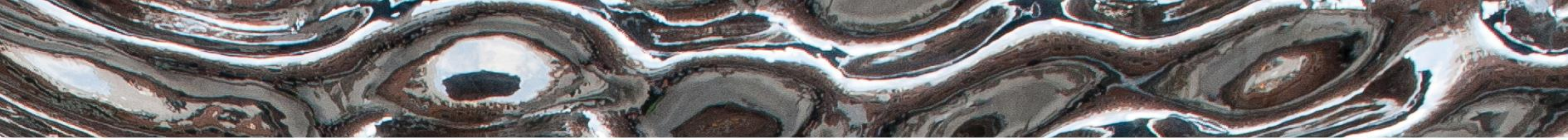


UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

IN3160, IN4160

Introduction to VHDL +Basic layout for VHDL

Yngve Hafting

Messages:

- Questions before we start?

Course Goals and Learning Outcome

https://www.uio.no/studier/emner/matnat/ifi/IN3160/index-eng.html

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 behavior 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 the basic structure of VHDL
  - Know which design entities there are
  - Know how assignment and statements works
  - Know the basic functionality of processes
  - Be able to create designs using VHDL
  - Know the relation between phsyical signals and their declaration.
  - Know the difference between basic coding styles
- Know basic layout principles
  - Guidelines for capital letters
  - Basic layout types
  - Principles for indentation, commenting, naming, punctuations

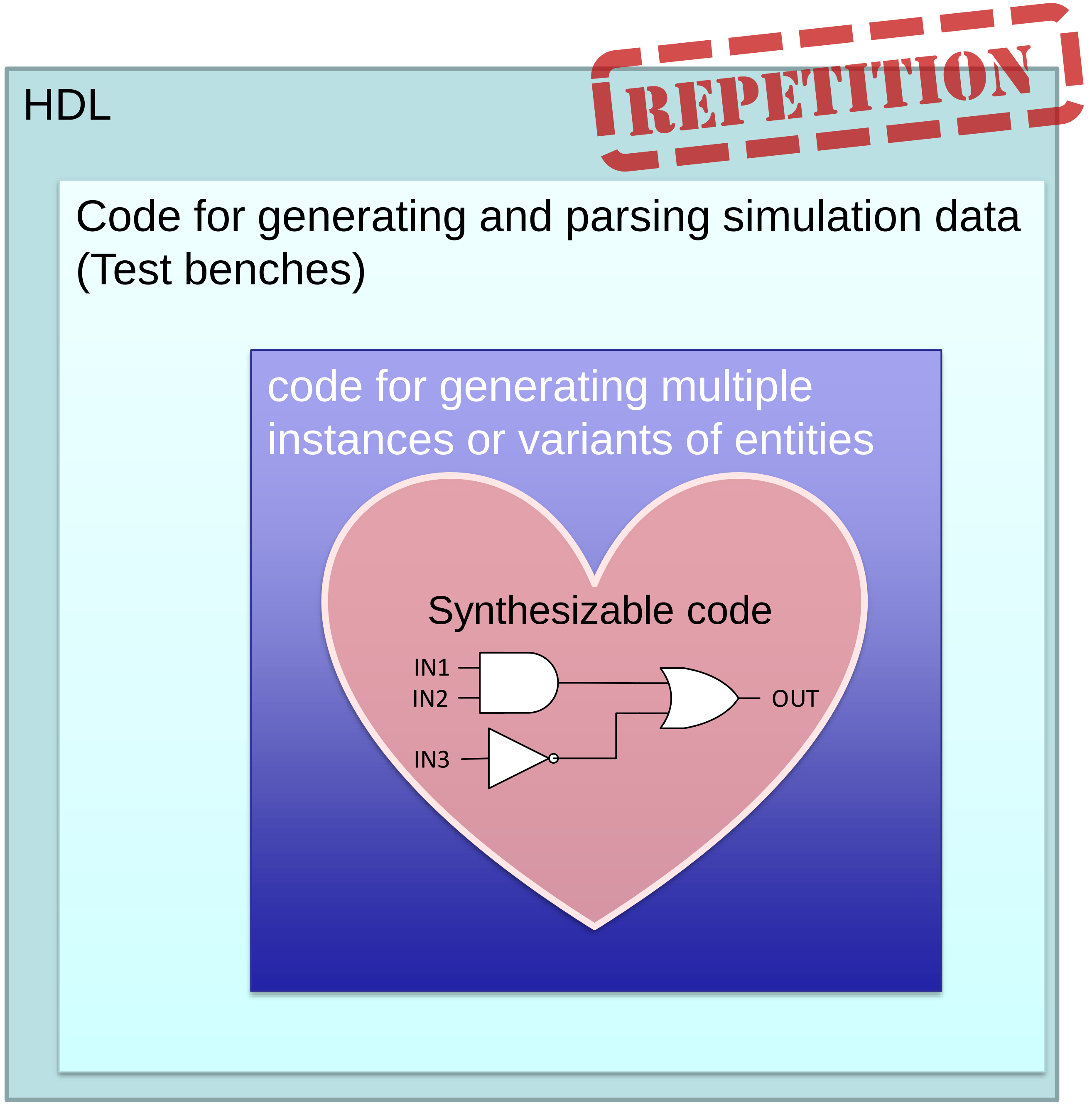
Overview

- Repetition
- VHDL Structure (Partly repetition from IN2060)
  - Design entities
  - Statements
  - Signals, variables, vectors
  - Processes
  - Libraries
  - STD_LOGIC
  - Operators
  - String literals
- Code layout principles
- Next lesson: Combinational logic

- Assignments and suggested reading for this week

HDL

- VHDL = VHSIC HDL:
  - Very High Speed Integrated Circuit **Hardware Description Language**
  - **The purpose is to generate digital circuits, and verify their function through simulation.**
  - **Synthesizable (realizable) code generates circuits that are always on = work concurrently (in parallel)**.
  - Code for simulation include things such as file I/O which cannot be synthesized.
  - Testbenches can and will use some synthesizable elements, but will in general look more like other sequential languages, and use sequential statements.
    This may be confusing at times...
  - VHDL does come with several libraries


REPETITION
HDL
Code for generating and parsing simulation data (Test benches)
code for generating multiple instances or variants of entities
Synthesizable code
IN1
IN2
IN3
OUT

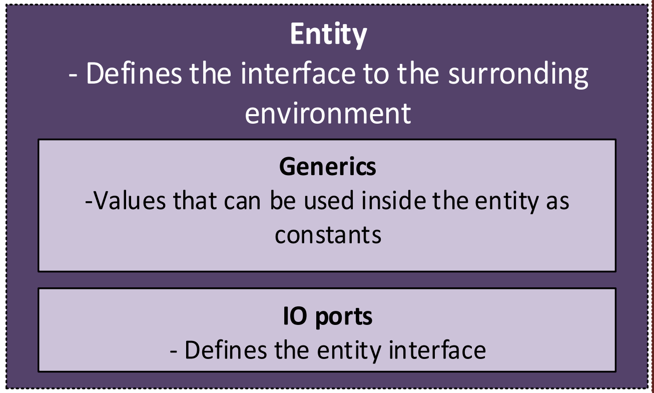
VHDL structure

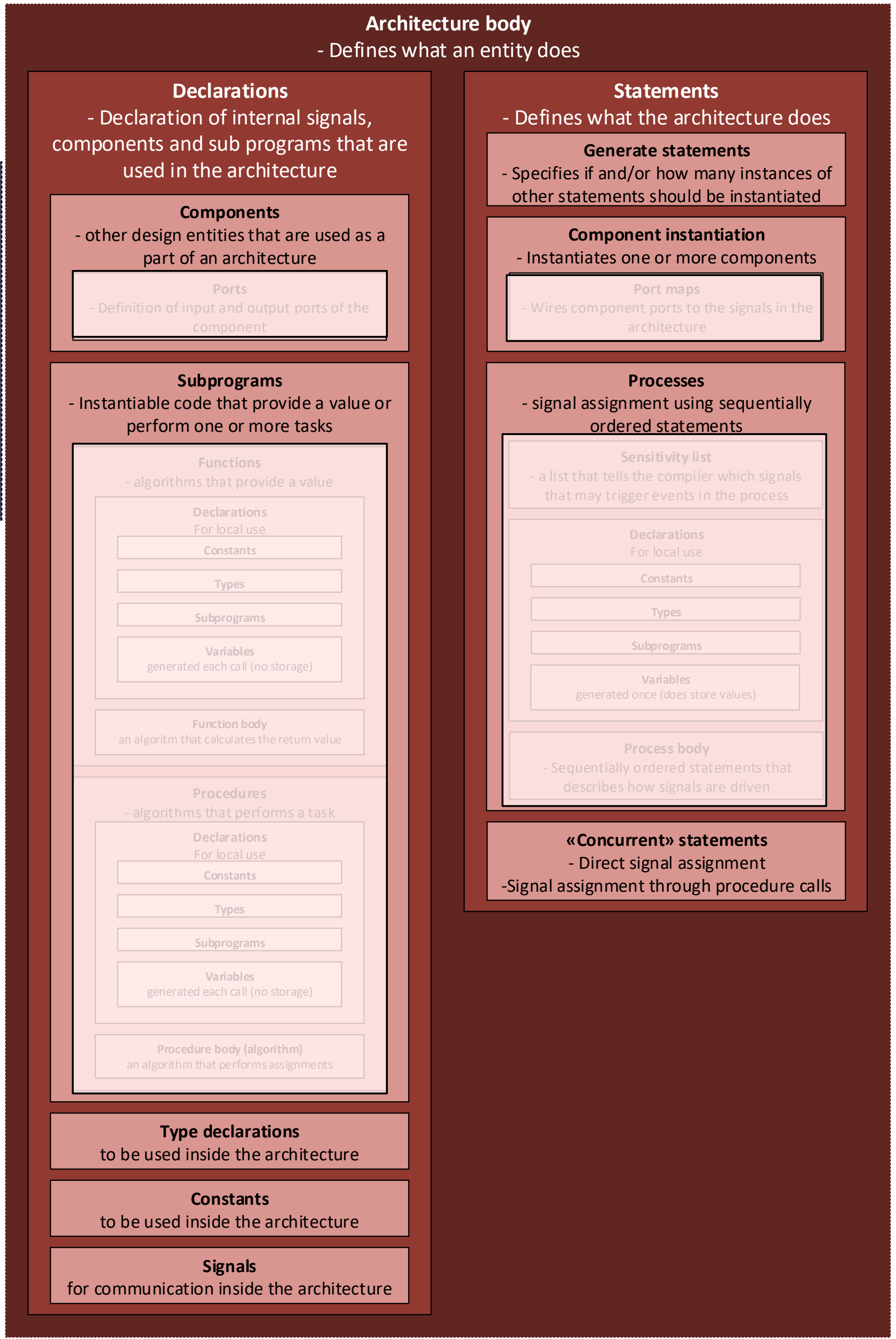
- Design entities
- Architecture styles
- Ports and signals
- Vectors
- Assignment
- Libraries
- STD_LOGIC data type
- Operators

Design entities in VHDL

- 5 types of design entities
  - Entity
  - Architecture *body*
  - Package
  - Package body
  - Configuration
- Each entity can have its own file...

Design-entities:
Entity and architecture

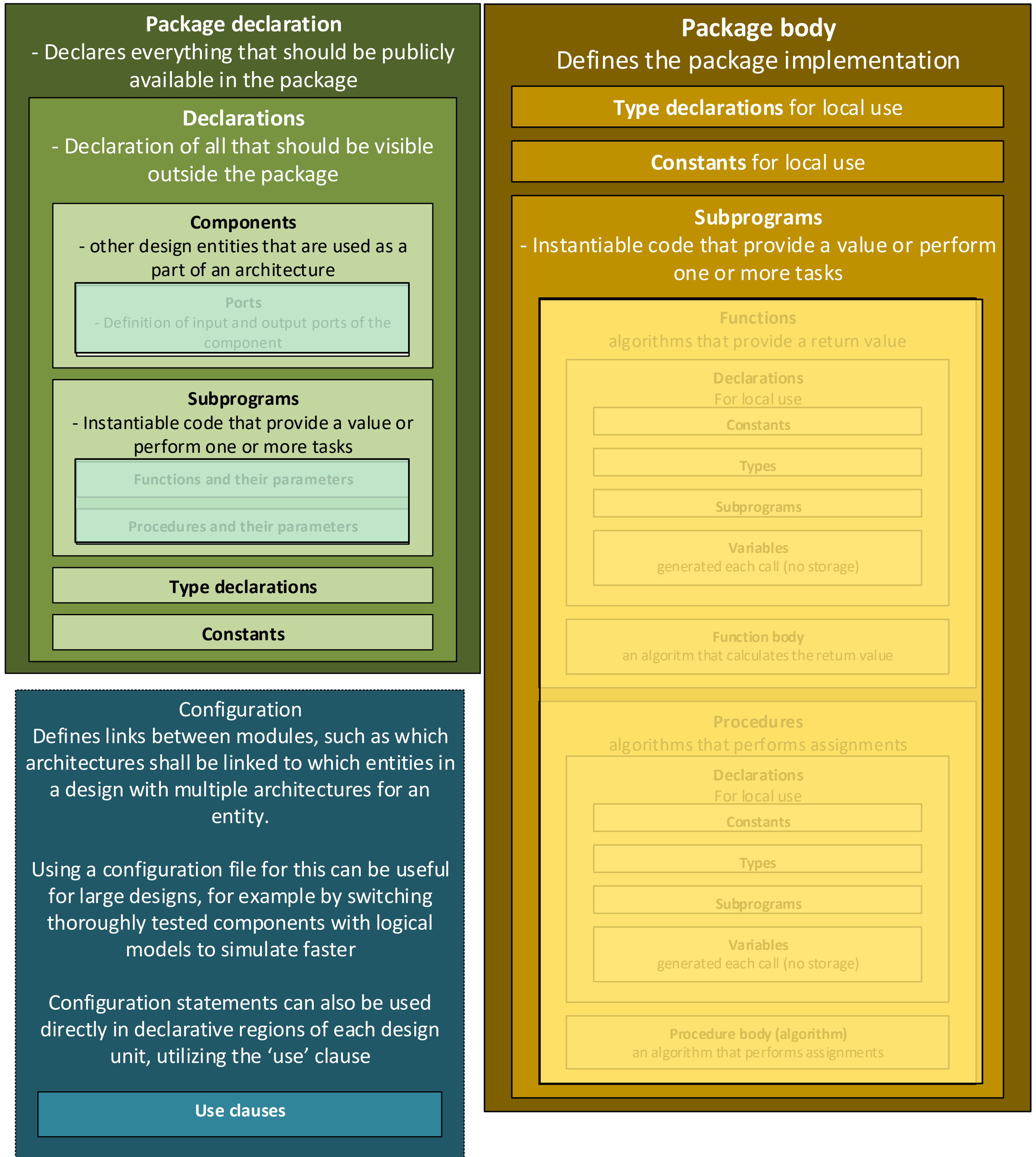

Entity
- Defines the interface to the surrounding environment
Generics
-Values that can be used inside the entity as constants
IO ports
- Defines the entity interface


Architecture body
- Defines what an entity does
Declarations
- Declaration of internal signals, components and sub programs that are used in the architecture
Components
- other design entities that are used as a part of an architecture
Ports
- Definition of input and output ports of the component
Subprograms
- Instantiable code that provide a value or perform one or more tasks
Functions
- algorithms that provide a value
Declarations
For local use
Constants
Types
Subprograms
Variables
generated each call (no storage)
Function body
an algoritm that calculates the return value
Procedures
- algorithms that performs a task
Declarations
For local use
Constants
Types
Subprograms
Variables
generated each call (no storage)
Procedure body (algorithm)
an algorithm that performs assignments
Type declarations
to be used inside the architecture
Constants
to be used inside the architecture
Signals
for communication inside the architecture
Statements
- Defines what the architecture does
Generate statements
- Specifies if and/or how many instances of other statements should be instantiated
Component instantiation
- Instantiates one or more components
Port maps
- Wires component ports to the signals in the architecture
Processes
- signal assignment using sequentially ordered statements
Sensitivity list
- a list that tells the compiler which signals that may trigger events in the process
Declarations
For local use
Constants
Types
Subprograms
Variables
generated once (does store values)
Process body
- Sequentially ordered statements that describes how signals are driven
«Concurrent» statements
- Direct signal assignment
-Signal assignment through procedure calls

- E. and A. is often put into the same file
- In larger designs these may be separated, several architectures can be used for one entity.
  - Simulation vs modules for synthesis
- *Details will be revealed later..*

Design entities:
Package, P.body, Configuration

- VHDL uses and can be used to create packages
- We will almost always use packages in precompiled libraries.
- Configuration files can be used to specify which components or architectures that shall be used in (large) designs
  - (Not a primary concern for in3160)


Package declaration
- Declares everything that should be publicly available in the package

Declarations
- Declaration of all that should be visible outside the package

Components
- other design entities that are used as a part of an architecture

Ports
- Definition of input and output ports of the component

Subprograms
- Instantiable code that provide a value or perform one or more tasks

Functions and their parameters

Procedures and their parameters

Type declarations

Constants

Package body
Defines the package implementation

Type declarations for local use

Constants for local use

Subprograms
- Instantiable code that provide a value or perform one or more tasks

Functions
algorithms that provide a return value

Declarations
For local use

Constants

Types

Subprograms

Variables
generated each call (no storage)

Function body
an algoritm that calculates the return value

Procedures
algorithms that performs assignments

Declarations
For local use

Constants

Types

Subprograms

Variables
generated each call (no storage)

Procedure body (algorithm)
an algorithm that performs assignments

Configuration
Defines links between modules, such as which architectures shall be linked to which entities in a design with multiple architectures for an entity.

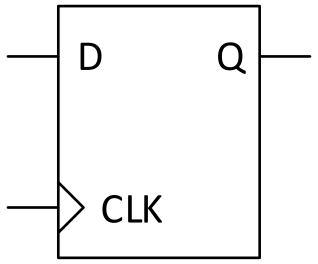
Using a configuration file for this can be useful for large designs, for example by switching thoroughly tested components with logical models to simulate faster

Configuration statements can also be used directly in declarative regions of each design unit, utilizing the 'use' clause

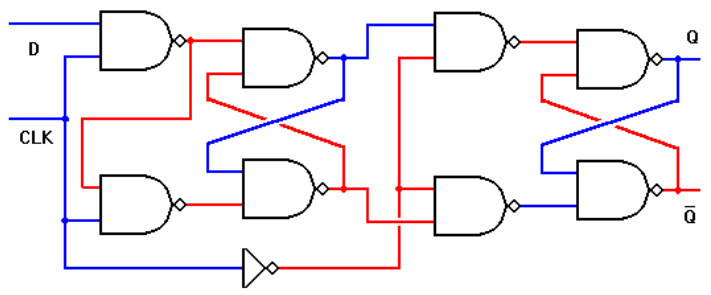
Use clauses

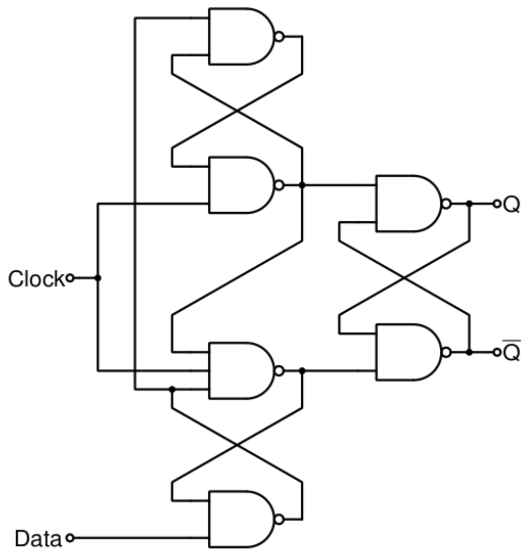
VHDL Entity and Architecture

- Entity defines Input and Output ports in the design
  - There is only one entity in a vhdl file..
- Architecture defines what the design does.
  - There can be several architectures for an entity
  - Architectures, may be defined using different styles (next slide)
    - “RTL” and “Dataflow” are names providing information;
      - changing these names would not change function.



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

entity D_flipflop is
  port(
    clk: in std_logic;
    D  : in std_logic;
    Q  : out std_logic
  );
end entity D_FLIPFLOP;


architecture RTL of D_flipflop is
begin
  process (clk) is
  begin
    if rising_edge(clk) then
      Q <= 'D';
    end if;
  end process;
end architecture RTL;


architecture data_flow of D_flipflop is
  signal e, f, g, h, i, j, k, l: std_logic;
begin
  -- concurrent statements
  e <= NOT (D AND clk);
  f <= NOT (e AND clk);
  g <= NOT (e AND h);
  h <= NOT (f AND g);
  i <= NOT (g AND NOT clk);
  j <= NOT (h AND NOT clk);
  k <= NOT (l AND i);
  l <= NOT (k AND j);
  Q <= k;

end architecture data_flow;
```





UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

4 main abstraction levels

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

entity D_flipflop is
  port(
    clk: in std_logic;
    D: in std_logic;
    Q: out std_logic
  );
end entity D_flipflop;
```

```vhdl
architecture RTL of D_flipflop is
begin
  process (clk) is
  begin
    if rising_edge(clk) then
      Q <= 'D';
    end if;
  end process;
end architecture RTL;
```

RTL, register transfer level

- high level
- easy to read
- describes registers and what happens between them
- «default» for sequential logic

Data Flow

- typically used for optimization
- will easily become unreadable if used extensively.
- *Use higher level code unless absolutely necessary*

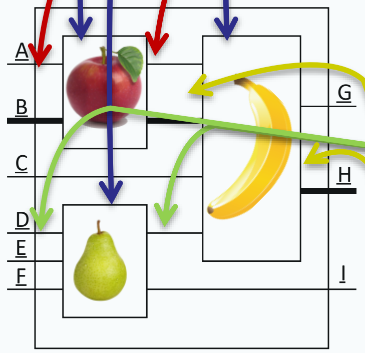
```vhdl
architecture data_flow of D_flipflop is
  signal e, f, g, h, i, j, k, l: std_logic;
begin
  -- concurrent statements
  e <= not (D and clk);
  f <= not (e and clk);
  g <= not (e and h);
  h <= not (f and g);
  i <= not (g and not clk);
  j <= not (h and not clk);
  k <= not (l and i);
  l <= not (k and j);
  Q <= k;
end architecture data_flow;
```

Structural

- Ties components together
- Typically used in test benches, and when using predefined components

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

entity my_thing is
  port (
    A: in std_logic;
    B: in std_logic_vector(5 downto 0);
    C, D, E, F: in std_logic;
    G: out std_logic;
    H: out std_logic_vector(64 downto 0);
    I: out std_logic
  );
end entity my_thing;
```

```vhdl
architecture structural of my_thing is
  signal js: std_logic;
  signal ks: std_logic_vector(64 downto 0);
  signal ls: std_logic;
 component apple is
  port (
    A: in std_logic;
    B: in std_logic_vector(5 downto 0);
    C: out std_logic;
    D: out std_logic_vector(64 downto 0)
  );
 end component;

 component pear is
  port (
    A, B, C: in std_logic;
    D, E: out std_logic
  );
 end component;

component banana is
  port (
    smurf: in std_logic_vector(64 downto 0);
    cat, dog, donkey: in std_logic;
    horse: out std_logic;
    monkey: out std_logic_vector(64 downto 0)
  );
 end component;

begin -- port map (component => My_thing)
 U1: apple port map(
   A => A,
   B => B,
   C => js,
   D => ks
 );

 U2: pear port map(
   A => D,
   B => E,
   C => F,
   D => ls,
   E => I
 );

 U3: banana port map(
   smurf  => ks,
   cat    => js,
   dog    => C,
   donkey => ls,
   horse  => G,
   monkey => H
 );

end architecture structural;
```

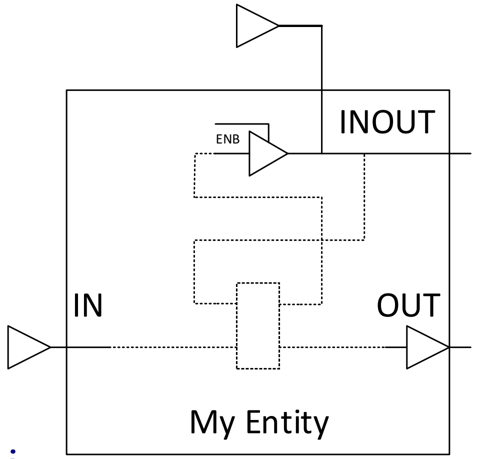


Behavioral

- *Simulation models only = Software*
- Not for synthesis or implementation

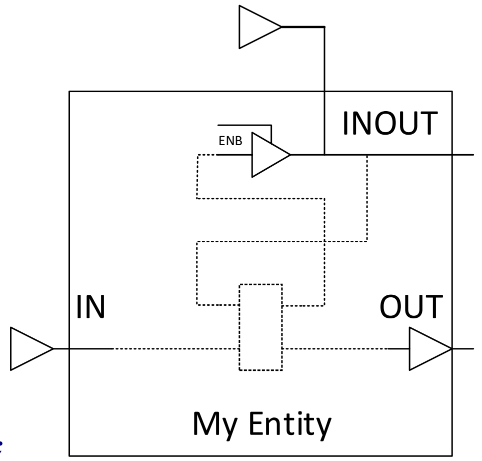
Ports and signals

- Ports define the entity interface
  - IN:
    - *can only be read*
    - cannot be driven or assigned internally
  - OUT:
    - *can only be driven*
      - *(In VHDL 2008, output ports can be read as an internal signal).*
    - Should be assigned in the architecture
      - *It is bad practice not to do so.*
  - *INOUT*
    - *Can be both driven and read (typical use is for buses)*
- Signals are internal
  - For connecting internal modules, subprograms and processes.



```
library IEEE;
  use IEEE.std_logic_1164.all;

entity my_entity is
  port(
    Ain    : in    std_logic;
    Binout : inout std_logic;
    Cout   : out   std_logic
  );
end entity my_entity;

architecture dataflow of my_entity is
  signal enb, s1, s2, s3, s4 : std_logic;

begin
  -- ...

  s1 <= Ain;
  Cout <= s2;
  -- reading INOUT is OK
  s3 <= Binout;
  -- setting INOUT should implement tri-state
  Binout <= s4 when enb else 'Z';

end architecture dataflow;
```

Ports continued



INOUT is for tying input and output to the same pin

- ***should*** **implement tristate functionality**.
- 'Z' means it is not driven (tristate)
- Typically to be used when connecting a bus with multiple drivers.

DO NOT use INOUT for convenience!

- The compiler will not alert you if you are driving from two sources simultaneously.
  - *May cause electrical faults*
- INOUT may infer inferior structures (long delays)

use an extra signal *unless you need to assign both input and output to the same physical pin (location)***.**

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

entity my_entity is
  port(
    Ain    : in     std_logic;
    Binout : inout  std_logic;
    Cout   : out    std_logic
  );
end entity my_entity;

architecture dataflow of my_entity is
  signal enb, s1, s2, s3, s4 : std_logic;

begin
  -- ...

  s1 <= Ain;
  Cout <= s2;
  -- reading INOUT is OK
  s3 <= Binout;
  -- setting INOUT should implement tri-state
  Binout <= s4 when enb else 'Z';

end architecture dataflow;
```

Statements and assignments

- Statements and processes
  - defines how a digital circuit works
  - assign drivers to signals
  - All statements must be valid at all times
    - A circuit may have sequential behavior...
      - But all logic is present and functional at all times
  - Processes are complex statements
    - Assigns values to one or more signals
    - Can have local variables.. (more on those later)
  - A signal can only be assigned once in an architecture
    - We do not want two circuits to apply voltage on the same wire...

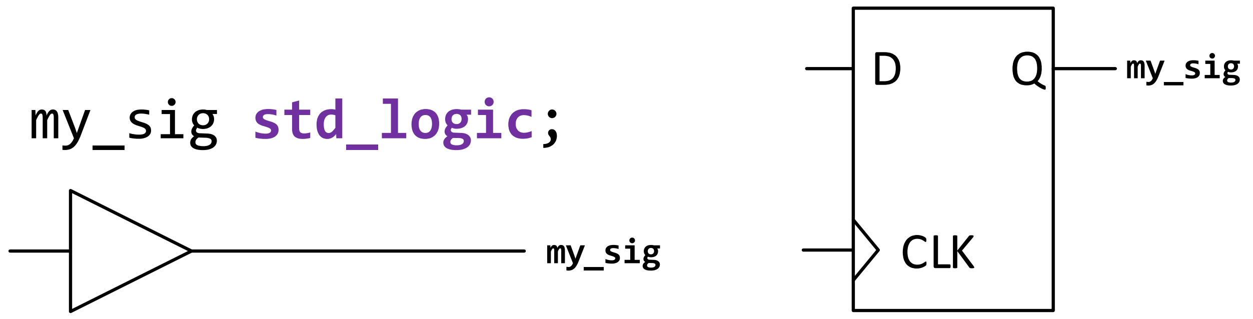
Examples:

```
with input select isprime <=
    '1' when x"1" | x"2" | x"3" | x"5" | x"7" | d"11" | x"d",
    '0' when others;
```

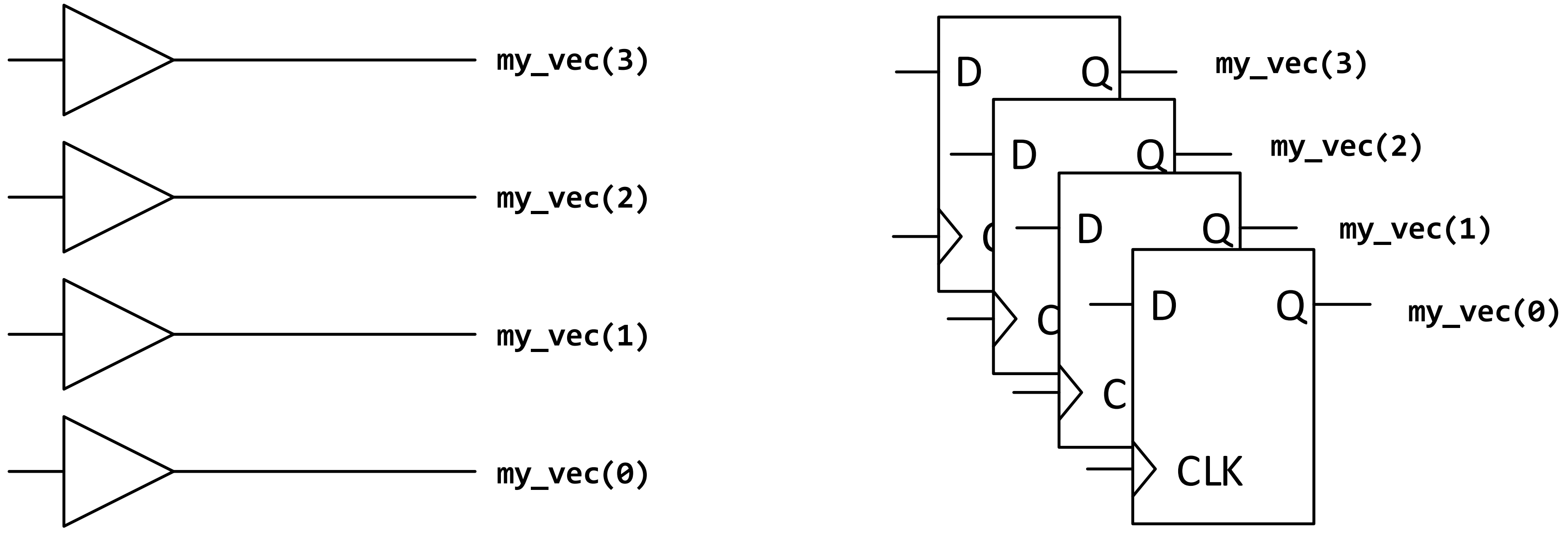
```
process(Reset, Clock) is
    variable Q : Unsigned(7 downto 0);
  begin
    if Reset = '0' then
      Count <= zero_byte;
    elsif rising_edge(Clock) then
      Q :=
        unsigned(Count)     when  Enable  else
        unsigned(Data)      when not Load else
        unsigned(Count) + 1 when not Mode else
        dec_count(unsigned(Count));
      Count <= std_logic_vector(Q);
    end if;
  end process;
```

«Vectors»

- `signal my_sig std_logic;`



- `signal my_vec std_logic_vector(3 downto 0);`



Signals and variables

- Signals are for inter-architecture communication
  - Between processes, modules and subprograms
- Variables are subprogram(or process)-internal
  - To make code clearer, and more local.
- Example note:
  - placement of s&v declarations
  - Signal assignment order is irrelevant outside processes

```
architecture example of sigvar is
  -- (signal) declarations
  signal S, T : std_logic;

begin
  -- statements
  S <= A and B;

  process (A,B) is
    -- (variable) declarations
    variable V : std_logic;
  begin
    -- process body
    V := '0';
    -- …

  end process;

  X <= S XOR T;
end architecture;
```

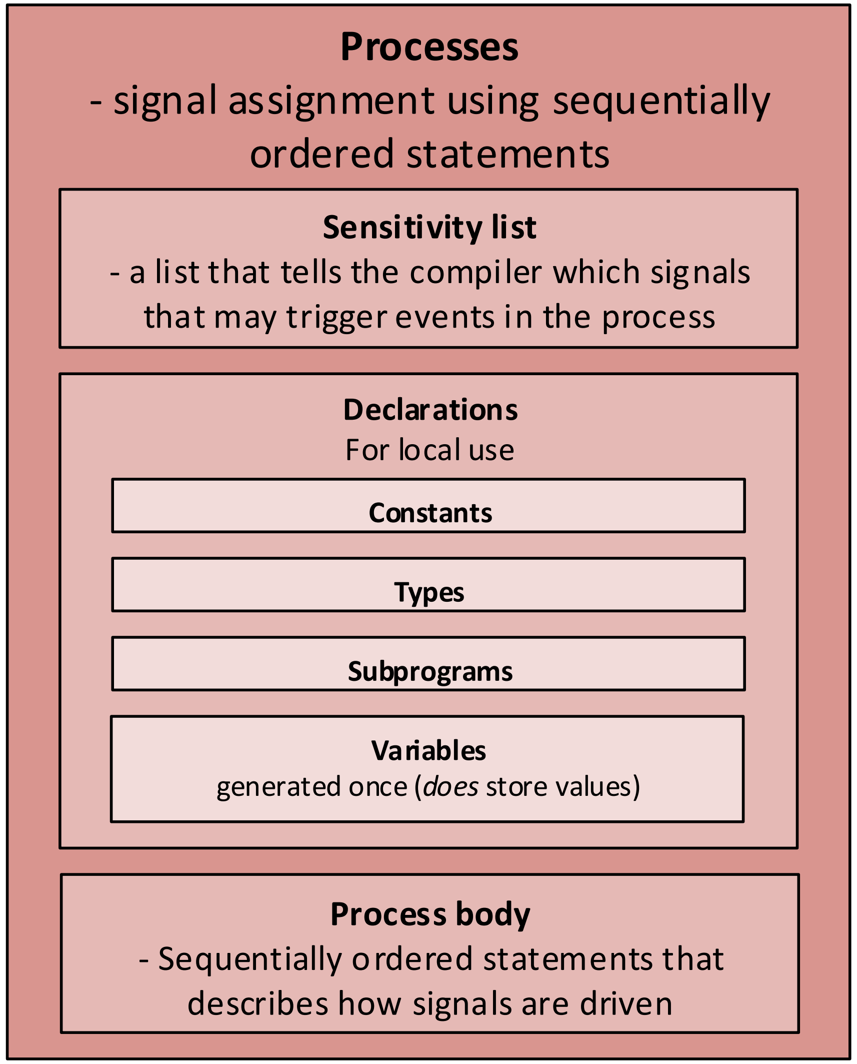
Signal and variable assignment

- Signals are assigned concurrently in statements
  - both in and outside processes
  - Signals are assigned using <=
  - Signals uses event based updates
    - ie after a process is complete.
- Variables can only be used inside processes and subprograms
  - Variables are assigned using :=
  - Variables are updated immediately in simulation
  - *Processes can have variables store values*
    - *Initialized at the beginning of simulation*
  - *Subprograms (procedure, function) can not have variables store values*
    - *initialized on every call*

```
A <= B; -- A reads B, or A is assigned to B’s ouput,
        -- (A is a signal)

C := B; -- C is given B’s value, C is a variable
        -- variables are used internally in processes.


D(6 downto 0) <= E(3 downto 1) & (others => ‘0’);
            -- D is a vector having 7 input signals
            -- D(6) <= E(3)
            -- D(5) <= E(2)
            -- D(4) <= E(1)
            -- D(3 downto 0) <= "0000"
```

Processes

- A process is one (concurrent) statement
  - Ensures one driver for each signal by using priority.
    - "Signals are only updated only once"..
  - The process body has sequential *priority*
    - Last assignment takes precedence over previous.
    - Variables *can* be assigned multiple times within a process body(!)
      - They *can* act as several signals
        - » Generally this should be avoided
  - sensitivity list
    - determines when the process body is invoked *during simulation*
    - *Event triggered*
  - *Can* be used to make sequential logic
    - Clocked events infers flipflops (or latches)


Processes
- signal assignment using sequentially ordered statements

Sensitivity list
- a list that tells the compiler which signals that may trigger events in the process

Declarations
For local use

Constants

Types

Subprograms

Variables
generated once (*does* store values)

Process body
- Sequentially ordered statements that describes how signals are driven

Process example
(this example ≠ good code)

```
library IEEE;
  use IEEE.std_logic_1164.all;

entity sigvar is
  port(
    A, B : in  std_logic;
    X    : out std_logic
  );
end entity sigvar;
```

- The use of V would not be allowed a signal outside a process.
- A variable can be used in place of multiple physical signals within a process. (= bad practice...)
- Signals will be assigned one driver in a process.

```
architecture example of sigvar is
  -- declarations
  signal S, T : std_logic;

begin
  -- statements
  S <= A and B;

  process (A,B) is
    -- decalarations
     variable V : std_logic;

  begin
    -- process body
    V := '0';
    if (A = '1') then
      V := '1';
    end if;
    if (B = '1') then
      V:= '1';
    end if;
    T <= V;
  end process;

  X <= S XOR T;
end architecture;
```

Libraries and Data types

- VHDL is built upon use of libraries and packages.

- You can both use existing ones, and create your own.

- Most used is the IEEE library, which contains the most used data types and functions
  - The built-in standard (std) package, containing:
    - bit, integer, natural, positive, boolean, string, character, real, time, delay_length
  - std_logic_1164 (which defines the STD_LOGIC type)
  - numeric_std (numeric operations for std_logic_vectors)
  - std_logic_textio (to provide IO during simulation)
  - numeric_bit (numeric operations for bit vectors)
  - etc.

STD_LOGIC *TYPE* (requires std_logic_1164 package from IEEE library)

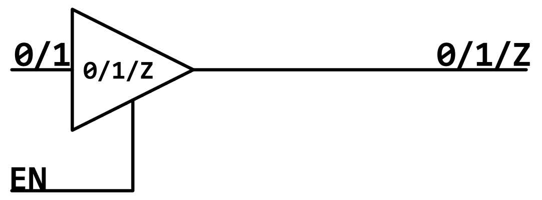
- STD_LOGIC is a **type** that has the following possible values

– ‘U’	Uninitialized (Typically seen in simulation before initializing values)
– ‘X’	Unknown (typically when a signal is driven to both 0 and 1 simultaneously)
– **‘0’**	**Driven low**
– **‘1’**	**Driven High**
– **‘Z’**	**Tristate**
– ‘W’	Weak unknown (when driven by two different weak drivers)
– ‘L’	Weak ‘0’ (Typically for simulating a pulldown resistor)
– ‘H’	Weak ‘1’ (Typically for simulating a pullup resistor)
– ‘-’	Don’t care (Typically for assessing results in simulator).

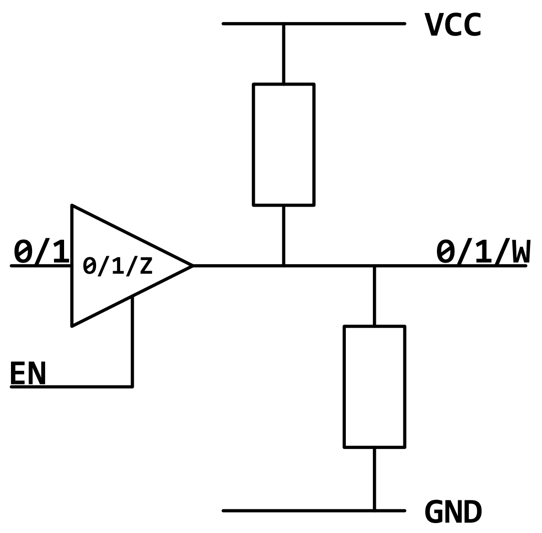
- **You will only *assign*** synthesizable **signals to ‘0’, 1’ and ‘Z’**

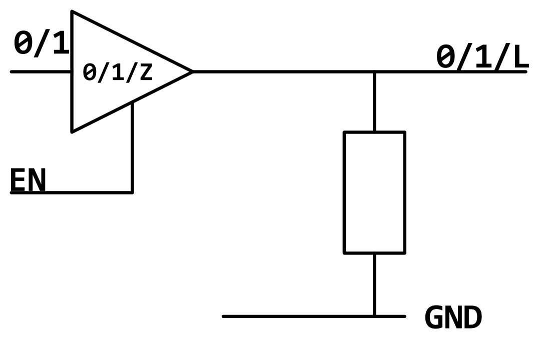
- Type STD_LOGIC_VECTOR is array (NATURAL range <>) og STD_LOGIC
  - STD logic vector is used for hardware. For simulation, other types (such as integer) may be faster. Thus we use STD_LOGIC for hardware interactions, and other types when possible for test bench code.

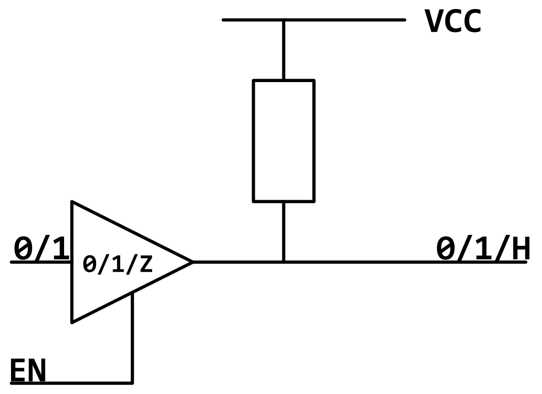
std_logic -- values



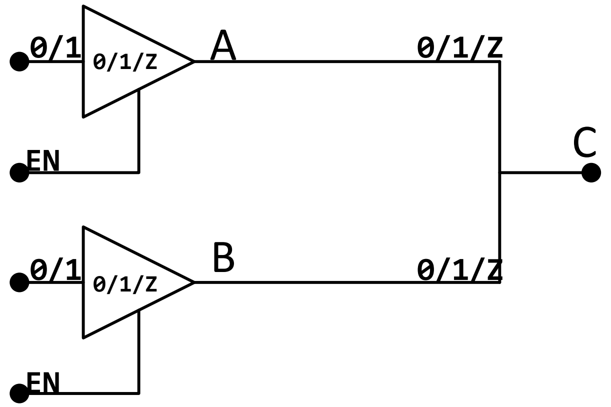
Value	Name	Usage
‘U’	Uninitialized state	Used as a default value
'X'	Forcing unknown	Bus contentions, error conditions, etc.
‘0’	Forcing zero	Transistor driven to GND
‘1’	Forcing one	Transistor driven to VCC
‘Z’	High impedance	3-state buffer outputs
‘W’	Weak unknown	Bus terminators
‘L’	Weak zero	Pull down resistors
‘H’	Weak one	Pull up resistors
‘-’	Don’t care	Used for synthesis and advanced modeling





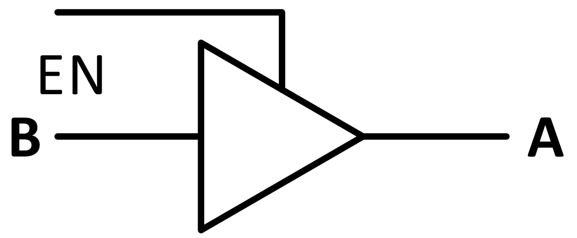


```
-- multiple drivers
signal c : std_logic;
```



Signal A/B	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'–'
'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'
'X'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'0'	'U'	'X'	'0'	'X'	'0'	'0'	'0'	'0'	'X'
'1'	'U'	'X'	'X'	'1'	'1'	'1'	'1'	'1'	'X'
'Z'	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'X'
'W'	'U'	'X'	'0'	'1'	'W'	'W'	'W'	'W'	'X'
'L'	'U'	'X'	'0'	'1'	'L'	'W'	'L'	'W'	'X'
'H'	'U'	'X'	'0'	'1'	'H'	'W'	'W'	'H'	'X'
'–'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'

Tri-state buffer



- Hardware can only read ‘0’ or ‘1’
- We can set tristate (high impedance) to allow other sources to drive a bus.
- Simulation tools can use all possible STD_LOGIC values.

```
A <= B when EN = '1' else 'Z';

-- ekvivalent med
TRISTATE:
process (B,EN)
begin
  if EN = '1' then
    A <= B;
  else
    A <= 'Z';
  end if;
end process;
```

VHDL operator priority

- Functions are interpreted from left to right (in reading order).
- **Use paranthesis to govern priority**!

Prioritet	Operator klasse	Operatorer
1 (first)	miscellaneous	`**, abs, not`
2	multiplying	`*, /, mod, rem`
3	sign	`+, -`
4	adding	`+, -, &`
5	Shift	`sll, srl, sla, sra, rol, ror`
6	relational	=, /=, <, <=, >, >=, ?=, ?/=, ?<, ?<=, ?>, ?>=
7	logical	`And, or, nand, nor, xor, xnor`
8 (last)	condition	`??`

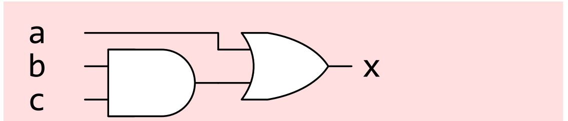
Examples (elaboration on next page):

```
a <= a or b and c;   ==   a <= (a or b) and c;

z <= a and not b and c; == z <= a and (not b) and c; == z <= c and (a and (not b));

y <= a and not (b and c); -- z=1 kun for a=1, b=0, c=1.   y=1 for a=1 og (b eller c)=0.
```

VHDL operator priority

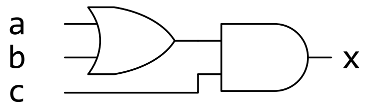
Examples:

```
x <= a or b and c;  ==  x <= (a or b) and c;
```



```
x <= a or (b and c);
```

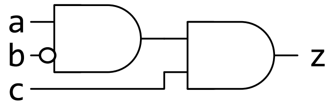
(What you might want)



```
x <= a or b and c;
```

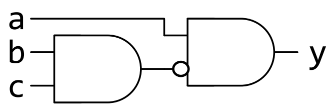
(what you actually will get)

Try
a := ‘1’
b := ‘1’
c := ‘0’



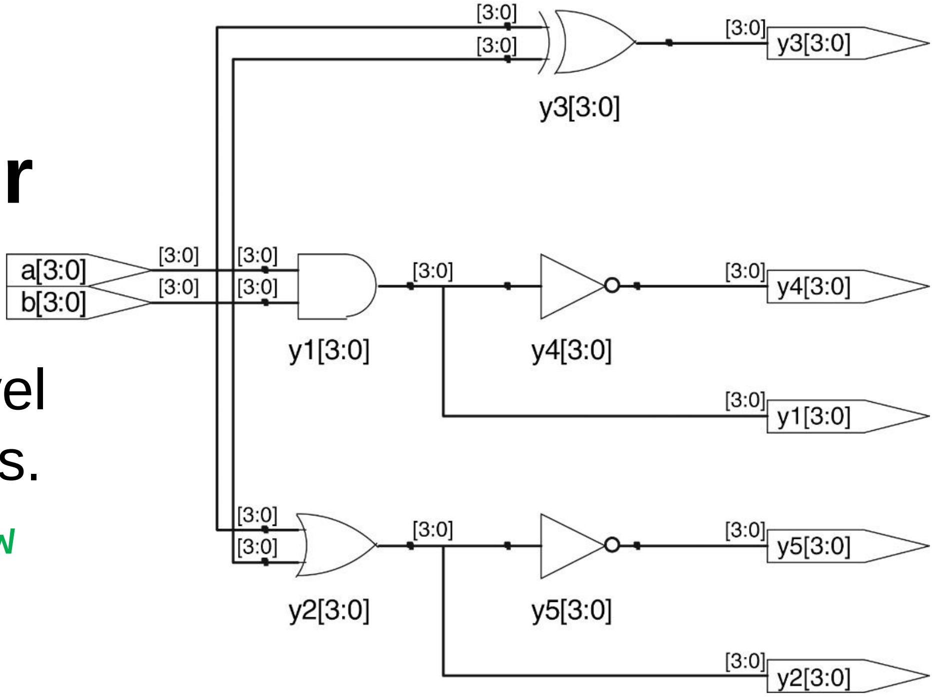
```
-- z=1 only when a=1, b=0, c=1.
```

```
z <= a and not b and c; == z <= a and (not b) and c; == z <= c and (a and (not b));
y <= a and not (b and c);
```



```
--  y=1 when a=1, b=0 or when a=1, c=0.
```

Bit operators and reduction operator

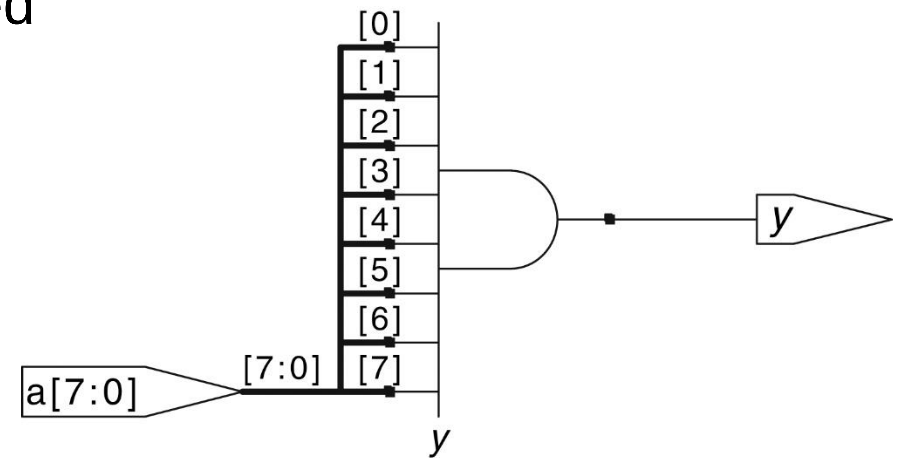


- **and**, **or**, **not**, **xor**, **xnor** operators will work at bit level when they are placed between two signals or vectors.
  - y1 <= a and b; -- is equal to the lines below

```
y1(3) <= a(3) and b(3);
y1(2) <= a(2) and b(2);
y1(1) <= a(1) and b(1);
y1(0) <= a(0) and b(0);
```

- I VHDL2008 (not earlier) these operators can be used for reduction

```
y <= and a ; -- is equal to the figure ->
```



- **xor** can be used this way to generate (even) parity for a signal.

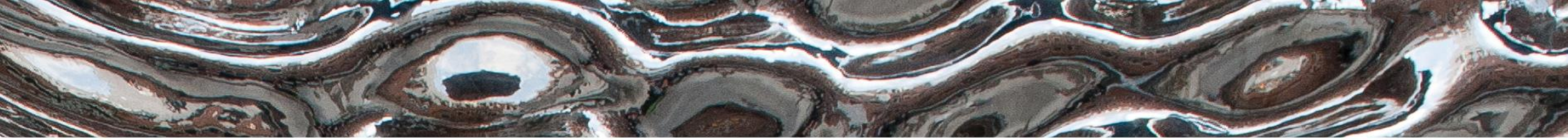
VHDL Bit String Literals

Binary, **D**ecimal, he**X**adecimal, **O**ctal, **U**nsigned, **S**igned

<ant bit><U/S><B/D/O/X><numbers of type B/D/O/X >

```
B"1111_1111_1111" -- Equivalent to the string literal
"111111111111".
X"FFF" -- Equivalent to B"1111_1111_1111".
O"777" -- Equivalent to B"111_111_111".
X"777" -- Equivalent to B"0111_0111_0111".
B"XXXX_01LH" -- Equivalent to the string literal
"XXXX01LH"
UO"27" -- Equivalent to B"010_111"
UO"2X" -- Equivalent to B"011_XXX"
SX"3W" -- Equivalent to B"0011_WWWW"
D"35" -- Equivalent to B"100011"
```

```
12UB"X1" -- Equivalent to B"0000_0000_00X1"
12SB"X1" -- Equivalent to B"XXXX_XXXX_XXX1"
12UX"F-" -- Equivalent to B"0000_1111_----"
12SX"F-" -- Equivalent to B"1111_1111_----"
12D"13" -- Equivalent to B"0000_0000_1101"
12UX"000WWW" -- Equivalent to B"WWWW_WWWW_WWWW"
12SX"FFFC00" -- Equivalent to B"1100_0000_0000"
12SX"XXXX00" -- Equivalent to B"XXXX_0000_0000"
8D"511" – Error (> 2^8)
8UO"477" – Error (>2^8)
8SX"0FF" – Error (cannot have 255 using 8 bit signed)
8SX"FXX" – Error (cannot extend beyond 8 bit)
```



UiO : **Institutt for informatikk**

Det matematisk-naturvitenskapelige fakultet

IN3160

Code Layout (15 min)

Kilde: Ricardo Jasinski: Effective Coding with VHDL, Chapter 18

Overview

- Why bother thinking about layout?
- What constitutes a good layout scheme?
- Basic layout types
- Indentation
- Paragraphs and spaces

Why bother thinking of layout?

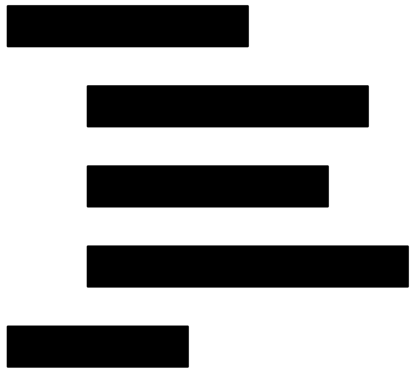
```
pRoCeSS(clock,reset) bEGIn iF resET then oUTpuT <="0000"; elSE IF RISING_edge
(ClOck) tHEN cASE s Is When 1=>outPUT<= "0001"; wHEn 2046=> oUTpuT <="0010";WheN
31=>OutPut<="0100";when OTHERs=>OUTput <= «1111"; end CASe; END if;END proCESS; --
Q.E.D.
```

A good layout scheme…

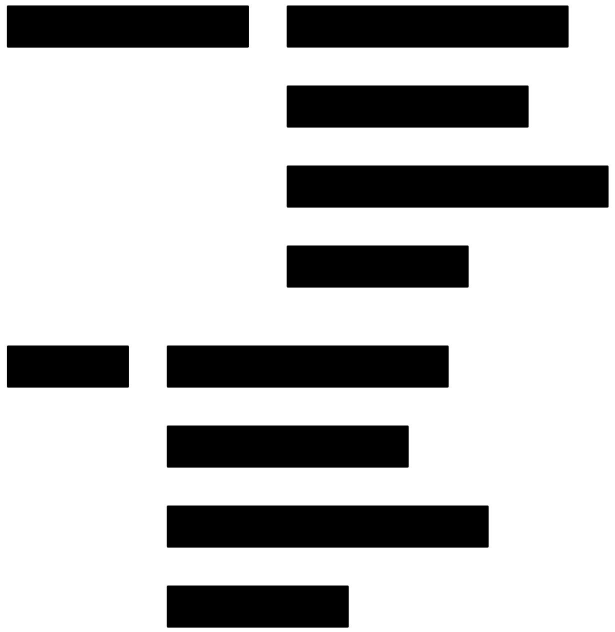
1. …accurately matches the structure of the code
2. …improves readability
3. …affords changes
4. …is consistent (few exceptions)
5. …is simple (few rules)
6. …is easy to use
7. …is economic

Basic layout types

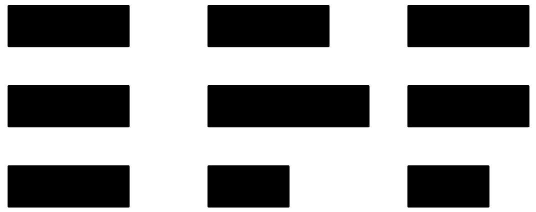
Block layout



Endline layout



Column layout



Block layout (What you should use *most of the time*)

- Accurately matches structure
- relatively tidy
- readable,
- easy to maintain, etc.

```
process (clock, reset)
begin
  if reset then
    output <= "0000";
  else if rising_edge(clock) then
    case s is
      when 1 => output <= "0001";
      when 2046 => output <= "0010";
      when 31 => output <= "0100;
      when others => output <= "1111";
    end case;
  end if;
end process;
```

Endline layout (*avoid this*)

- Harder to maintain when code changes.
- Looks tidier, but isn't faster than pure block
- Will get messy- poor match of code hierarchy
- Long lines..!

```
process (clock, reset)
begin
  if reset then output <= "0000";
  else if rising_edge(clock) then case s is
                                  when 1 => output <= "0001";
                                  when 2046 => output <= "0010";
                                  when 31 => output <= "0100";
                                  when others => output <= "1111";
                                  end case;
  end if;
end process;
```

Column layout (use sparingly)

- Can be easier to read than pure block layout (scanning vertically)
- Harder to maintain.
- Best to use when columns are unlikely to change, and statements are related.
  - Typically used for 2D arrays.

```
process (clock, reset)
begin
  if reset then
    output <= "0000";
  else if rising_edge(clock) then
    case s is
      when      1 => output <= "0001";
      when      2 => output <= "0010";
      when    333 => output <= "0100";
      when others => output <= "1111";
    end case;
  end if;
end process;
```

Indentation

- *Use indentation to match code hierarchy*
- 2-4 spaces has been proven most efficient
  - *Along with a monospace font, such as* `courier`, `consolas..`
- Use space rather than tabulator sign.
  - Tabulator spaces may be interpreted differently in different editors.
  - Most editors can be set up for this.
- Example:

```
entity ent_name is
  generic (
    generic_declaration_1;
    generic_declaration_2;
  );
  port(
    port_declaration_1;
    port_declaration_2;
  );
end entity ent_name;
.
.
.
process (sensitivity_list)
  declaration_1;
  declaration_2;
begin
  statement_1;
  statement_2;
end process;
```

Paragraphs and comments

- Paragraphs should be used to separate chunks that does not need to be read all at once.

- Paired with comments that this make for good readability

- Comments should be indented as according to the code it is referring to.

```
-- Find character in text RAM corresponding to x, y
text_ram_x := pixel_x / FONT_WIDTH;
text_ram_y := pixel_y / FONT_HEIGHT;
display_char := text_ram(text_ram_x, text_ram_y);

-- Get character bitmap from ROM
ascii_code := character’pos(display_char);
char_bitmap := FONT_ROM(ascii_code);

-- Get pixel value from character bitmap
x_offset := pixel_x mod FONT_WIDTH;
y_offset := pixel_y mod FONT_HEIGHT;
pixel := char_bitmap(x_offset)(y_offset);
```

Line length and wrapping

- Try to keep line length within reasonable limits
  - 80, 100 and 120 characters is widely used.

- When wrapping lines:
  - break at a point that clearly shows it is incomplete, such as
    - after opening paranthesis
    - after operators or commas (`&, +, -, *, /`)
    - after keywords such as «`and`» or «`or`»
  - consider one item per line…

```
-- one item/line + named association
Paddle <= update_sprite(
  sprite => paddle,
  sprite_x => paddle_position.x + paddle_increment.x,
  sprite_y => paddle_position.x + paddle_increment.y,
  raster_x => vga_raster_x,
  raster_y => vga_raster_y,
  sprite_enabled => true
);
```

```
-- several items/line
paddle <= update_sprite(paddle, paddle_position.x + paddle_increment.y,
  paddle_position.y + paddle_increment.y, vga_raster_x, vga_raster_y, true);
```

Spaces

- Punctuation symbols (comma, colon, semicolon)
  - use spaces as you would in regular prose:
    - Never add space before punctuation symbols
    - Always add space after punctuation symbols
  - *no exceptions*

```
-- too much
function add ( addend : signed ; augend : signed ) return signed ;

-- better
function add(addend: signed; augend: signed) return signed;
```

- Parantheses
  - Add a space before opening paranthesis.
  - Add a space or punctuation symbol after closing paranthesis
  - Except:
    - array indices and routine parameters;
    - *expressions*.

```
-- consider this expression:
a + b mod c sll d;

-- better
(a + (b mod c)) sll d;
```

```
-- consider
(-b+sqrt(b**2-4*a*c))/2*a;

-- better
(-b + sqrt(b**2 – 4*a*c)) / 2*a;

-- too much
( - b + sqrt( b ** 2 – 4 * a * c ) ) / 2 * a;
```

Naming conventions - Letter case and underscores

```
noconventionnaming -- don’t go there
UpperCamelCase -- OK used consequently
lowerCamelCase -- OK used consequently
snake_case or underscore_case
SCREAMING_SNAKE_CASE or ALL_CAPS
```

- Do not use ALL_CAPS too frequently.
- Use editor colors/ **bold** for higlighting **keywords**
- Try to avoid mixing snakes_andCamels.
- Treat acronyms/ abbreviations as words
  - "UDPHDRFromIPPacket" ***vs*** "UdpHdrFromIpPacket" ***vs*** "udp_hdr_from_ip_packet"
- VHDL packages tend to favour snake_case and ALL_CAPS
- *Suggestion:*
  - Use snake_case for all **names** except **constants** and **generics** that use ALL_CAPS

Suggested reading, Mandatory assignments

- D&H:
  - 1.5 p 13-16
  - 3.6 p 51-54
  - 6.1 p 105-106

(Layout is lecture only)

- Oblig 1: «Design Flow»
- Oblig 2: «VHDL»
  - See canvas for further instruction.