : integer;
begin  -- process value
    if rst_n = '0' then
        x_r <= 0;
y_r <= 0;
    elsif clk'event and clk = '1' then
        k_v := to_integer(unsigned(a_in));
x_r <= to_integer(unsigned(a_in));
y_r <= k_v + x_r;
    end if;
end process value;
z <= y_r;
end rtl;

• Note that variable is used within clock sensitive process
  – It’s value is updated only when there’s event on clock (in simulator!)

• However, when implemented, k_v is part of the comb. logic “cloud” that constantly executes
  – Short-hand notation for part of the comb. logic
  – Immediate value assignment
In this case, the variable is not very useful, but used for demonstration purposes only.
Example (4): Simulation vs. Real HW

architecture rtl of ac is
  signal x_r : integer;
  signal y_r : integer;
  signal z : integer;
begin  -- rtl

  value: process (clk, rst_n)
    variable k_v : integer;
begin  -- process value
    if rst_n = '0' then
      x_r <= 0;
      y_r <= 0;
    elsif clk'event and clk = '1' then
      k_v := to_integer(unsigned(a_in));
      x_r <= to_integer(unsigned(a_in));
      y_r <= k_v + x_r;
    end if;
  end process value;

  z <= y_r;
end rtl;

Input changes and internal wire k_v has the exactly the same value.

In HW:

Note: simulator’s view does not fully match real world, although the register x_r and outputs y_r and z are the same!
VHDL Pitfalls

1. Identifiers
   – VHDL isn’t case sensitive (e.g., Input and input are the same)
   – (But some tool’s are…)

2. Misspelled If statement
   – ELSIF written as ELSE IF

3. Wrong string delimiters
   – ’0001’ instead of ”0001”, or ”0” instead of ’0’
   – 0 instead of ’0’ or vice versa

4. Misused reserved words
   – Reserved words used as object names: IN, OUT, BUFFER, AND, NAND, OR

5. Incomplete case statement
   – VHDL requires all conditions to be presented
   – Conditions must be locally static (determined at compile-time)
VHDL Pitfalls (2)

6. Expression evaluation
   – Following operations have equal precedence: AND, OR, NAND, NOR, XOR
   – Example of illegal expression: a OR b AND c
   – Corrected expression: a OR (b AND c)

7. Signal assignment from multiple sources
   – E.g., signal is reset in sequential process, but it is assigned outside the process
   – You must define only one driver for a signal!
   – A process, a concurrent statement…
   – Three-state/Weak logic are allowed at chip I/O
Notes on Syntax

• Signal declarations are before architecture’s `begin` clause
• Signal assignment is `<=`, variable assignment is `:=`
  – Depends on the object in the left hand side
  – Source type can be different `sig <= var; var := sig`
• Equality comparison is `=`, inequality `/=`
• Entity ports and generic declarations
  – Lines end with semicolon `;` except the last one
• Component instantiation
  – Assignments end with comma `,` except the last one
  – Mapping symbol points to right `=>`
• Emacs is a dear friend and helps a lot in these!

```vhdl
i_rx_chan_1 : n2h2_rx_chan
  generic map (  
    data_width_g => data_width_g,
    ...  
    addr_cmp_hi_g => hibi_addr_cmp_hi_g)  
  port map (  
    clk => clk,
    rst_n => rst_n,
    avalon_addr_in => mem_addr_r(i),
    ...  
    irq_out => irq_chan_r(i)
  );
```

No semi-colon
Points to the right
No comma allowed
Mistakes in Simulation vs. Synthesis

- The process sensitivity list is incomplete!
  - May hide bugs in the design
  - Synthesis ignores sensitivity list but simulation relies on it's completeness
  - Always use compile option `check_synthesis` with ModelSim

- Using non-synthesizable structures or datatypes

- E.g., using don’t care operator
  - E.g., `when "1--" => ...`
  - May simulate well (ModelSim supports) but does not synthesize!

- Behavior of reset and three-state logic 'Z'

- Timing of variables

- Delays and indefinite loops are not synthesizable
VHDL Pitfall: NULL-Statement in CASE

```
-- Null statement in case statement
-- at combinatorial process
-- produces latch in synthesis to hold the last value.

ENTITY null_statement IS
    PORT (sel : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
           output : OUT STD_LOGIC);
END null_statement;
ARCHITECTURE example OF null_statement IS
BEGIN -- Example
    PROCESS (sel)
    BEGIN -- PROCESS
        CASE sel IS
            WHEN "00" => output <= '0';
            WHEN "10" => output <= '1';
            WHEN OTHERS => NULL;
        END CASE;
    END PROCESS;
END example;
```
Top 20 Errors in VHDL

- This is a list of the most common VHDL errors. The Top 10 account for about 40% of all errors. The Top 20 account for about 60% of all errors.
- Source: Doulos VHDL Golden reference guide

<table>
<thead>
<tr>
<th>The Top 20: Errors 1-10</th>
<th>The Top 20: Errors 10-20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Missing or misplaced begin in architecture / process / subprogram</td>
<td>Missing sensitivity list / wait statement in process</td>
</tr>
<tr>
<td>Swapping &lt;= := and =</td>
<td>Sensitivity list and wait statement in same process</td>
</tr>
<tr>
<td>Missing or extra end if / end case / end loop / end process etc</td>
<td>Same name used twice</td>
</tr>
<tr>
<td>Wrong quotes around characters / strings / integers</td>
<td>elseif or endif (instead of elsif and end if respectively)</td>
</tr>
<tr>
<td>Incompatible types in assignments / operators</td>
<td>Using the wrong user defined identifier</td>
</tr>
<tr>
<td>Missing or extra ; at end of declaration / statement</td>
<td>Extra / missing / mistyped character</td>
</tr>
<tr>
<td>Misspelt identifier</td>
<td>Missing signals in sensitivity list</td>
</tr>
<tr>
<td>Undefined signal / variable / constant</td>
<td>Mismatched vector lengths</td>
</tr>
<tr>
<td>Missing library / use</td>
<td>Using a reserved identifier</td>
</tr>
<tr>
<td>Wrong separator , / ; / : /</td>
<td>Reading out ports</td>
</tr>
</tbody>
</table>

Arto Perttula 7.11.2016
EDA Tool Pitfalls

• These apply to many languages, though
1. Some tools behave **strangely** if there are white spaces in paths
   - E.g., My Documents\rocket.vhd, source codes\cool_stuff.vhd
   - E.g., EDA tool just freezes or doesn't find a file (without telling what it’s looking for…)
   - Scandinavian alphabets å,ä,ö,Å,Ä,Ö are sometimes confusing also
2. Tool project files store file paths as absolute instead of relative ones
   - E.g., rom_init =/home/jeppe/sw/prom.hex instead of ../sw/prom.hex
   - Hard/impossible to move project files
     • To other designers or your customer!
     • To directory without white spaces in its name
   - More common than you think!
   - The same path definition may be stored into multiple files
     • Correction is tedious at best, impossible in case of binary files
   - Re-creating a project file from scratch and imitating the settings is error-prone, annoying and waste of time
EDA Tool Pitfalls (2)

4. Version incompatibilities
   – Opening files in different version of the tool fails
     • Worst part: some parts seem to work, so you'll start working until some things silently fail and you'll go crazy
     • Worst part #2: file does not tell which version it is. Go figure!
   – ToolA v1.2 works well only with ToolB 2.71 but not with older or newer ones
     • Worst part: vendor does not offer older versions anymore
   – You must document clearly which versions you have used

5. Strange problems when developing in both Linux and Windows (and Cygwin)
   – Both / and \ are used in path definitions
   – Different line breaks (use dos2unix or unix2dos)
   – Latin-1 vs. UTF-8 vs. some other character encoding
   – Certain tools/licenses are available only for Linux and some Windows, and you'll need both

6. Windows both allows and does not allow file name to be over 255 characters
   – Tool might be able to create a temporary file with such a name...
   – But the same tool cannot read or remove it!
EDA Tool Pitfalls (3)

7. Too few user privileges
   – Some tools store user’s information into installation directory…
   – But user doesn't necessarily have write access to that folder, only the admin folks who installed the SW

8. Vendor lock-in: moving to other tools is too difficult
   – Customer is tied to certain tool or chip vendor because it is too laborious and costly to move all legacy code to new environment
     a) Proprietary file format that cannot be converted to others
     b) Standard format is broken on purpose by incorporating “helpful additions”
        • Might prevent opening and editing the files in other programs
     c) Some tools pose stricter rules which code constructs are acceptable
     d) Organization shares some data only in a stupid format, say .pdf instead of .xlsx

9. Node-locked licenses are tied to certain computer, e.g., it’s MAC address
   – Let’s hope that one computer doesn't break…
   – You’ll may need a wired LAN, not just any internet connection
EDA Tool Pitfalls (4)

10. Internal database gets corrupted
   - Some things work, some things do not, go figure
   - Remove the generated files and try again
   - > rm syn/db; rm sim/work
   - Therefore, you must keep your sources strictly separated from the project setting files and generated ones

11. User does not read error and warning messages carefully
   - Not actually a tool problem… ;)
   - However, vague messages are tools fault
     • ”A file not found” Which file? Where did you look for it?
Summary

• Combinatorial logic described with concurrent statements and processes. Sequential logic only with processes.
• Generics, if-generate, for-generate and assertions are great
• Loop bounds, signal widths, signals slicing indices, and generic values must be known at compile time (=synthesis-time)