

IN 3160, IN4160

**Simulation: testbenches, processes, signals and variables**  
**VHDL: conditional statements and structure**

Yngve Hafting



# Messages

# VHDL Testbench

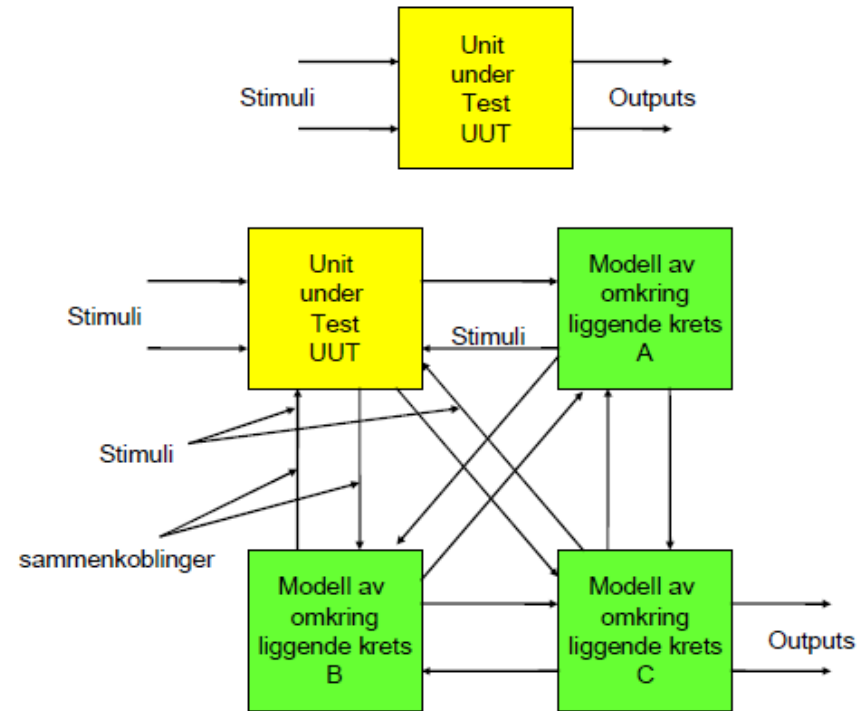
- Stimuli generator and test environment in VHDL
- Powerful possibilities for simulation
  - File I/O
    - Reading test patterns from file
    - Writing results to file and compare it to a file with the correct answers
    - The file with the correct answers can also be read, and comparison between the result and the blueprint can be executed in the testbench
  - Can build in modules for surrounding circuits
    - Especially important if we have a two-way communication between UUT (Unit Under Test) and surrounding circuits (Handshake signals)
- Gives simulator independent testbench.

# Testbench

- In a test bench you can
  - include other modules (that are more or less verified)
  - or provide data for one unit at a time.
- Often we rely on simulation models created by others.
  - Bus functional models etc.
  - *Can be required to achieve certifications in industry..*

- Note:

- *In our cocotb setup, we need one-tooplevel VHDL file*
  - *We can make a new toplevel with several models inside*



# VHDL Testbench structure

- Libraries
- Empty entity (can have generics, but no IO)
- Component declaration for each entity that is a part of the test
- Signals for all ports we would like to manipulate
- Component instantiation
  - «DUT» is likely the module we would like to test
- One or more processes that
  - evaluates output vectors
  - set input test vectors at specific time intervals
  - reports findings and results to screen or file.

```
-- Library declarations
library IEEE;
use IEEE.std_logic_1164.all;
use ...

-- Empty entity
entity tb_design is
end entity tb_design;

architecture behavioral of tb_design is

    -- component declaration(s) --
    component X_OR is ...
    end component;

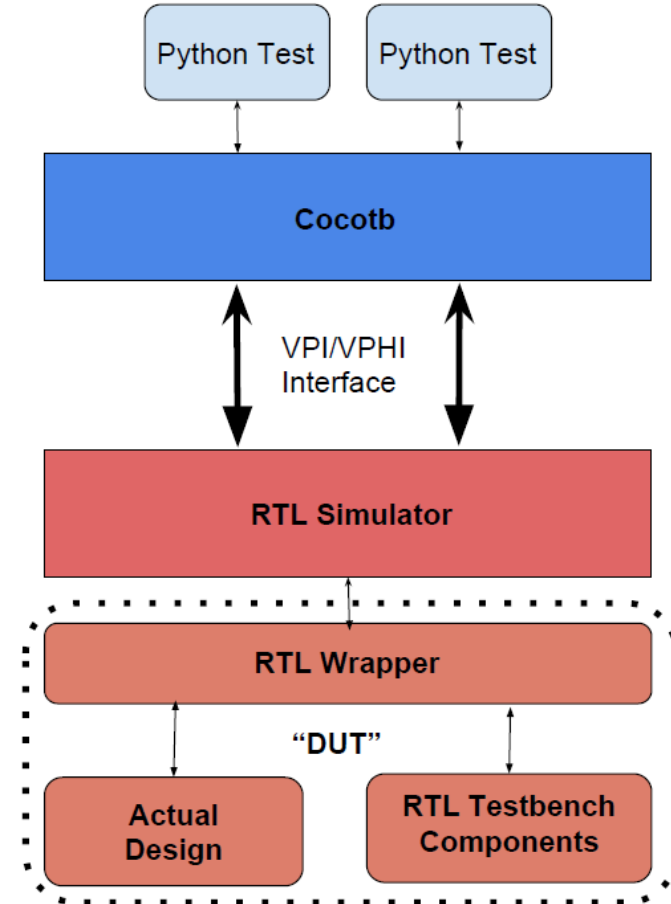
    -- signal declaration --
    signal ...

begin
    -- DUT instantiation --
    DUT : x_or
    generic map(...) port map(...);

    -- standalone tests -
    ...
    -- stimuli generation --
    stimuli: process is
    begin
        ...
        report ("TESTING FINISHED!");
        std.env.stop;
    end process;
end architecture behavioral;
```

# Cosimulation: Cocotb and python testbenches

- *Cosimulation*: Design and testbench simulated independently
- Communication through VPI/VHPI interfaces, represented by cocotb "triggers".
- When the Python code is executing, **simulation time is not advancing**.
- When a trigger is awaited, the testbench waits until the triggered condition is satisfied before resuming execution.
- Available triggers include:
  - Timer(time, unit): waits for a certain amount of simulation time to pass.
  - Edge(signal): waits for a signal to change state (rising or falling edge).
  - RisingEdge(signal): waits for the rising edge of a signal.
  - FallingEdge(signal): waits for the falling edge of a signal.
  - ClockCycles(signal, num): waits for some number of clocks (transitions from 0 to 1).



# Cocotb: Coroutines, tasks and triggers

- All signals in the design hierarchy can be probed and set in python
- "async def" is used when defining coroutines
- Multiple triggers can be used
  - enable tests running independently
    - `cocotb.start_soon(<coroutine>)`
      - Starts the coroutine as soon when "awaiting" the next time
      - Used to start clock generation,
    - `await(<task/trigger>)` <https://docs.python.org/3/library/asyncio-task.html>
      - Waits until the task is finished or trigger condition occurs
      - `await ReadOnly()` is used to let signals settle after other triggers such as `await Edge(<dut.signal>)`
        - » You do not want to read signals before all delta delays are completed...

```
async def stimuli_generator(dut):  
    ''' Generates all data for this tesbench'''  
    for i in range( 2**len(dut.input)):  
        await start_soon(set_stimuli(dut, i))
```

```
async def compare(dut):  
    ''' Compares simulated output with predicted output '''  
    while 1:  
        await Edge(dut.input) # Test on each new input  
        await ReadOnly()     # Wait for output to settle  
        assert dut.output == predictor(dut), (  
            "output ({out}) is not as predicted: XOR({inp})"  
            .format(out=dut.output.value, inp=dut.input.value))  
        dut._log.info(  
            "output ({out}) is XOR({inp})"  
            .format(out=dut.output.value, inp=dut.input.value))
```

```
@cocotb.test()  
async def main_test(dut):  
    """ Starts comparator and stimuli """  
    dut._log.info("Running test...")  
    start_soon(compare(dut))  
    await start_soon(stimuli_generator(dut))  
    dut._log.info("Running test...done")
```

## Cocotb keywords

- Cocotb documentation:  
<https://docs.cocotb.org/en/stable/>
- Coroutines generally in python:  
<https://docs.python.org/3/library/asyncio-task.html>



## Processes and the event queue (in simulation)

- Simulation uses an event queue to keep track of what happens.
- A process is invoked as a result of a change in one of the signals in the sensitivity list.
  - The whole process is "run" through «within that delta delay».
  - Each signal assignment is added to the queue of delta delays
    - **Only changes in signals that are in the sensitivity list will trigger the process again.**
  - Variable updates does not trigger any new events.
    - (They are updated immediately...)

# How processes work with signals and variables...

- A process must work in a predictable, deterministic way for both creating and simulating circuits
- Signals
  - Represent physical wires and drivers in the architecture
    - A wire can only have a single voltage at any given time.
  - Are updated once in a process invocation
    - This happens only at process exit.
      - No intermediate values are held.
      - A value that has been changed cannot be read as changed within the process.
    - When assigned multiple times within a process, the latest will be given priority
      - Allows for default values
    - Makes inferring storage elements (FFs+latches) deterministic and comprehensible.
- Variables
  - Variables *are local* to the process.
    - They must be both assigned and read within a process
  - Their value is *intended for intermediate purposes*
    - Making code more readable by turning complex statements into several simpler statements
    - By taking value(s) that can be used within the process
  - They *can* be given values multiple times within a single process invocation
    - *Doing so- is generally not a good idea*
  - Placement determines whether they will infer storage elements such as latches and flipflops!
    - **Using variables for storage is considered bad practice** in most circumstances

# VHDL processes

- Process sensitivity
  - Decides when a process is invoked in *simulation*
  - «**SHOULD not**» **interfer with how HW is made...**
    - *Do not trust this..!*
  - Good practice:
    - Use keyword `all` for combinational logic: `process (all) is...`
    - Use clock (and reset when asynchronous) for sequential logic: `process (clk)...`

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
entity My_thing is
  port(A: in STD_LOGIC;
       F: out STD_LOGIC
  );
end entity My_thing;
```

```
architecture Behavioral of My_thing is
  signal b : STD_LOGIC;
begin
  signal_update: process(a)
  begin
    if A = '1' then b <= '1';
    else b <= '0';
    end if;

    if b = '1' then F <= '1';
    else F <= '0';
    end if;

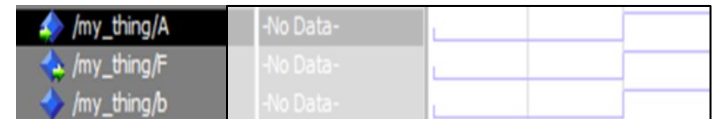
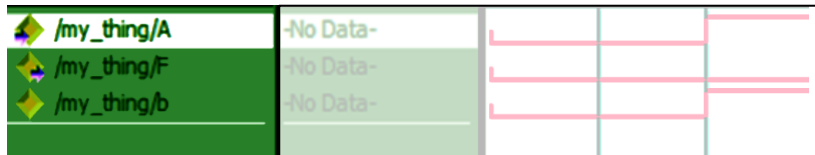
  end process;
end architecture Behavioral;
```

# Sensitivity list

## What happens with F?

Assume a changes from '0' to '1'

```
signal_update: process(a,b)
begin
  if a = '1' then
    b <= '1';
  else
    b <= '0';
  end if;
  if b = '1' then
    f <= '1';
  else
    f <= '0';
  end if;
end process;
```



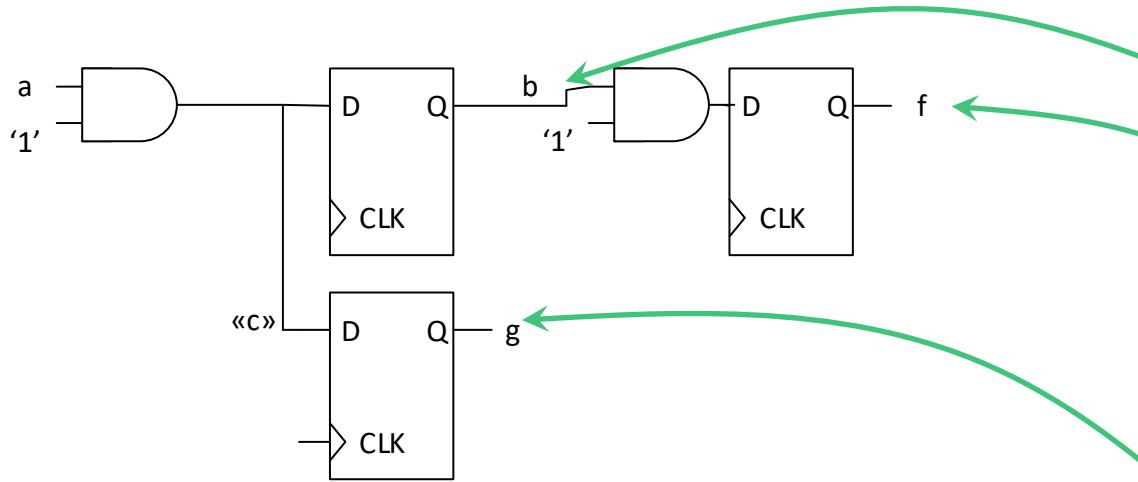
# Signals vs. variables (sequential logic example)

- Exercise:
- Assume all signals are 0, then
  - signal a changes from 0 to 1.
- On which clock cycles does f and g change value; first, second, third?

*Try for 1 minute:*

*Time's up...*

```
signal_var_update : process(clk)
    variable c : std_logic;
begin
    if rising_edge(clk) then
        if a = '1' then
            b <= '1';
            c := '1';
        else
            b <= '0';
            c := '0';
        end if;
        if b = '1' then
            f <= '1';
        else
            f <= '0';
        end if;
        if c = '1' then
            g <= '1';
        else
            g <= '0';
        end if;
    end if;
end process;
```



NOTE: *c could* be assigned multiple places in the process.

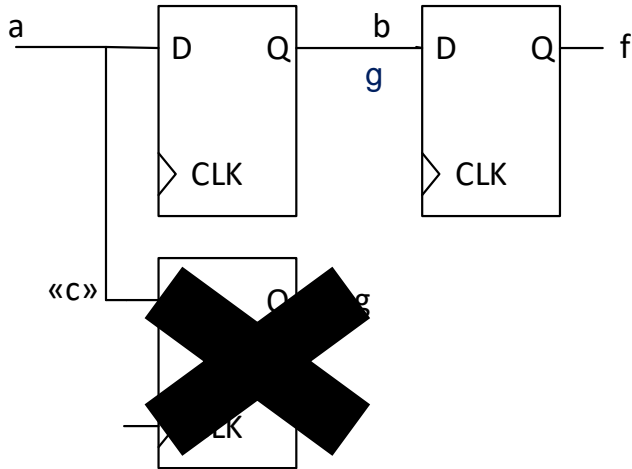
How would that affect the diagram..?

**Variables update «immediately»**  
**Signals are assigned «where» the process ends**  
when the process statement updates as a whole

```
signal_var_update :  
process (clk)  
    variable c : std_logic;  
begin  
    if rising_edge (clk) then  
        if a = '1' then  
            b <= '1';  
            c := '1';  
        else  
            b <= '0';  
            c := '0';  
        end if;  
        if b = '1' then  
            f <= '1';  
        else  
            f <= '0';  
        end if;  
        if c = '1' then  
            g <= '1';  
        else  
            g <= '0';  
        end if;  
    end if;  
end process;
```

## Digression:

- Simplified...



```
signal_var_update :  
process (clk)  
  variable c : std_logic;  
begin  
  if rising_edge (clk) then  
    b <= a;  
    c := a;  
    f <= b;  
    g <= c; -- g <= a  
  end if;  
end process;
```

```
signal_var_update :  
process (clk)  
  variable c : std_logic;  
begin  
  if rising_edge (clk) then  
    if a = '1' then  
      b <= '1';  
      c := '1';  
    else  
      b <= '0';  
      c := '0';  
    end if;  
    if b = '1' then  
      f <= '1';  
    else  
      f <= '0';  
    end if;  
    if c = '1' then  
      g <= '1';  
    else  
      g <= '0';  
    end if;  
  end if;  
end process;
```

## Default values in processes

```
: architecture Sequential2 of priority is
: begin
:   process (a) is
:   begin
:     valid <= '1';
:     if a(3)='1' then
:       y <= "11";
:     elsif a(2)='1' then
:       y <= "10";
:     elsif a(1)='1' then
:       y <= "01";
:     elsif a(0)='1' then
:       y <= "00";
:     else
:       valid <= '0';
:       y <= "00";
:     end if;
:   end process;
: end architecture Sequential2;
```

- Ensures we always have an output value (*avoiding latches*).
- Be reasonable with use of “default” values in a process
  - Does only change where it’s necessary
  - This works because processes are compiled sequentially...
    - The *last assignment* within the *process* will take precedence
  - Don’t bury default values within nested ifs...
    - Readability and maintainability suffer if you do..
- Default values are commonly used for state machine outputs
  - typically active in one state only...



# Signals and variables

- Signals
  - A signal can be used within the whole architecture
  - Connect to other architectures through the entity ports
  - Changes value when simulation exits a process (or statement)
- Variables
  - Variables are declared and *only used locally* within a process (function or procedure)
  - Assigned using “:=“ (Ex: var := ‘1’ ;)
  - Unlike a signal the variable changes value **immediately** in simulation
    - Immediately = based on position, *read from top to bottom*.
    - *can have multiple values* within one process.
  - Variables are useful to keep intermediate results in algorithms
    - **Subprograms initialize variables every run.**
    - **Process variables initialize once, when simulation starts**
- Both signals and variables can be used for storage
  - Both FFs and latches.
  - Variables that are read «before» written will accomplish this = BAD PRACTICE!...

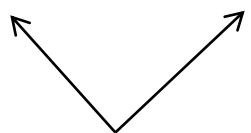
## Rule of thumb:

- **Signal** for all registers (FFs)  
“out is a **signal** that can be read outside the entity”
- **variable** for everything else that does not need to be visible outside...

## Simulation of VHDL models

- The *time* datatype is defined in `std.vhd`
- The function *now* returns current simulation time

```
type time is range -2147483647 to 2147483647
units
  fs;
  ps = 1000 fs;
  ns = 1000 ps;
  us = 1000 ns;
  ms = 1000 us;
  sec = 1000 ms;
  min = 60 sec;
  hr = 60 min;
end units;
```



Implementation  
specific

- Simulation-time starts at 0

## Simulation specific code (Non synthesis)

- Assertions (*will be ignored by synthesis tools*)
  - Can be used to create error messages and notifications based on results
  - Ex: `assert (e = c) report( "e differs from c") severity error`
    - Read «assert» as «if not <boolean expression> then» [report...]
- `wait for <time>` (*will be ignored by synthesis tools*)
- Warning...:
  - «`wait for / wait on <signal>`» can be used in synthesizable code
    - Makes sequential (latched or flip-flopped) logic.
    - Better: use the IEEE1164 keyword «`rising_edge`» or «`falling_edge`» to ensure operation

# Suggested reading, corresponding assignments

## Combinational logic

- D&H
  - 3.6
  - 6.1, 6.2, 6.3 (p105-109)
  - (6.4-6.9 p110- 120 .. Not syllabus)
  - 6.10 p121-123
  - 7.1 p 129-143
  -

## Verification

- D&H:
  - 2.1.4 p 27
  - 7.2 p 143-148
  - 7.3 p 148-153
  - 20.1 p 453 – 456
- Oblig 1: «Design Flow»
  - See canvas for further instruction.
- Oblig 2: «VHDL»

IN 3160, IN4160

**VHDL**

**conditional statements and structural design**

Yngve Hafting



In this course you will learn about the design of advanced digital systems. This includes programmable logic circuits, a hardware design language and system-on-chip design (processor, memory and logic on a chip). Lab assignments provide practical experience in how real design can be made.

*After completion of the course you will:*

- understand important principles for design and testing of digital systems
- **understand the relationship between behaviour and different construction criteria**
- **be able to describe advanced digital systems at different levels of detail**
- **be able to perform simulation and synthesis of digital systems.**

**Goals for this lesson:**

- Know conditional statements in VHDL
  - .
  - how to implement these structures using VHDL
    - *If, case, when-else, select*
    - *Loops*
    - *Type casting*
    - *Shift operators*
    - *Dataflow vs RTL descriptions*
- Know how to generate complex structures in VHDL
  - generate

## Section overview

- VHDL:
  - Sensitivity list
  - Signals and variables example
  - If, case, when-else, select
  - Loops
  - Structural coding
    - Generate
    - Generics

## Next lesson: Building blocks

Decoders vs encoders

Decoder

Multiplexer

Encoders

Arbiters

Shifters

Comparators

ROM

RAM

## If and case in **VHDL Processes**

- **if** and **case** are used much like in other programming languages like C, Java etc.
  - *Their similarity in syntax may lead to errors if we do not understand how they work in digital circuits...*
  - If-tests can test on multiple signals/variables
    - built in priority
  - Case-tests uses single signal/variable (vector=OK)
    - No built in priority because the same signal are being used everywhere in the test



**If**

- Must be in process
- **Multiple conditions**
- **Multiple targets**
- prioritizes

- First option has priority
  - (think of two-input multiplexers)
- Can be used to infer latches and Flipflops
  - FF when edge triggered (**if** `rising_edge(c1k)` **then...**)
  - **Latch when not sufficiently specified!**
    - This is a trap, avoid this!
- Can be nested using «**elsif**»
  - Can replace any other conditional statement
    - *Not recommended!*
  - Avoid deep nesting
  - ~4 degrees should be maximum...

## If example (*all input specified*):

inp1	inp2	a	b
1	1	1	1
1	0	1	0
0	1	0	Latched
0	0	0	Latched

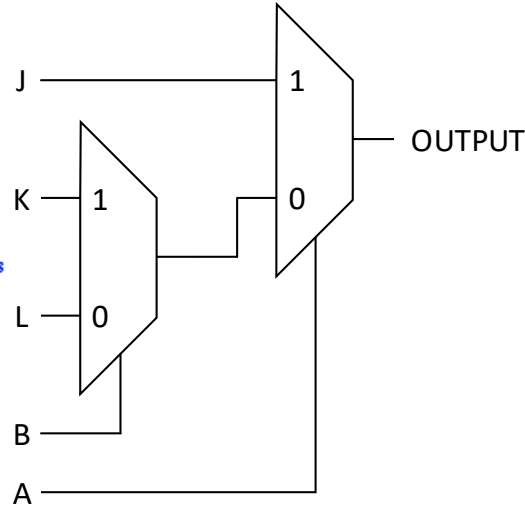
```
process(all) is
begin
  if inp1 then
    if inp2 then
      a <= '1';
      b <= '1';
    else
      a <= '1';
      b <= '0';
    end if;
  else
    a <= '0';
  end if;
end process;
```

```
process(all) is
begin
  if inp1 then
    a <= '1';
    b <= inp2;
  else
    a <= '0';
    -- b ass. missing
  end if;
end process;
```

***Always specify all outputs for all conditions of inputs!***

```
entity My_thing is
  port(A,B,J,K,L: in STD_LOGIC;
        OUTPUT: out STD_LOGIC);
end entity My_thing;
```

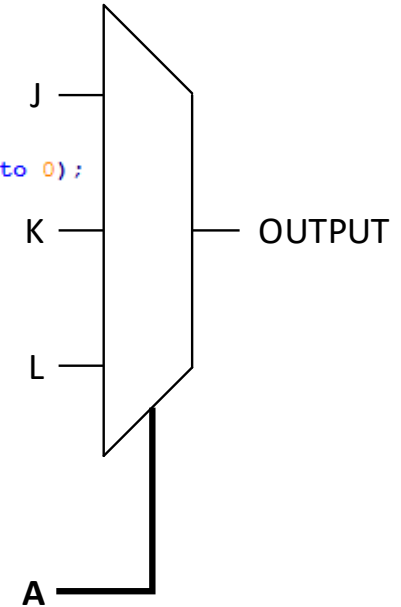
```
architecture prioritized of My_thing is
begin
  process(all)
  begin
    if A = '1' then
      OUTPUT <= J;
    elsif B = '1' then
      OUTPUT <= K;
    else
      OUTPUT <= L;
    end if;
  end process;
end architecture prioritized;
```



## if../case..

```
entity My_thing is
  port(A : in STD_LOGIC_VECTOR(1 downto 0);
        J,K,L: in STD_LOGIC;
        OUTPUT: out STD_LOGIC);
end entity My_thing;

architecture nonpri of My_thing is
begin
  case A is
    when "01" =>
      OUTPUT <= J;
    when "10" =>
      OUTPUT <= K;
    when others =>
      OUTPUT <= L;
  end case;
end architecture nonpri;
```



## If nesting vs. chaining (*using elsif*)

```
process(all) is
begin
  if (input = 4d"1") then
    isprime <= '1';
  else
    if (input = 4d"2") then
      isprime <= '1';
    else
      if (input = 4d"3") then
        isprime <= '1';
      ...
    end if;
  end if;
else isprime <= '0';
end if;
end process;
```

```
process(all) is
begin
  if (input = 4d"1") then isprime <= '1';
  elsif (input = 4d"2") then isprime <= '1';
  elsif (input = 4d"3") then isprime <= '1';
  ...
  else isprime <= '0';
  end if;
end process;
```

# If nesting for priority – danger zone

Sometimes it can make sense to use nesting

- clocked processes and state machines
- It is easy infer latches
  - When not all input options are covered
  - When some output is not covered for all options

*Consider other options when creating CL*

- *improve readability*
- *Reduce risk for latches*
- *It is OK to nest other statements within if...*
  - *select ...*
  - *when ... else*
  - *case ...*

```
process(all) is
begin
  if (inp1 = a) then
    if (inp2 = b) then
      if (inp3 = c) then
        <statement 1>
        <statement 2>
      else
        <statement 3>
      end if;
    end if;
  else
    <statement 4>
  end if;
end process;
```

# Example

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

entity latches is
port(
  invec : in std_logic_vector(1 downto 0);
  outvec : out std_logic_vector(3 downto 0);

  input : in std_logic;
  out1, out2 : out std_logic
);
end entity latches;
```

- **Nesting if-statements will conceal these errors easily,** thus providing an endless source of errors

```
architecture poor of latches is
begin
  -- if invec = "11" => outvec is latched
  missing_input: process(all) is
  begin
    if invec = "00" then
      outvec <= "0000";
    elsif invec = "01" then
      outvec <= "1110";
    elsif invec = "10" then
      outvec <= "0110";
    end if;
  end process;

  -- if input='1' then out2 is latched.
  -- if input='0' then out1 is latched.
  missing_output: process(all) is
  begin
    if input then
      out1 <= '1';
    else
      out2 <= '0';
    end if;
  end process;
end architecture poor;
```

# Case

- Must be in process
  - **single input vector**
  - **Multiple targets**
  - Every alternative has same priority
- 
- Every option for *input* must be declared
    - ‘**when others**’ can be used
      - be wary of changes in input *type*...
    - Can infer latches too...
      - When not defining all outputs for all inputs
  - Matching case- «case?»
    - Allowes for don't care's

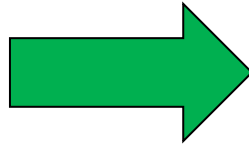
```
process(input) is
begin
  case input is
    when x"1" | x"2" | x"3" | x"5" | x"7" |x"b" | x"d" =>
      isprime <= '1';
    when others => isprime <= '0';
  end case;
end process;
```

**The typical use-case for case is state machines.**

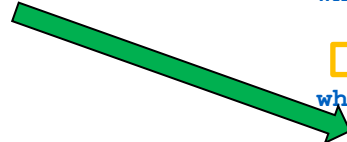
Case is excellent when you want to set several output vectors depending on one state vector.

## Case creating latches:

Default values can be a good solution when using case statements.



'null' statement should only be used in CL when using default values for all outputs.



```
process(input) is
begin
  isprime <= '0';
  isfour <= '0';
  case input is
    when x"1" | x"2" | x"3" | x"5" | x"7" | x"b" | x"d" =>
      isprime <= '1';
      isfour <= '0';
    when x"4" =>
      isprime <= '0';
      isfour <= '1';
    when others =>
      null;
  end case;
end process;
```



= latch inferred



## When ... else

- Can be used concurrently (outside processes).
- **Multiple conditions**
- **Single target**
- prioritizes

- Can replace if statements for *single target*
- Can infer FF's/latches
- Compact
  - Suitable when complexity is low

```
isprime <=
  '1' when input = x"1" else
  '1' when input = x"2" else
  '1' when input = x"3" else
  '1' when input = x"5" else
  '1' when input = x"7" else
  '1' when input = x"b" else
  '1' when input = x"d" else
  '0';
```

```
q <= '0' when reset else 'd' when rising_edge(clk);
```

```
a <= b when en;
```

^^ always keep 'else' in mind...

## With ... select

- Can be used concurrently
- **single input vector**
- **Single target**
  - Must have all input cases defined

- Can also infer latches
  - *Least likely*

- Feedback obvious 😊

- Compact and readable

```
with input select isprime <=  
  '1' when x"1" | x"2" | x"3" | x"5" | x"7" | x"b" | x"d",  
  '0' when others;
```

```
with a select g <=  
  16d"1" when 16d"1",  
  16d"4" when 16d"2",  
  16d"8" when 16d"3",  
  g when others;
```

# If, case, when ... else, with select - summary

- When in doubt...
  - Try **‘with...select’**
    - This will force you to make visible choices.
- *Only* use **‘if’**...
  - When you need to prioritize conditions...
  - and have multiple targets
    - Typically used for clocked processes.
- It is fine to **use select...** or **when/else** inside **if** and **case**
  - *Do you need if inside if?..*
  - *Case inside case? ..*
  - Readability suffers when nesting several levels of if or case

Statement	Targets	Conditions	Process
<b>if</b>	<i>Multiple</i>	<i>Multiple</i>	Required
<b>case</b>	<i>Multiple</i>	<b>Single</b>	Required
<b>when ... else</b>	<b>Single</b>	<i>Multiple</i>	Optional
<b>with ... select</b>	<b>Single</b>	<b>Single</b>	Optional

Whatever you choose,  
keep the following in mind:

*define*

- all outputs *for*
- all conditions

# Loops in VHDL

- Both simulation and synthesizable code
- Three types
  - Simple loop- until exit
  - While- loop condition is true
  - For loop
    - Counted
      - Numbers or elements/ ‘range
    - Loop parameter static
      - Can be increased using ‘next’
      - ‘next when <condition>’
- ‘exit’+(optional loop\_label)
  - Can be used in all loops
  - Innermost loop is default
  - Nested loops: use label

```
--SIMPLE LOOP--  
variable i: integer := 0;  
...  
loop  
    statements;  
    i := i + 1;  
    exit when i = 10;  
end loop;  
  
--WHILE LOOP--  
variable i: integer := 0;  
...  
while i < 10 loop  
    statements;  
    i := i + 1;  
end loop;  
  
--FOR LOOP--  
for i in 1 to 10 loop  
    statements;  
end loop;  
  
--FOR LOOP2--  
type frukt_type is (eple, pære, banan);  
...  
frukt_loop: for f in frukt_type loop  
    statements;  
    when <condition1> next frukt_loop;  
    when <condition2> exit frukt_loop;  
end loop;
```

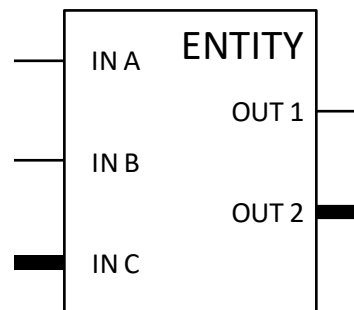
# Entity/architecture

- Entity and architecture are the two most fundamental building blocks in VHDL
- Entity
  - Connection to the surroundings
  - Port description
    - Input/output/bi-directional signals
- Architecture
  - Describes behavior
  - An entity can have many architectures
  - Can be used to describe the circuit on several levels of abstraction:
    - Behavioral (for simulation)
    - RTL (Register Transfer Level)
    - Dataflow
    - Structural
      - Post synthesis (netlist)
      - Post Place & Route (netlist + timing)

**REPETITION**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity My_thing is
    generic(width: integer := 8);
    port(INA, INB : in STD_LOGIC;
         INC : in STD_LOGIC_VECTOR(width-1 downto 0);
         OUT1: out STD_LOGIC;
         OUT2: out STD_LOGIC_VECTOR( width/2 - 1 downto 0)
    );
end entity My_thing;
```



# Generics

- In addition to the port description an entity can have a generic description
- **Generics** can be used to make parameterized components (generic)
  - can be used for structural information
    - both synthesis and simulation
  - can be used for timing information
    - for simulation only
  - Example 1:
    - Time delay can vary between circuits, but the behavior is the same
  - Example 2:
    - The number of bits can vary between circuits, but the behavior is the same

```

24: entity And2 is
25:   generic (delay : DELAY_LENGTH := 10 ns);
26:   port (x, y : in BIT; z: out BIT);
27: end entity And2;
28:
29: architecture ex2 of And2 is
30: begin
31:   z <= x and y after delay;
32: end architecture ex2;

```

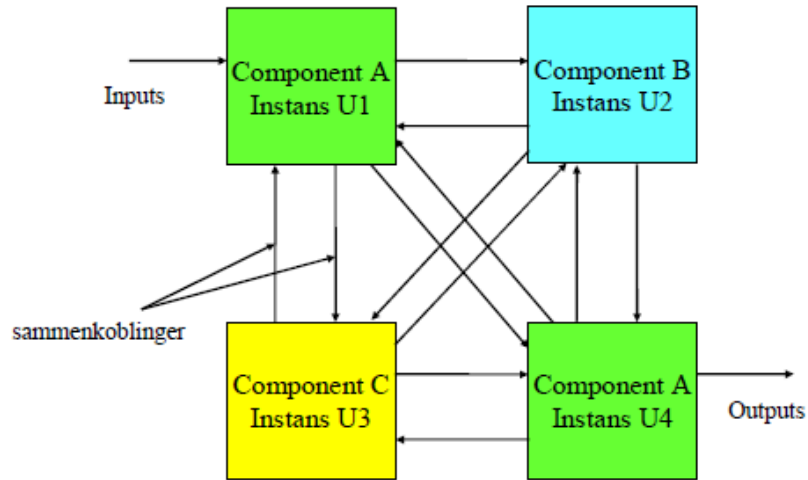
```

architecture Structural of My_tb is
  component And2
  .. port( x,y : in BIT; z : out BIT);
  end component;
  signal a,b,c : BIT;
begin
  MY_COMP1: And2
  .. generic map(delay => 1 us);
  .. port map(x=>a, y=>b, z=>c);
end architecture Structural;

```

DELAY\_LENGTH is a subtype of the type time from the predefined (always in use) package “std”

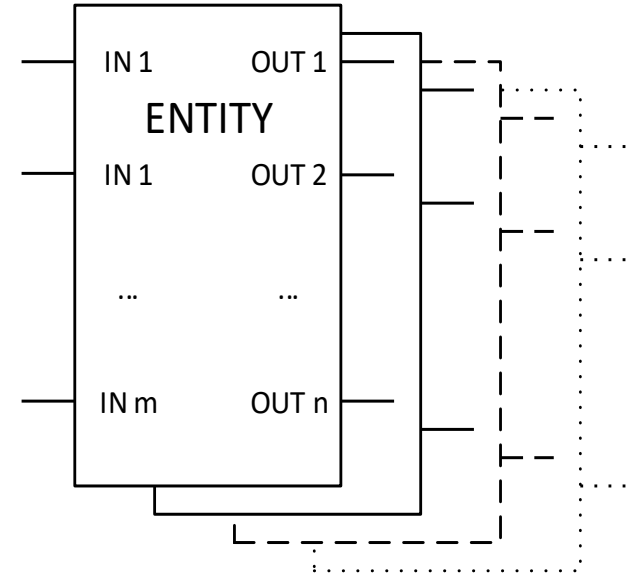
# Structural design



- Every Component instance has an underlying Entity/architecture pair
- We can easily re-use «entities»
- We *can* make a hierarchic design with as many levels we want
  - Try keep design hierarchy manageable...

## Structural design

- Reuse of modules (entities and architectures)
- Generic modules (generics)
  - For example scalable bus widths
  - Configurable functionality
- Breaking up big designs to smaller and more manageable building blocks
  - Think functional blocks
  - Connection of functional blocks (entities/components/modules)
- Easier to collaborate within a design team
  - Well defined interface between modules
- Any entity-/architecture pair can be used as a building block in a structural description
  - Pairing of components





## Structural design (netlist)

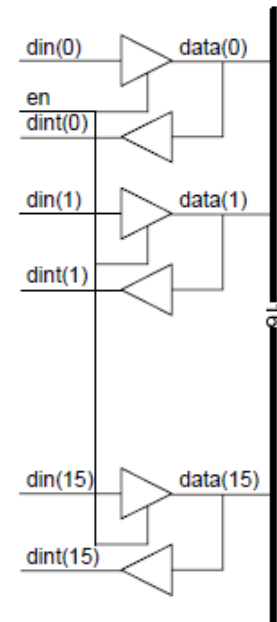
```
25: architecture netlist2 of comb_function is
26:
27:   component And2 is
28:     port (x, y : in BIT; z: out BIT);
29:   end component And2;
30:
31:   component Or2 is
32:     port (x, y : in BIT; z: out BIT);
33:   end component Or2;
34:
35:   component Not1 is
36:     port (x : in BIT; z: out BIT);
37:   end component Not1;
38:
39:   signal p, q, r : BIT;
40:
41: begin
42:   g1: Not1 port map (a, p);
43:   g2: And2 port map (p, b, q);
44:   g3: And2 port map (a, c, r);
45:   g4: Or2 port map (q, r, z);
46: end architecture netlist2;
```

- A netlist is a description of components used, and their connections
  - Synthesizing *is* creating a netlist using the available primitives for a (PL/ASIC) device.
  - The top level in larger designs is normally purely structural
- Component declaration pick up entities from «work» library
- The last compiled architecture are being used unless specified different
- Port mapping:
  - «Association» *can* be done by position
    - Will lead to disasters when making changes.
  - named association is less error prone.  
ex: `g1: Not1 port map (x=>a, z=>p);`

# Structural design with **generate** statement

- **generate** - loop
  - can build multiple components .
  - requires indexable parameters in some connected signals
  - non-indexable signals will be connected to all instances
- Example: Bidirectional bus
- **generate** can be used to conditionally build structures
  - **if** and **case** + **generate**
    - **Conditions must be resolved at compile-time**
      - only constants/generics, *no signals*
      - *This is not runtime-reconfiguration...*

```
bidir_bus_inst: for i in 0 to 15 generate
  buft_inst: buft port map (data(i),en,din(i));
  ibuf_inst: ibuf port map (dint(i),data(i));
end generate;
```



# Suggested reading

- D&H 7.1- 7.3 p129-153