

Lectures 2-4: Introduction to System design, VHDL Basics

TIE-50206 Logic synthesis

Erno Salminen

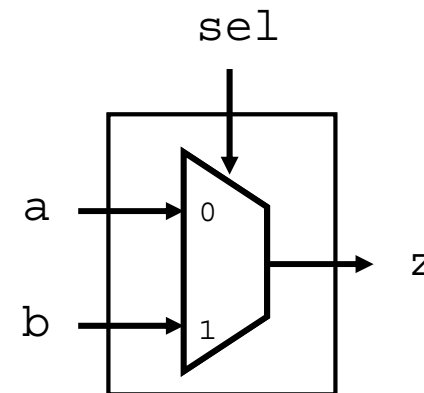
Tampere university of technology

Fall 2014

```
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 PROCESS;
END syn;
```

synthesize

model



Contents

■ 1 Introduction to System design

- Abstraction
- Main phases

■ 2 VHDL basics

- Entity – the interface
 - Ports, generics
- Architecture – the behavior
 - Signals, types
 - Process, component instantiation, control statements
- Library, package

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 made by Ari Kulmala
 - and other previous lecturers (Teemu Pitkänen, Konsta Punkka, Mikko Alho...)



1. Introduction to system design

1a. Representation (View) and abstraction

Examples of different views

■ View: different perspectives of a system

1. Behavioral view:

- Describe functionalities and i/o behavior
- Treat the system as a black box

2. Structural view:

- Describe the internal implementation (components and interconnections)
- Essentially a block diagram (or schematic)

3. Physical view:

- Add more info to structural view: component size, component locations, routing wires
- E.g. layout of a print circuit board

Examples of different views (2)

inputs:

button0_in, button1_in

...

outputs:

led0_out

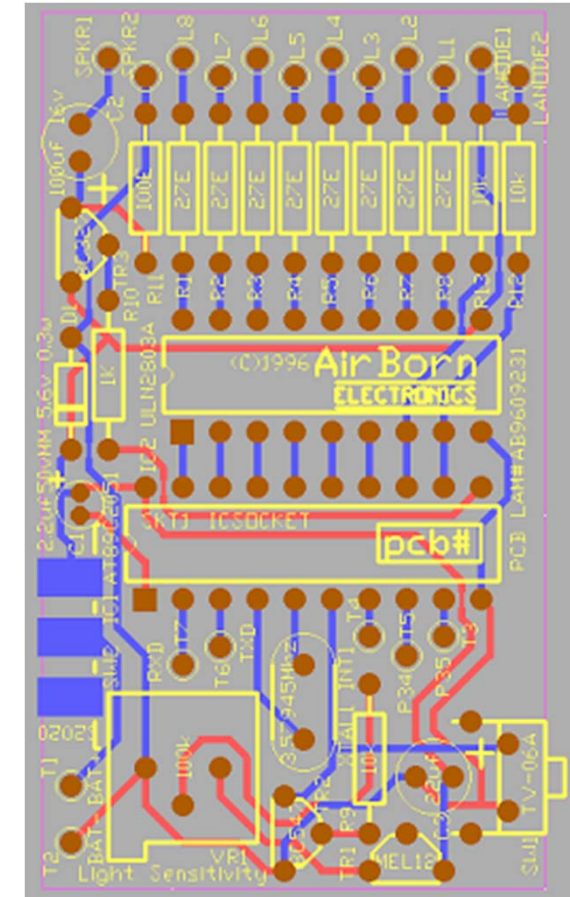
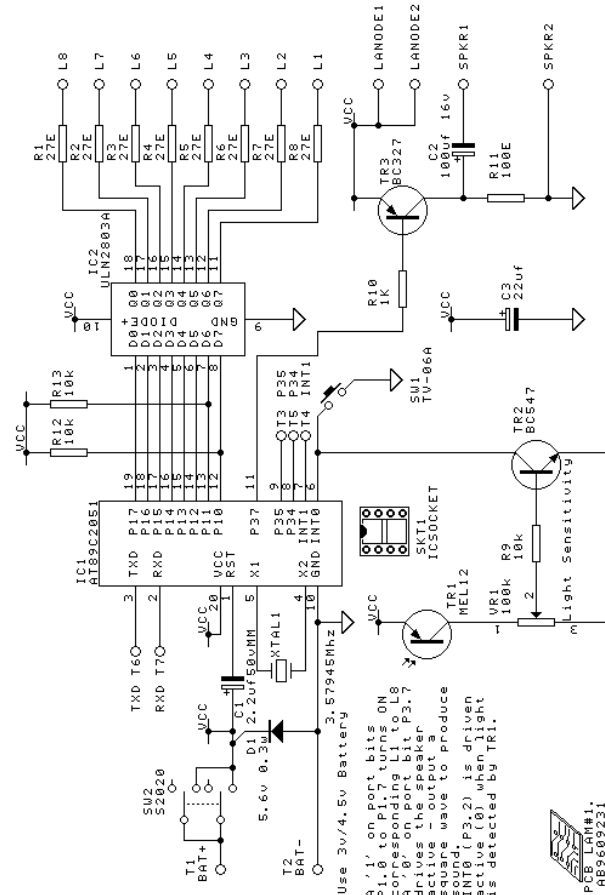
audio_out

...

Function:

When user presses
button1, then...

When...



1. Behavioral

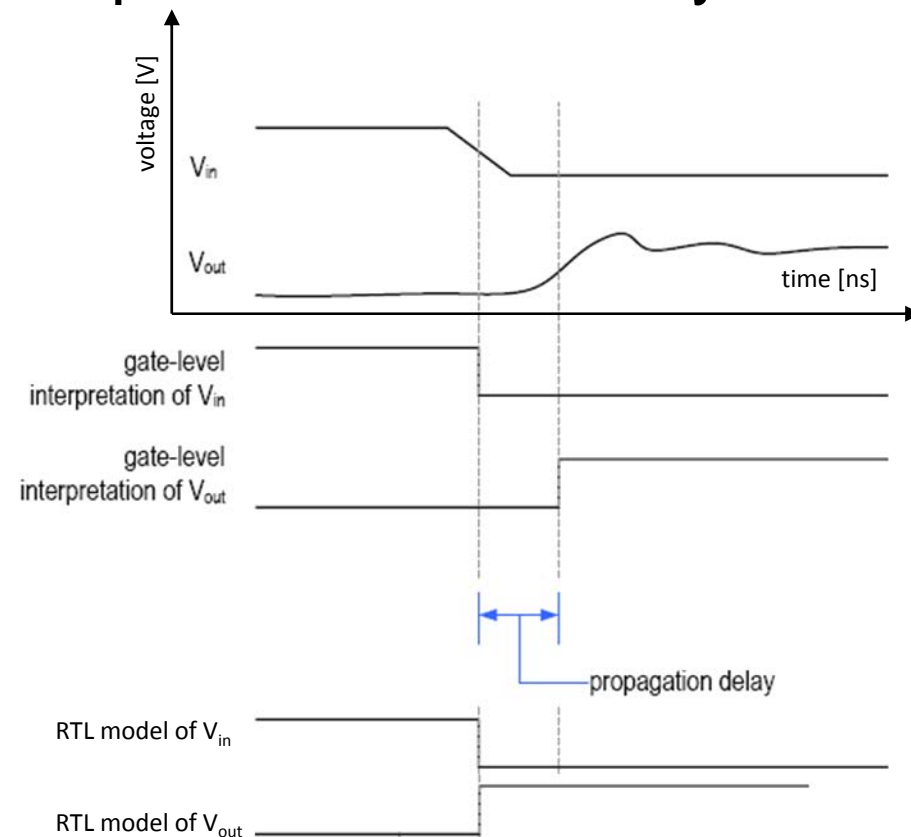
2. Structural

3. Physical

← higher abstraction

Complexity management

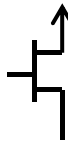

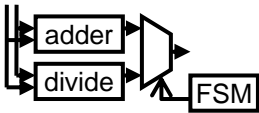
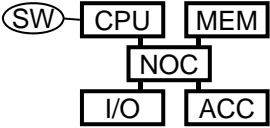
- Q: How to manage complexity for a chip with 10 million transistors?
- A: Abstraction – a simplified model of a system
 - Show the selected features
 - Ignore many details
- E.g., timing of an inverter



Levels of abstraction in HDL

1. Transistor level, lowest abstraction
 2. Gate level
 3. Register transfer level (RTL)
 - Typical level nowadays in addition to structural
 4. Behavioral (Processor) level, highest abstraction
 5. (Manager view: everything works just by snapping fingers...)
- Characteristics of each level
 - Basic building blocks
 - Signal representation
 - Time representation
 - Behavioral representation
 - Physical representation

Summary of abstractions

Level	typical blocks	signal representation	time representation	behavioral description	physical description	Example block	Course
transistor	transistor, resistor	voltage	continuous function	differential equation	transistor layout		ELT-xxxx
gate	and, or, xor, flip-flop	logic 0 or 1	propagation delay	Boolean equation	cell layout		DigiPer. Dig Suunn
RT	adder, mux, register	integer, system state	clock tick	extended FSM	RT level floor plan		<i>Dig.Suunn.</i> this
behavioral	processor, memory	abstract data type	event sequence	algorithm in C	IP level floor plan		System design

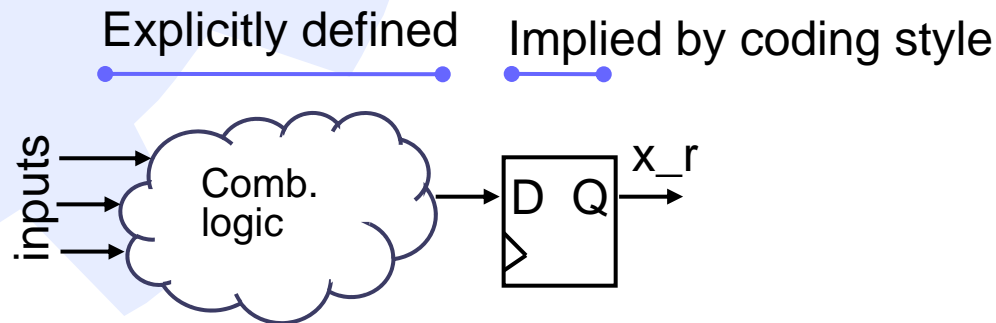
This course focuses on RTL

Behavioral description

- An *untimed* algorithm description with no notation of time or registers (or even interface)
- The tools automatically place the registers according to the constraints set by the designer
- E.g. FFT described in Matlab/C
- The designer gives constraints to a behavioral synthesis tool
 - Maximum latency, clock frequency, throughput, area
 - Interface
- The tool explores the design space and creates the timing-aware circuit
- Not very well supported yet.

Register-transfer level (RTL)

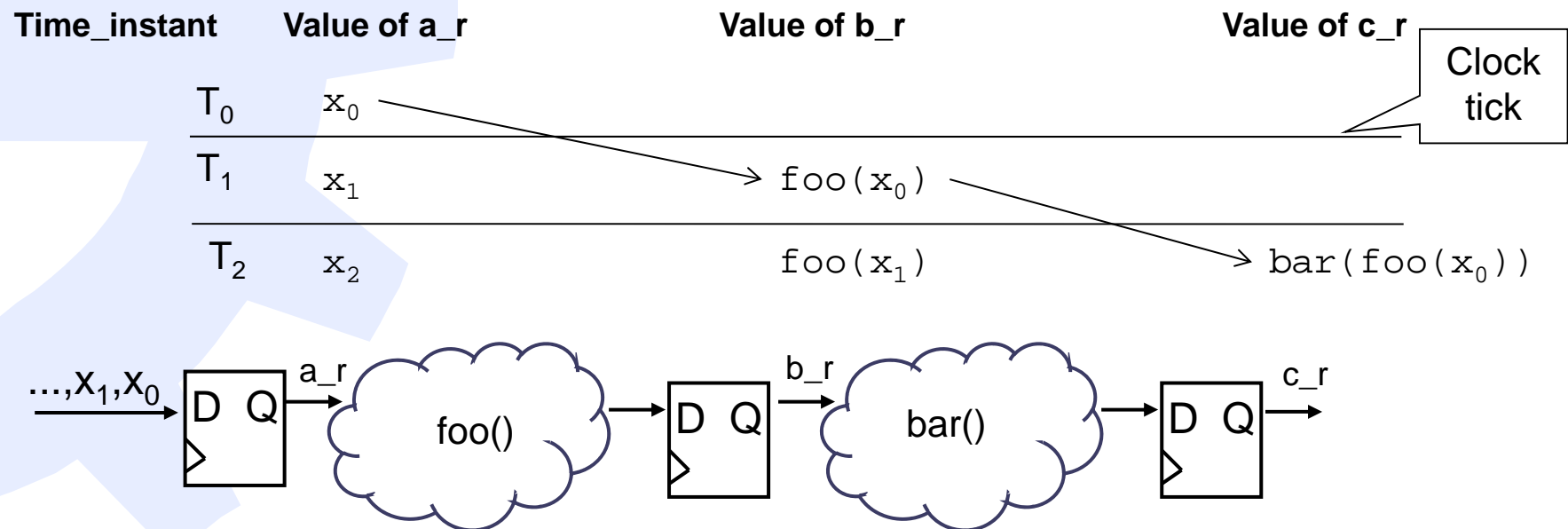
- Typically, HW description languages use RT level
- The idea is to represent the combinational logic before registers
 - The logic between registers, i.e. between register transfers
- The registers are "implied" not explicitly defined in VHDL
 - Synchronous processes imply register and are covered in later lectures
- Comb. logic is created by synthesis tool and depends on
 1. right-hand-side of the signal assignment (e.g. `x_r <= a+b;`)
 2. preceding control structures (`if sel='1',`
`for(i=0;i<9;i++)...`)



Note that you can create purely combinatorial logic using *RTL abstraction*

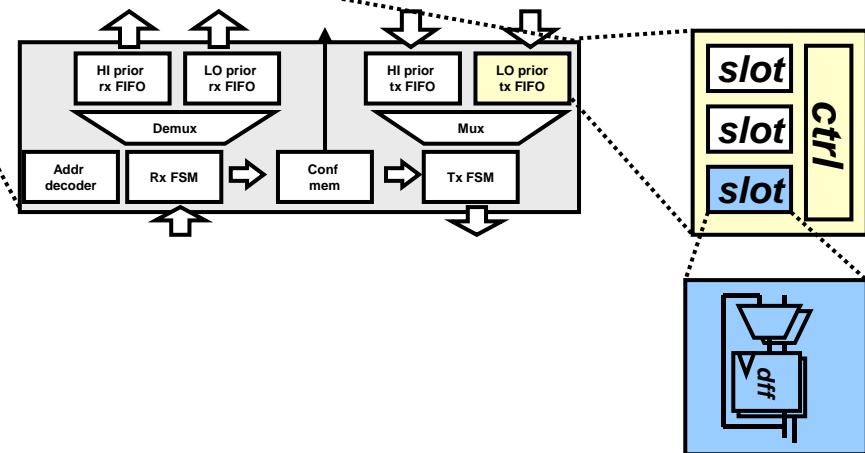
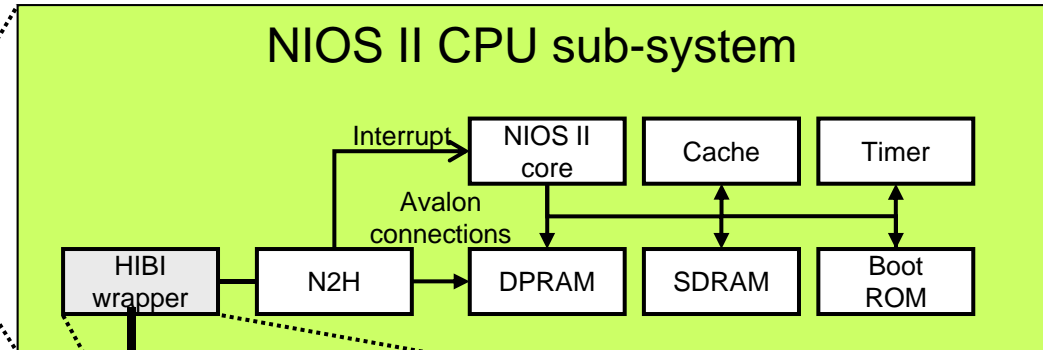
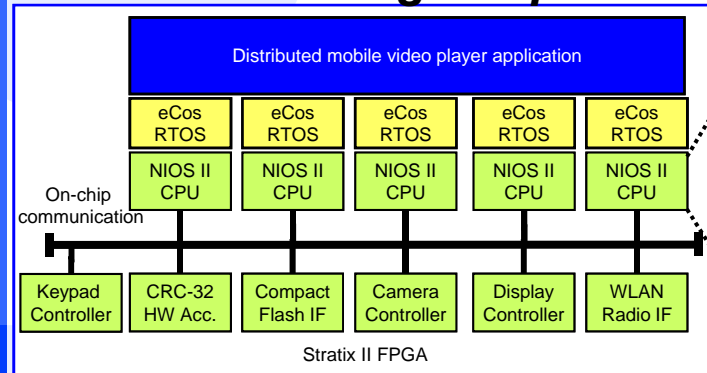
Register-transfer level (RTL) (2)

- RT (Register Transfer) is a bit misleading term
- Two meanings:
 1. Loosely: represent the module level
 2. Formally: a design methodology in which the system operation is described by how the data is manipulated and moved among registers.



Key for success: Hierarchy

Hierarchical design: top level



- All systems are designed and implemented hierarchically
- The same component can be replicated and used in many products
- Usually only knowledge of external behavior is required, not the internals

Structural VHDL description

- Circuit is described in terms of its components.
- High-level block diagram
- Black-box components, modularity
- For large circuits, low-level descriptions quickly become impractical.
- **Hierarchy is very essential to manage complex designs**

Syntax:

Corresponds to the entity

Parameters

Ports

```
component component_name
  generic (
    generic_declaration;
    generic_declaration;
    . . .
  );
  port (
    port_declaration;
    port_declaration;
    . . .
  );
```

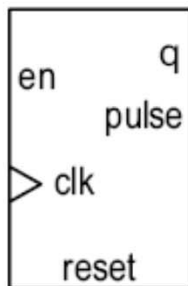
Example

■ A hierarchical two-digit decimal counter

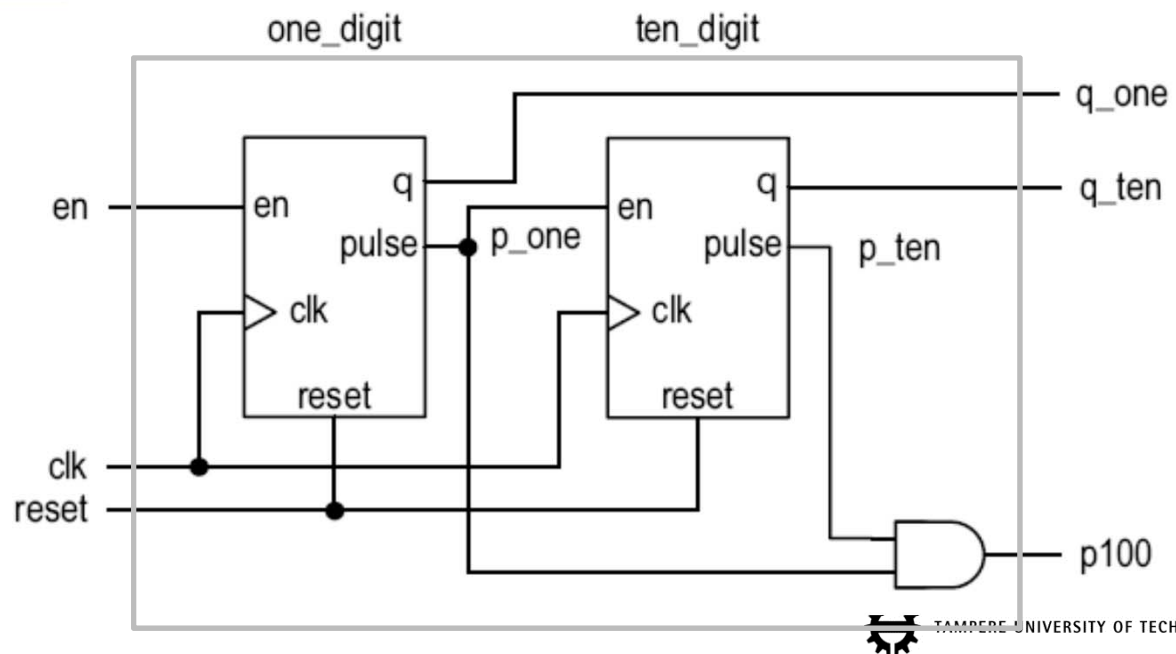
- Pulse=1 when q is 9,
- p100=1 when both q_ten and q_one are 9
- However, if en goes 0, when q=9, something strange happens...

■ Let's concentrate on the structure...

Single
counter
component



Top-level block diagram, "hundred_counter"



Example implemented in VHDL

Top-level entity

```
library ieee;
use ieee.std_logic_1164.all;
entity hundred_counter is
  port(
    clk, reset: in std_logic;
    en: in std_logic;
    q_ten, q_one: out std_logic_vector(3 downto 0)
    p100: out std_logic
  );
end hundred_counter;
```

```
architecture vhd1_87_arch of hundred_counter is
  component dec_counter
    port(
      clk, reset: in std_logic;
      en: in std_logic;
      q: out std_logic_vector(3 downto 0);
      pulse: out std_logic
    );
  end component;
  signal p_one, p_ten: std_logic;
```

Internal signals

```
begin
  one_digit: dec_counter
    port map (clk=>clk, reset=>reset, en=>en,
              pulse=>p_one, q=>q_one);
  ten_digit: dec_counter
    port map (clk=>clk, reset=>reset, en=>p_one,
              pulse=>p_ten, q=>q_ten);
  p100 <= p_one and p_ten;
end vhd1_87_arch;
```

Instantiate the counters as in schematic and connect the signals to ports.

Notation for mapping ports:

<port name in the component> =>
<signal name or port in higher level>

"We will use an existing component called dec_counter"

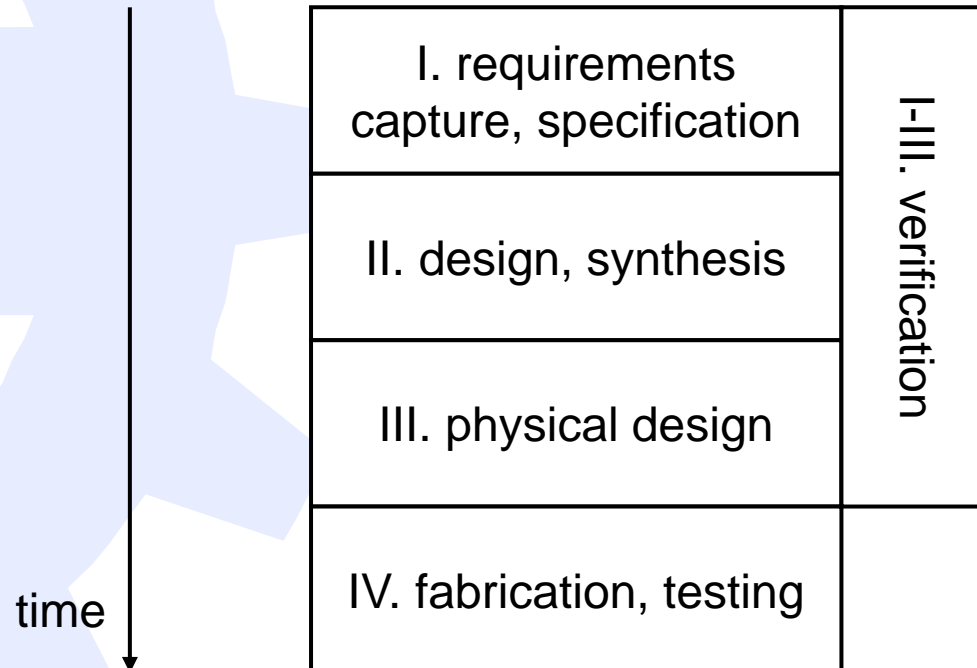


1b. Development Tasks



System development

- Developing a digital system is a refining and validating process
- Main tasks:



I. Specification

- Capture the

1. use cases, requirements
2. non-functional requirements (performance, cost, power consumption, silicon area)

- Usually informal, natural language (English, Finnish) completed with tables and illustrations

- Formal methods are being studied and their importance will increase

II. Design, synthesis

- A refinement process that realizes a description with components from the lower abstraction level
 - Manual/automated
- The resulting description is a structural view in the lower abstraction level
 - A synthesis from VHDL code obtains netlis (gates and flip-flops)
 - Estimates the size, max. frequency and power consumption
- Type of synthesis:
 - High-level synthesis
 - RT level synthesis
 - Gate level synthesis
 - Technology mapping
- **Emphasis of this course**

III. Physical Design

- Placement of cells and routing or wires
 - Refining from structural view to physical view
 - Derive layout of a netlist
- Circuit extraction:
 - Determine wire resistance and capacitance accurately to estimate timing and power
- Others
 - Derivation of power grid and clock distribution network, assurance of signal integrity etc.

I-III. Verification

- Check whether a design *meets the specification* and performance goals
- Concern the correctness of the initial design and the refinement processes
- Two aspects
 1. Functionality (e.g. is the answer 42?)
 2. Non-functional (e.g. performance)
- Takes ~40-80% of design time

I-III. Methods of Verification

1. Simulation

- Spot check: cannot verify the absence of errors
- Can be computationally intensive

2. Hardware emulation with reconfigurable HW

- Almost real-time, connection to external devices

3. Timing analysis

- Just check the worst case delay, automated

4. Formal verification

- Apply formal mathematical techniques to determine certain properties, applicable only in small scale
- E.g, equivalence checking between two models

5. Specification/code review

- Explain the design/spec to others and they comment it
- Surprisingly powerful!

IV. Testing

- Testing is the process of detecting *physical defects* of a die or a package occurred at the time of manufacturing
 - Testing and verification are different tasks in chip design
- Difficult for large circuit
 - Must add auxiliary testing circuit into design
 - E.g., built-in self test (BIST), scan chain etc.
 - Some tests with specialized test-SW running on chip
- Locating the fault is not always needed
 - Faulty chips are simply discarded
- Basic tests are done at wafer-level
 - Sub-set of tests also for packaged chips



1c. Development Flow

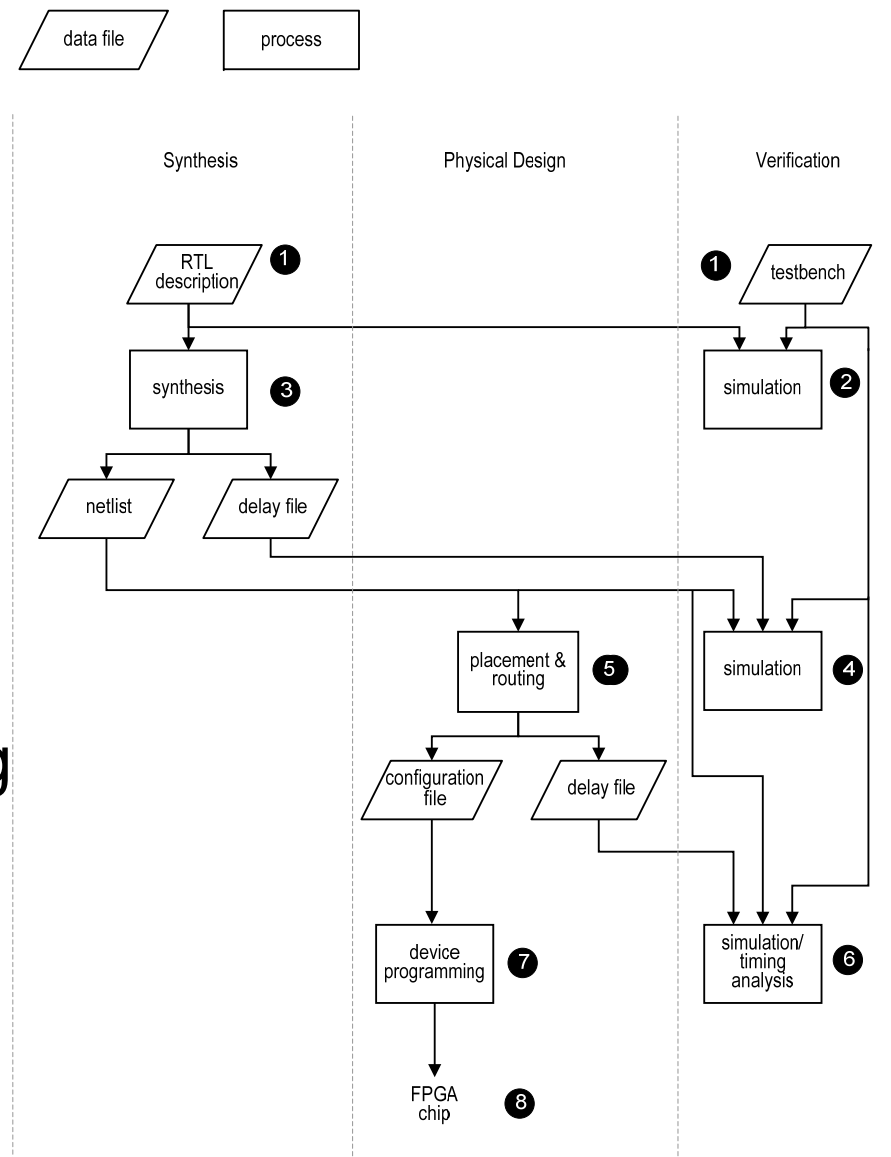



EDA software

- EDA (Electronic Design Automation) software can automate many tasks
- Mandatory for success together with re-use!
- Can software replace human hardware designer? (e.g., C-program to chip)
- Synthesis software
 - should be treated as a tool to perform transformation and local optimization
 - cannot alter the original architecture or convert a poor design into a good one
 - See also the so called “Mead & Conway revolution”
- EDA tools abstraction level in functional description has not increased significantly since mid-90s when RT-level gained popularity
 - Increased abstraction always causes some penalty in performance, area etc. when increasing abstraction, but significant improvement in time to design

Design flow

- Medium design targeting FPGA
- Circuit up to 50,000 gates
- Note that testbench development at the same time as RTL (or before that)
- Large design targeting FPGA need also
 - Design partition
 - More verification
 - I/O-verification, external interfaces



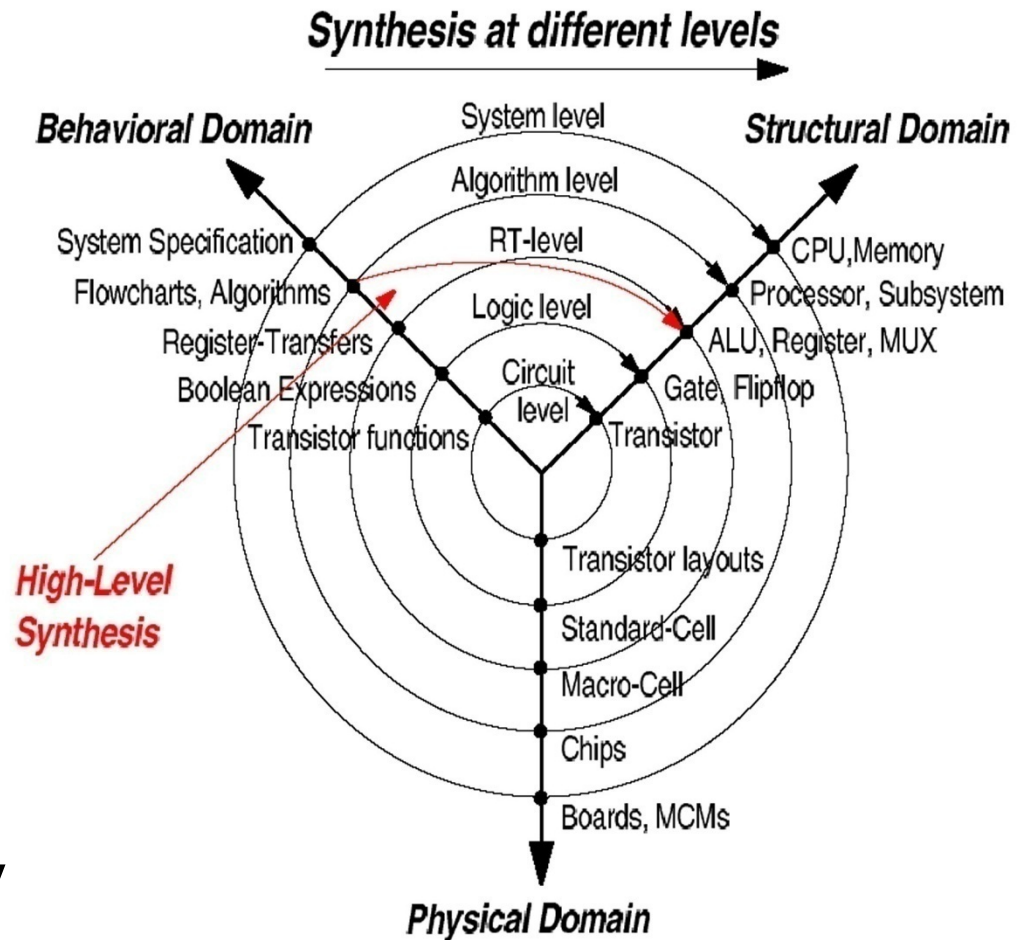


2. Very High Speed Integrated Circuit Hardware Description Language (VHSIC HDL = VHDL)

2a. Basics

Why (V)HDL?

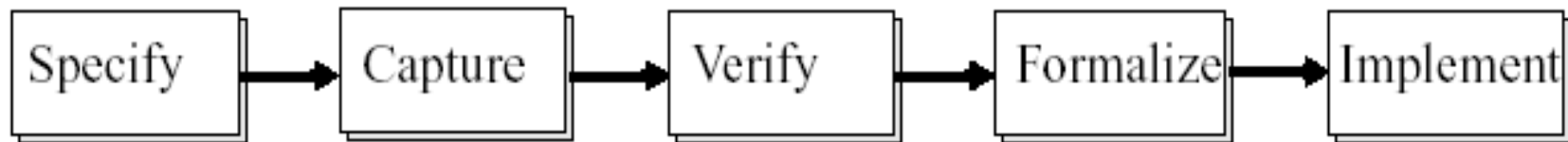
- Interoperability
- Technology independence
- Design reuse
- Several levels of abstraction
- Readability
- Standard language
- Widely supported
- ➔ Improved productivity



What is VHDL?

- **VHDL = VHSIC Hardware Description Language**

- (VHSIC = Very High Speed IC)



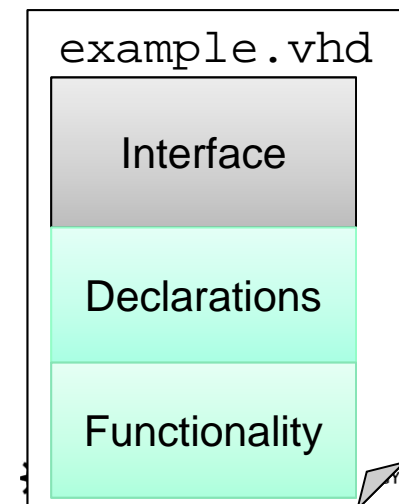
- Design specification language
- Design entry language
- Design simulation language
- Design documentation language
- An alternative to schematics

A brief VHDL history

- Developed in the early 1980s
 - for managing design problems that involved large circuits and multiple teams of engineers
 - originally for documentation, synthesis developed soon after
 - funded by U.S. Department of Defence
- First publicly available version released in 1985
- IEEE standard in 1987 (IEEE 1076-1987)
 - IEEE = Institute of Electrical and Electronics Engineers
- An improved version of the language was released in 1994
 - IEEE standard 1076-1993
 - No major differences to '87, but some shortcuts added

VHDL

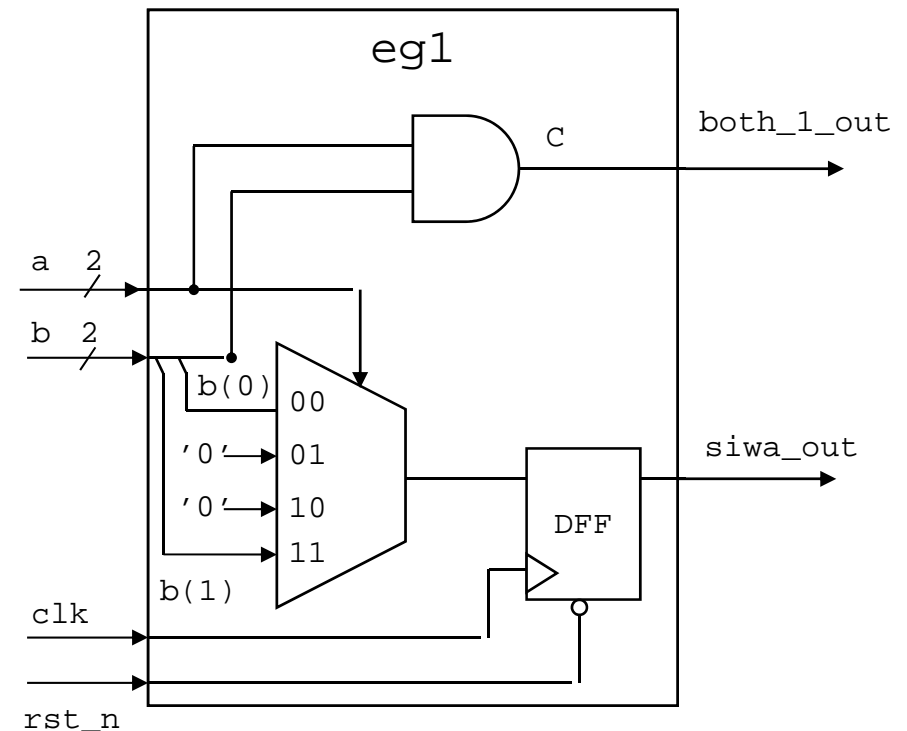
- Parallel programming language(!) for hardware
 - Allows sequential code portions also
- Modular
 - Interface specification is separated from the functional specification
- Allows many solutions to a problem
- The coding style matters!
 - Different solutions will be slower and/or larger than others
 - Save money!
- Case-insensitive language
 - Examples (usually) show reserved words in CAPITALS
- Widely used language



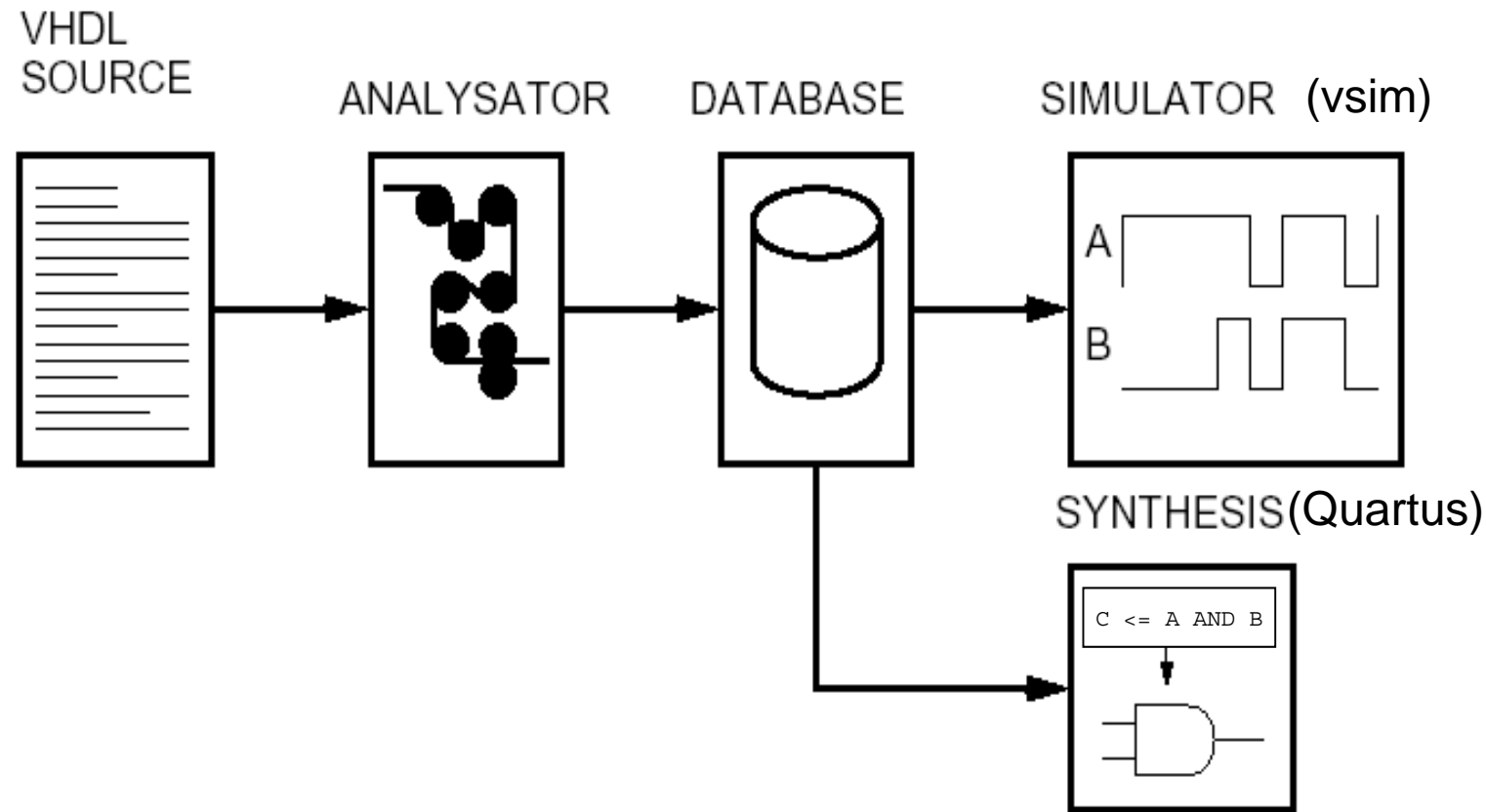
My first VHDL Example

```
ENTITY eg1 IS
  PORT (
    clk      : IN    STD_LOGIC;
    rst_n    : IN    STD_LOGIC;
    a,b      : IN    STD_LOGIC_VECTOR(1 DOWNTO 0);
    both_1_out: OUT   STD_LOGIC;
    siwa_out  : OUT   STD_LOGIC
  );
END eg1;

ARCHITECTURE rtl OF eg1 IS
  SIGNAL c : STD_LOGIC;
BEGIN
  both_1_out <= c;
  c          <= a(0) AND b(0);
  PROCESS ( clk, rst_n )
  BEGIN
    IF rst_n = '0' THEN
      siwa_out <= '0';
    ELSIF clk'EVENT AND clk = '1' THEN
      IF a = '00' THEN
        siwa_out <= b(0);
      ELSIF a = '11' then
        siwa_out <= b(1);
      ELSE
        siwa_out <= '0';
      END IF;
    END IF;
  END IF;
END PROCESS;
END rtl;
```

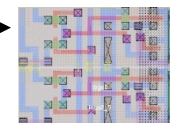


VHDL environment



(Tools used in this course are shown in parentheses)

Physical design



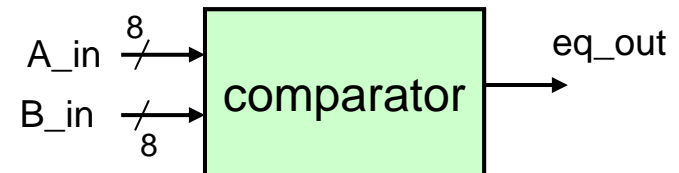
Entities – interfaces

Interface

Declarations

Functionality

- A black box with interface definition
 - Functionality will be defined in *architecture*
- Defines the inputs/outputs of a component (pins)
- Defines the generic parameters (e.g. signal width)
- A way to represent modularity in VHDL
- Similar to symbol in schematic
- Reserved word ENTITY



```
ENTITY comparator IS
PORT (
    a_in    : IN STD_LOGIC_VECTOR(8-1 DOWNT0 0);
    b_in    : IN STD_LOGIC_VECTOR(8-1 DOWNT0 0);
    eq_out  : OUT STD_LOGIC
);
END comparator;
```

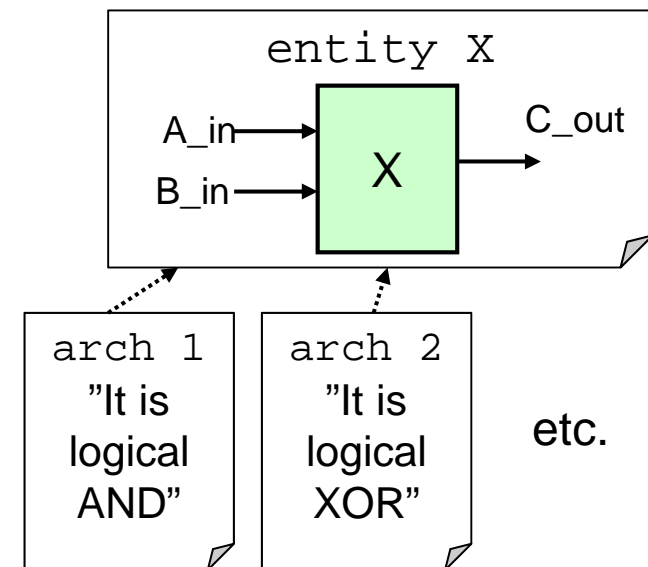
Architecture - internals

Interface

Declarations

Functionality

- Every entity has at least one architecture.
- Architecture specifies the internals of a design unit and is coupled to a certain entity
 - Defines functionality
- One entity can have several architectures
- Architectures can describe design on many levels
 - Gate level
 - RTL (*Register Transfer Level*)
 - Structural
 - Behavioral level



Architecture (2)

■ Example:

```
ARCHITECTURE rtl OF comparator IS
BEGIN
    eq_out <= '1' WHEN (a_in = b_in) ELSE '0';
END rtl;
```

■ Two main approaches

1. Define new functionality with control statements, e.g. if-for-case, (`rtl`), shown above
2. Instantiate existing components and define interconnections between them (`structural`)

Ports

Interface

Declarations

Functionality

- Provide communication channels (=pins) between the component and its environment
- Each port must have a name, direction and a type.
 - An entity may omit port declaration, e.g. in testbench
- Port directions:
 1. IN: A value of a port can be read inside the component, but cannot be assigned. Multiple reads of port are allowed.
 2. OUT: Assignments can be made to a port, but data from a port cannot be read. Multiple assignments are allowed.
 3. INOUT: Bi-directional, assignments can be made and data can be read. Multiple assignments are allowed. (not recommended inside a chip)
 4. BUFFER: An out port with read capability. May have at most one assignment (not recommended)

Signals

Interface

Declarations

Functionality

- Used for communication inside the architecture, carry data
 - Ports in behave like signals
- Can be interpreted as
 - a) Wires (connecting logic gates)
 - b) “wires with memory” (i.e., FFs, latches etc.)
- VHDL allows many types of signals
 - Bit vectors, integers, even multidimensional arrays and records.
- Declared in the architecture body's declaration section
- Signal declaration:
`SIGNAL signal_name : data_type;`
- Signal assignment:
`signal_name <= new_value;`

Other declarations

Interface

Declarations

Functionality

- Functions, procedures (subprograms)
 - Much like in conventional programming languages
- Component declaration
 - "We will use an adder that looks like this"
- Configuration
 - "We will use exactly this adder component instead of that other one"
 - Binds certain architecture to the component instance

Libraries and packages

Interface

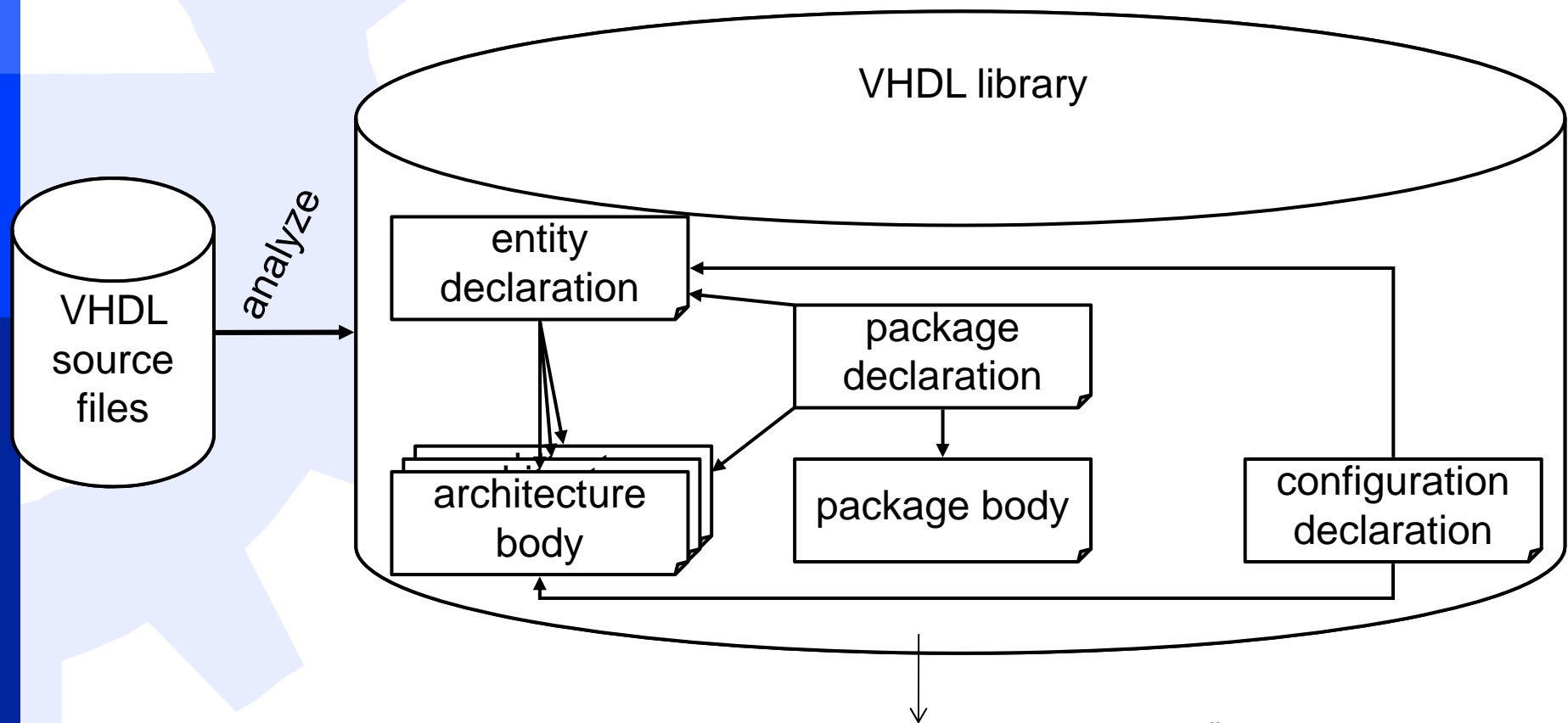
Declarations

Functionality

- Frequently used functions and types can be grouped in a package
- Libraries include several compiled packages and other design units
- Packages typically contain
 - Constants
 - Like header.h in conventional programming languages
 - General-purpose functions
 - E.g. Log2(x)
 - Design-specific definitions
 - E.g. own data types, records (structs)

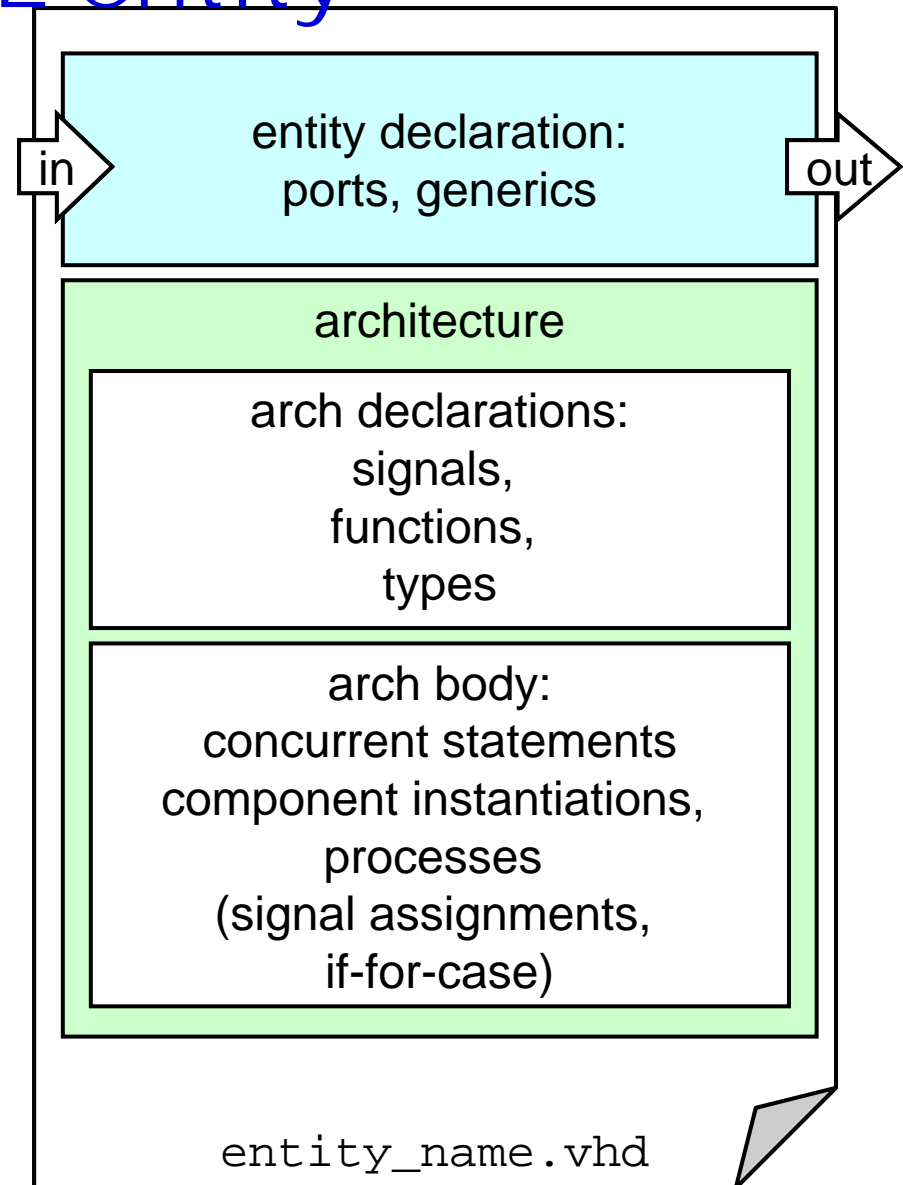
Design units

- Segments of VHDL code that can be compiled separately and stored in a library
- Library = directory of compiled VHDL files



Structure of VHDL entity

- Usually one entity plus one architecture per file
 - File named according to entity
- Architectures contains usually either
 - a) Processes, or
 - b) instantiations

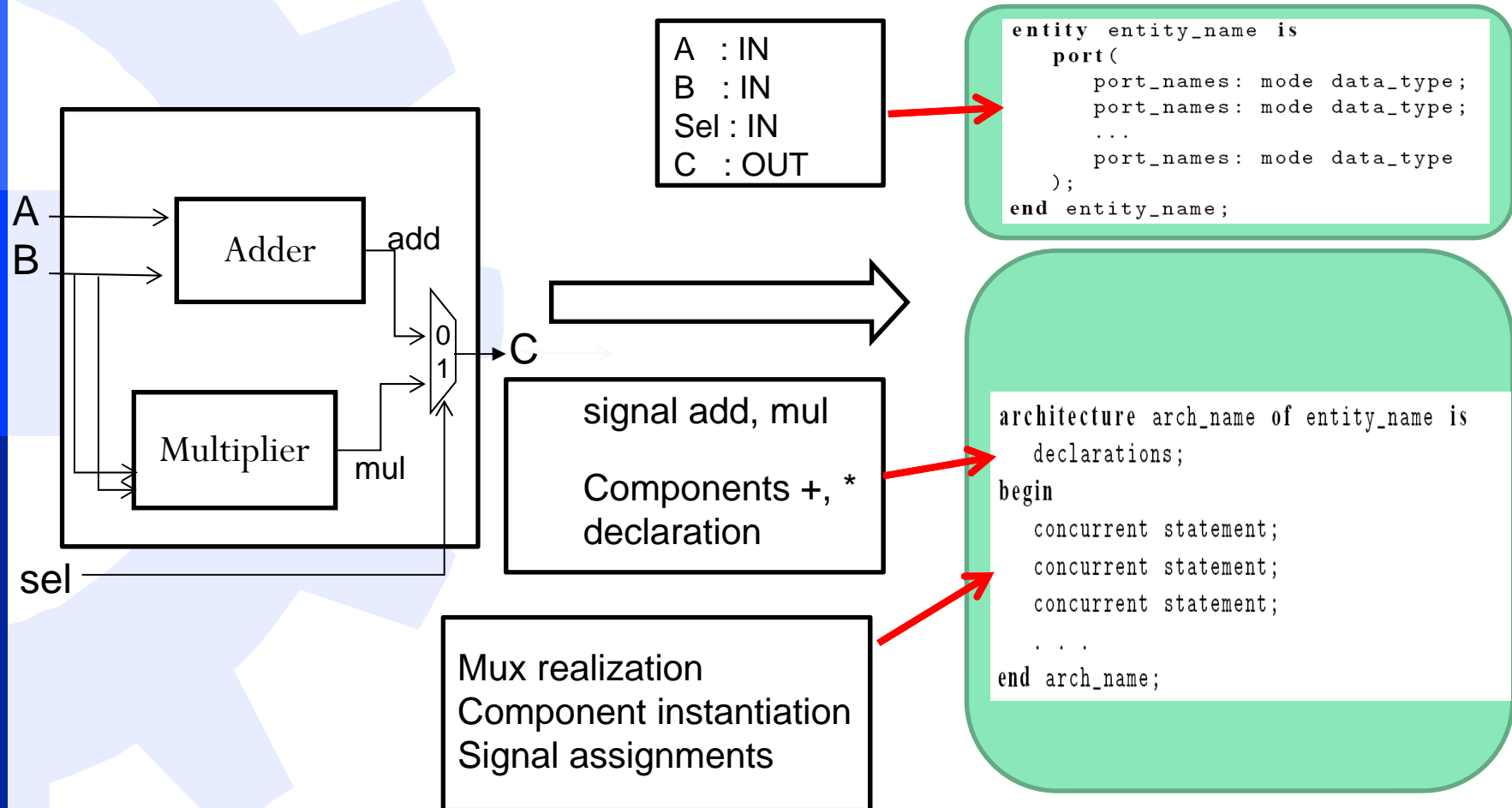


Relation between circuit and VHDL

Realization

Examples

GENERIC VHDL STRUCTURE



Even parity detection circuit

- Input: $a(2)$, $a(1)$, $a(0)$
- output: even

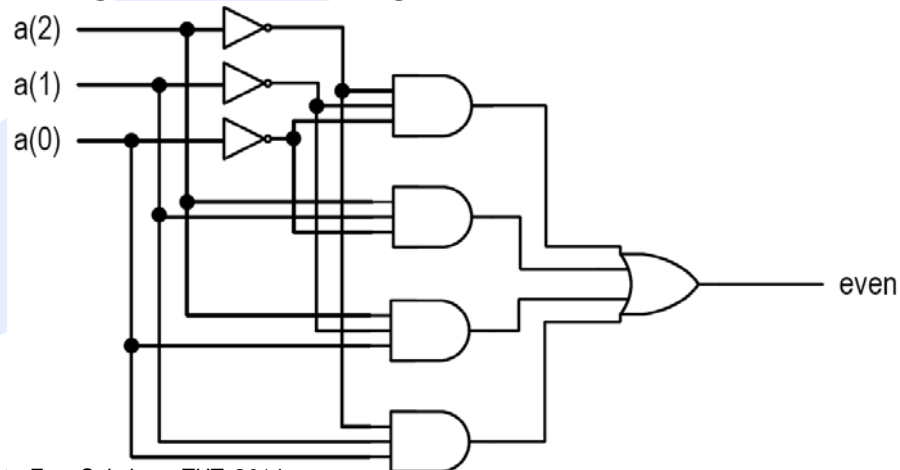
Boolean function:

$$\text{even} = a(2)' \cdot a(1)' \cdot a(0)' + a(2)' \cdot a(1) \cdot a(0) + a(2) \cdot a(1)' \cdot a(0) + a(2) \cdot a(1) \cdot a(0)'$$

Truth table:

$a(2)$	$a(1)$	$a(0)$	even
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Logic with basic gates:



Even parity detection circuit at gate-level VHDL

Defines packages that are used in the design

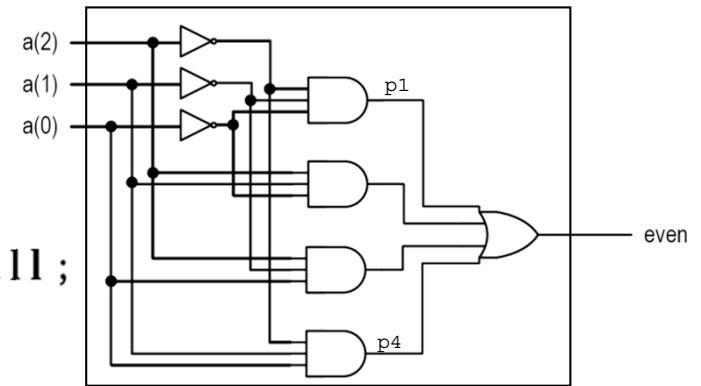
The interface of a block "even_detector"
Input a, 3 bits
Output even, 1 bit

Signals used internally

Functionality of the block (gate level representation). The order of assignments does NOT matter here.

```
library ieee;
use ieee.std_logic_1164.all;
entity even_detector is
    port (
        a: in std_logic_vector(2 downto 0);
        even: out std_logic);
end even_detector;

architecture eg_arch of even_detector is
    signal p1, p2, p3, p4 : std_logic;
begin
    even <= (p1 or p2) or (p3 or p4);
    p1 <= (not a(0)) and (not a(1)) and (not a(2));
    p2 <= (not a(0)) and a(1) and a(2);
    p3 <= a(0) and (not a(1)) and a(2);
    p4 <= a(0) and a(1) and (not a(2));
end eg_arch ;
```



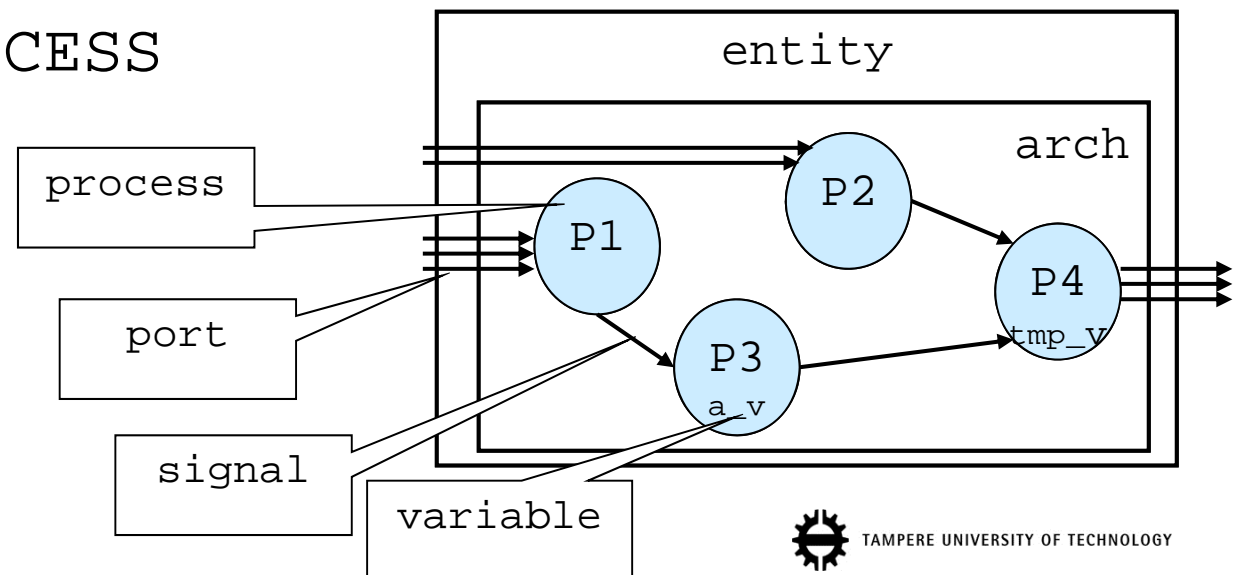


2b VHDL Processes



Process

- Basic modeling concept of VHDL
- The *whole process is a concurrent statement*
 - i.e. processes are executed *in parallel*
- Contains a set of *statements that are executed sequentially*
- VHDL description can always be broken up to interconnected processes
- Keyword PROCESS



Process (2)

- Processes are the basic building blocks of functional (in most cases that means RTL) description
- Practically every design has at least one process
- In VHDL, the flip-flops are generated with (synchronous) processes
 - No reserved word for registers in VHDL
 - Synthesis/simulation tools recognize certain process structures that implicate signals as registers (set of D-flip-flops)
 - To be covered in later lectures

Process (3)

- Resides in the architecture's body
 - A process is like a circuit part, which can be
 - a) active (known as *activated*)
 - b) inactive (known as *suspended*)
 - Its statements will be executed sequentially top-down until the end of the process
 - Written order of statements matters, unlike in concurrent statements
 - However, all signal assignments take place when process exits
 - Forgetting this is a Top-3 mistake for beginners
- ```
b <= 1; -- b was 5
c <= b; -- c gets the old value of b, i.e. 5
```
- Last assignment to a signal will be kept

# Process's sensitivity list

- A process is activated when any of the signals in the *sensitivity list* changes its value
- Process must contain either sensitivity list or *wait* statement(s), but NOT both.
  - Similar behavior, but sensitivity list is much more common

## ■ General format:

```
label: PROCESS(sensitivity_list)
 process_declarative_part
BEGIN
 process_statements
 [wait_statement]
END PROCESS;
```

Either but not both.

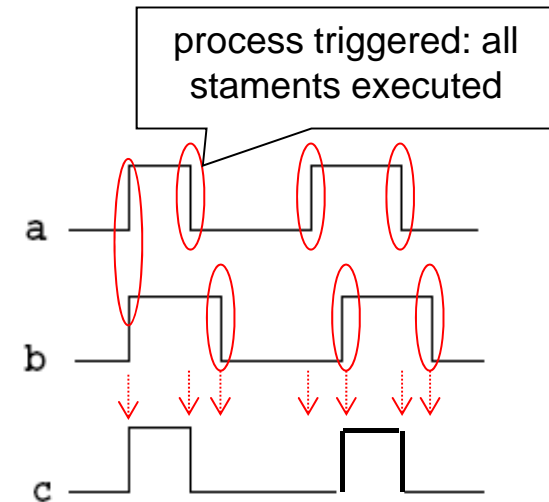
# Example sensitivity list

## ■ Process with sensitivity list:

```
ex_p: PROCESS (a, b)
BEGIN
 c <= a AND b;
END PROCESS ex_p;
```

## ■ Process is executed when value of a or b changes

- Type of a and b can be arbitrary: scalar, array, enumeration, or record
- ex\_p is a user defined label (recommended)



## Example (2)

- The same process with wait statement:

```
PROCESS
```

```
BEGIN
```

```
 WAIT ON a,b;
```

```
 c <= a AND b;
```

```
END PROCESS;
```

Wait for change on a or b,  
as in prev slide

- Bad process with incomplete sensitivity list:

```
PROCESS (a)
```

```
BEGIN
```

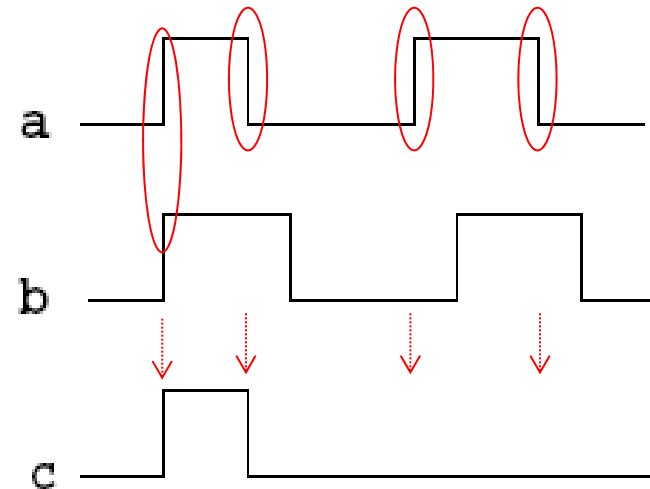
```
 c <= a AND b;
```

```
END PROCESS;
```

Trigger only when  
a changes

Not evaluated when b changes  
(simulation does not match synthesis  
!!!). **superbad**.

simulation: process  
with incomplete  
sensitivity list



# Example: last assignment is kept

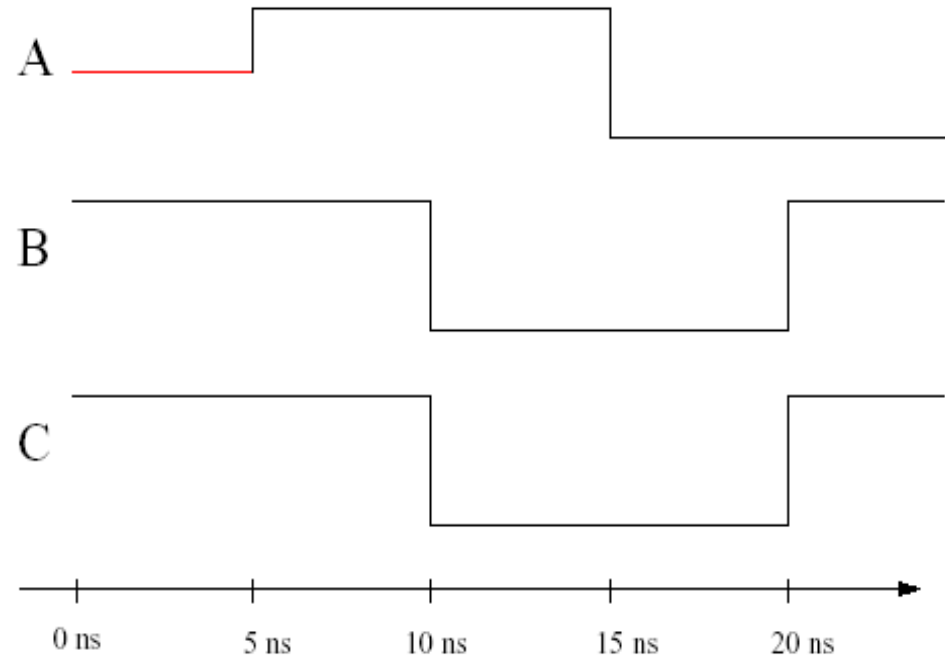
```
ENTITY Pitfall1 IS
END Pitfall1;

ARCHITECTURE behav OF
 Pitfall1 IS
 SIGNAL A, B, C :
 std_logic;
 BEGIN
 A <= '1' AFTER 5 ns,
 '0' AFTER 15 ns,
 '1' AFTER 25 ns;
 B <= '1' AFTER 0 ns,
 '0' AFTER 10 ns,
 '1' AFTER 20 ns;

 PROCESS (A, B)
 BEGIN -- process
 C <= A;
 ...
 C <= B;
 END PROCESS;
 END behav;
```

Input wave  
generation,  
(sim only)

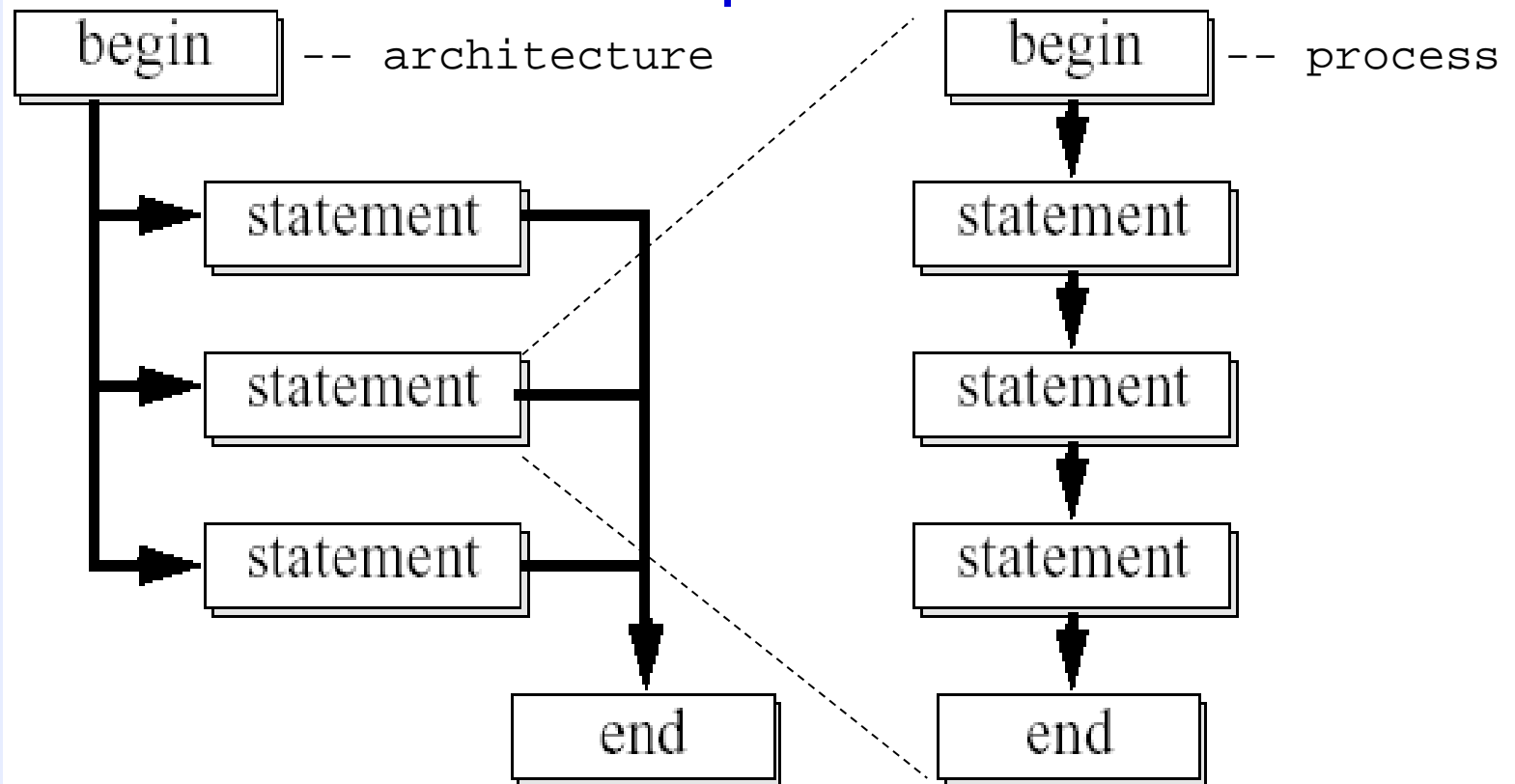
OK



Only the last assignment,  $C \leq B$ , is kept.

However, this is also useful. In a complex process, one can assign a default value at first and then override it later in some branch.

# Concurrent vs. sequential VHDL



| Modeling style     | Concurrent                                                | Sequential                                                   |
|--------------------|-----------------------------------------------------------|--------------------------------------------------------------|
| Location           | Inside architecture                                       | Inside process or function                                   |
| Example statements | process, component instance, concurrent signal assignment | if, for, switch-case, signal assignment, variable assignment |

# Concur./seq. VHDL : Example

```
ARCHITECTURE rtl OF rotate_left IS
```

```
 SIGNAL rot_r : STD_LOGIC_VECTOR(7 DOWNT0 0);
```

```
 BEGIN
```

```
 shift : PROCESS(rst, clk)
```

```
 BEGIN
```

```
 IF (rst = '1') THEN
```

```
 rot_r <= (others => '0'); -- reset the register
```

```
 ELSIF (clk = '1' AND clk'EVENT) THEN
```

```
 IF (load_en_in = '1') THEN
```

```
 rot_r <= data_in; -- store new value
```

```
 ELSE
```

```
 rot_r (7 DOWNT0 1) <= rot_r(6 DOWNT0 0);
```

```
 rot_r (0) <= rot_r(7);
```

```
 END IF;
```

```
 END IF;
```

```
 END PROCESS;
```

```
 q_out <= rot_r; -- connect DFF's output to output port
```

```
 END concurrent_and_sequential;
```

Concurrent

Sequential





## 2c. Signals, Variables, Constants



# Signals

- Signals carry the data
- Two possible assignment styles:
  1. Sequential signal assignments
    - Inside a process
    - Evaluated only whenever a process is triggered, e.g. every clock cycle
  2. Concurrent signal assignments
    - Outside any process
    - Used in the concurrent portion of the architecture
      - Typically these are simple signal-to-output or signal-to-signal assignments
    - "Always-on" assignments, continuously evaluated

## Signals (2)

### ■ Signal assignment operator is `<=`

- NOTE: Sequential and concurrent signal assignments look similar
- They are distinguished by their location on code (inside or outside a process)
- General form:

```
target <= [transport] expr
 [after t_expr{, expr after t_expr}];
```

- Assigns a future and/or current value(s)

### ■ Entity's ports are also signals

- Note that an out-port cannot be read

### ■ Signal is declared outside the processes

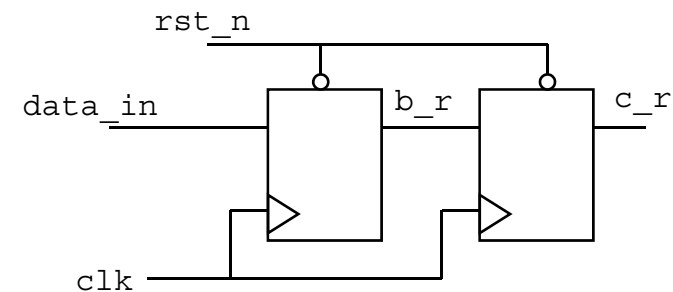
- In the architecture declaration

### ■ Location and right-hand side of the assignment infer some comb. logic (e.g. addition) or just a simple wire

# 1. Sequential Signal Assignments

- Signal assignment inside a process (or a subprogram).
- Assignments are executed every time an event occurs on any of the signals in the sensitivity list of the process.
- A process can have only *one driver* for a signal
  - Assignments are scheduled and do not occur until the process has been suspended
  - The last assignment takes effect when the process suspends
- Example: `data_in` has changed from 5 to 1, note that order of assignment does not actually matter here

```
PROCESS (clk, rst_n)
BEGIN
 IF (rst_n = '0') THEN
 . . .
 ELSIF (clk'EVENT AND clk='1') THEN
 b_r <= data_in; -- b is set to 1
 c_r <= b_r; -- c gets the old value of b, i.e. 5
 END IF;
END;
```



## 2a) Concurrent Signal Assignment

- Happens outside a process
- Assignment is executed any time an event occurs in the right-hand side.
- Examples

```
a <= '1'; -- These two are
data_out <= data_r; -- the most common
```

- (test bench code can use delays, after ignored in synthesis):

```
color <= red after 5 ns, yellow after 1 ns;
line <= transport b after 1 ps;
```

```
-- After/trasport delays are used only in
-- simulation. Synthesis does create an
-- assignment but without any delay (just
-- a wire)
```

# 1,2) Conditional Signal Assignment

## ■ Concurrently:

```
target <= value1 WHEN cond1 ELSE
 value2 WHEN cond2 ELSE
 value3;
```

## ■ Equivalent process:

```
PROCESS (cond1, cond2, value1, value2, value3)
BEGIN
 IF cond1 THEN
 target <= value1;
 ELSIF cond2
 target <= value2;
 ELSE
 target <= value3;
 END IF;
END PROCESS;
```

# 1,2) Selected Signal Assignment

## ■ Concurrently:

```
WITH expression SELECT
target <= value1 WHEN choice1,
 value2 WHEN choice2,
 value3 WHEN OTHERS;
```

## ■ Equivalent process:

```
PROCESS(expression,value1,value2,value3)
BEGIN
 CASE expression IS
 WHEN choice1 =>
 target <= value1;
 WHEN choice2 =>
 target <= value2;
 WHEN OTHERS =>
 target <= value3;
 END CASE;
END PROCESS;
```

Note that only a single case branch is executed. (No need for break commands like in C)

Unfortunately, case has some limitations compared to if-elsif. Expression must be 'locally static', i.e. value can be determined inside the design where it appears. E.g. actual value of a generic can be set from upper level, so it's not locally static

# VHDL semantics

- If a process is executed and a certain signal is not assigned, it will keep its previous value
  - Good and bad sides...
  - Example:

```
process (a,b)
begin
 if (a=b) then
 eq <= '1';
 end if ;
end process;
```

implies

```
process (a,b)
begin
 if (a=b) then
 eq <= '1';
 else
 eq <= eq; --'0' to create XNOR
 end if ;
end process
```

**= results in latch!**

**(in comb. processes, always include the *else-branch*)**

Note that this feature simplifies sequential processes.

Note also that eq is never nullified in example. Once it's high, it stays high until the end of the world



# Variables

- A storage facility with a single current value
- Variable value changes instantly as opposed to signal
  - Somewhat alike to variables in C, C++
  - But differs in many ways from regular programming languages
- Can be used in:
  - 1. Processes
  - 2. Functions
  - 3. Procedures
- No global variables in VHDL '87. They are local to a process or a function

```
PROCESS (...)
 VARIABLE Q1_v : STD_LOGIC;
 BEGIN -- PROCESS
 Q1_v := '0';
 Q2_v := not Q1_v; -- Q2 sees the "new" value of Q1
```

## Variables (2)

- Note: variable assignment operator is `:=`
- Variables should only be used as an intermediate storage
  - Short-hand notation to beautify the code
- Variables in processes retain their values during simulation, but in functions and procedures they do not
- However, never re-use variable value between iterations!
  - Re-using the variable value may will result in combinatorial feedback loop, which can be a catastrophe
  - Re-using the variable obfuscates the code
- Variables are slightly faster to simulate than signals
- Not recommended for processes. Later lecture provides more info why not

# Variables: example

## ■ THIS IS WRONG!

```
testi: process (clk, rst_n)
 variable temp_v : std_logic;
begin -- process testi
 if rst_n = '0' then
 c_out <= '0';
 temp_v := '0'; -- naughty habit

 elsif clk'event and clk = '1' then
 temp_v := temp_v and a_in(0);
 c_out <= temp_v or a_in(1);

 end if;

end process testi;
```

Variable is read before it is written inside the elsif branch, and hence it must retain the old value. It is not a good custom to infer registers with variables (although possible)

**Fig. The circuit we are trying to create:**

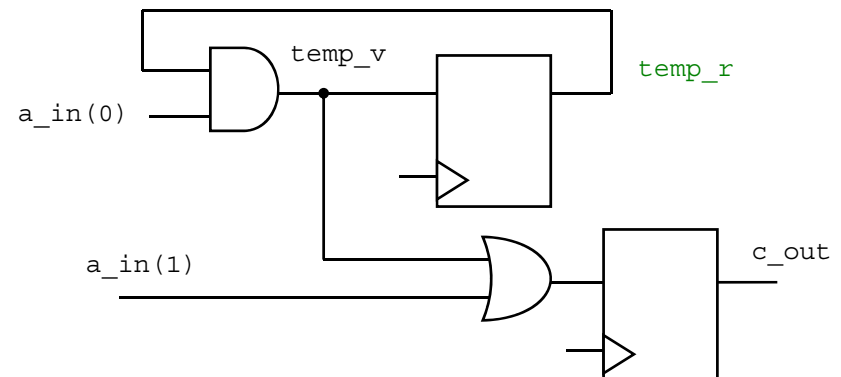
## ■ THIS IS RIGHT (but variable is not very useful)

```
testi: process (clk, rst_n)
 variable temp_v : std_logic;
begin -- process testi
 if rst_n = '0' then
 c_out <= '0';
 temp_r <= '0';

 elsif clk'event and clk = '1' then
 temp_v := temp_r and a_in(0);
 c_out <= temp_v or a_in(1);
 temp_r <= temp_v;

 end if;

end process testi;
```



# Constants

- An object that has a constant value and cannot be changed
- The value of constant is assigned when constant is declared
- May be declared globally (within packages) or locally
  - Constant declarations can be located in any declaration area
- Clarify the code as *magic numbers* get a symbolical, descriptive name

```
CONSTANT send_data_c : BOOLEAN := TRUE;
CONSTANT base_addr_c : STD_LOGIC_VECTOR(8-1 DOWNT0 0)
:= "10010001";
```





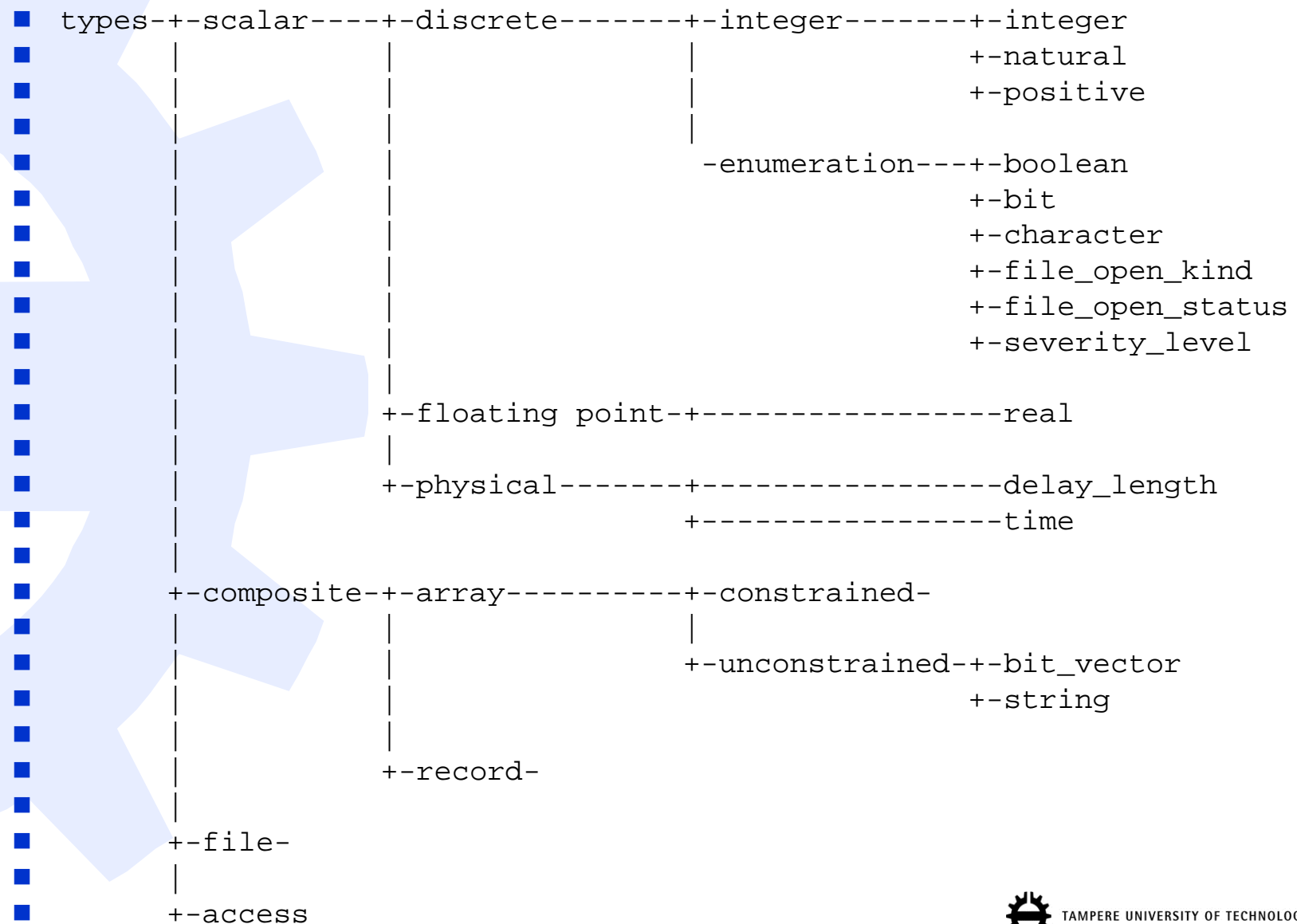
## 2d. Types in VHDL



# Types

- VHDL is strongly typed language.
- 1. Scalar types
  - integer types
  - enumeration types
  - physical types
  - real (floating point) types
- 2. Composite types
  - array types
  - record types
- 3. File Types
- 4. Access types

<http://www.cs.umbc.edu/help/VHDL/types.html>



# 1a. Scalar types: Integer

## ■ Minimum range (in VHDL '87):

- Symmetric 32-bit range
- From -2147483647 to +2147483647
- i.e.  $-(2^{31}-1)$  to  $2^{31}-1$ ,
- Simulator dependent, can be larger

## ■ Predefined integer types (built-in standard package):

```
TYPE INTEGER IS RANGE -xxxx TO yyyy;
SUBTYPE NATURAL IS INTEGER RANGE 0 TO INTEGER'HIGH
SUBTYPE POSITIVE IS INTEGER RANGE 1 TO INTEGER'HIGH
```

## ■ Examples of user defined integer types:

```
TYPE bit_int IS INTEGER RANGE 0 TO 1;
TYPE byte IS INTEGER RANGE 0 TO 255;
TYPE word IS INTEGER RANGE 0 TO 65535;
-- You should define integer range explicitly to avoid
-- area and delay overheads
```



# Peculiarities of integer

- Omitting negative values does **not** increase maximum value
  - Although in C/C++ it does
  - Note the name; `unsigned` is separate type (vector)
  - There is no (confusing) specifier `long` either
- VHDL standard defines that integer must support 32-bit “one’s complement range”
  - This way inverse number is always valid
- However, RTL Synthesis standard specifies that signed integer should be represented in two’s complement, supporting range of  $-(2^{31})$  to  $2^{31}-1$ 
  - The range of the used tool can be checked with attribute, `integer'low` and `integer'high`.
  - Modelsim and Quartus support this range
  - [http://ieeexplore.ieee.org/xpls/abs\\_all.jsp?tp=&isnumber=18052&arnumber=836335&punumber=6748](http://ieeexplore.ieee.org/xpls/abs_all.jsp?tp=&isnumber=18052&arnumber=836335&punumber=6748)

## 1b. Scalar types: Enumeration

- Has a set of user defined values
- Ordered
- Pre-defined: BIT, BOOLEAN, SEVERITY\_LEVEL, CHARACTER

- Examples of enumeration type declarations:

```
TYPE SEVERITY_LEVEL IS (NOTE,WARNING,ERROR,FAILURE);
TYPE BIT IS ('0', '1'); -- package STD
TYPE mybit IS ('0', '1', 'X', 'Z');
TYPE opcode IS (add, sub, lda);
TYPE state IS (idle, start, read, stop);
```

- NOTE:

- Enumeration literals can be overloaded. Look definitions of BIT and mybit
- Type of the enumeration literal must be determinable from the context

## 1c. Scalar types: Real

- REAL is the only predefined floating point type.
- Minimum range from -1E38 to +1E38.
  - simulator (implementation) dependent

- Examples:

```
TYPE int_real IS REAL RANGE -2147483647.0 TO
 +2147483647.0;
```

```
-- Conversion from integer to real as follows
r := REAL(i);
```

- NOTE:

- Some synthesis tools may support floating point types. To be safe, use only in simulation
- Fixed-point representation is often accurate enough
- <http://www.eda.org/fphdl/>



## 1d. Scalar types: Physical

- Represent physical quantities
- TIME: minimum range from -2147483647 to +2147483647, 32 bit range
  - only for simulation,
  - not synthesizable, there isn't logic to produce accurate arbitrary delay
  - very much simulator dependent

```
TYPE TIME IS RANGE -1E18 to 1E18
```

```
UNITS
```

```
fs;
```

```
ps = 1000 fs;
```

```
ns = 1000 ps;
```

```
us = 1000 ns;
```

```
ms = 1000 us;
```

```
sec = 1000 ms;
```

```
min = 60 sec;
```

```
END UNITS;
```



## 2a. Composite type: Array

- A collection of one or more values or objects of the same type.
- Unconstrained and constrained declarations possible
  - unconstrained declarations are good for re-using and generic descriptions
- Support slicing: one-dimensional array is created constraining a larger array.
- Arrays can be returned from functions.
- Multidimensional arrays possible
  - >3D may not be supported in synthesis

## 2a. Composite type: Array (2)

- Indexed with regular parentheses '()'

- Examples:

```
TYPE bit_vector IS ARRAY (NATURAL RANGE <>) OF BIT
```

```
TYPE defvec IS ARRAY (1 to 10) OF BIT;
```

```
TYPE string IS ARRAY (POSITIVE RANGE <>)
 OF BIT;
```

```
TYPE matrix IS ARRAY (INTEGER RANGE <> ,
 INTEGER RANGE <>)
 OF BIT;
```

```
-- using unconstrained array needs bounds
SIGNAL addr_r : bit_vector (7 downto 0);
SIGNAL ctrl_c : defvec;
```

## 2b. Composite type: Record

- Collection of objects with same class
  - constant, variable or signal
- Elements can be of any type
- Fields can be referenced with selected name notation (recname.fieldname)

```
TYPE location IS RECORD
 x : INTEGER;
 y: INTEGER;
END RECORD;
TYPE ififo IS RECORD
 rd_ptr: INTEGER;
 wr_ptr: INTEGER;
 data : real_array;
END RECORD;
SIGNAL coord_r : location;
...
coord_r.x <= 42;
y_out <= coord_r.y;
```

### 3. File handling: VHDL '87

- Sequential stream of objects
  - Last file element is end of file (EOF) mark
  - File elements can be read (file is IN mode)
  - Elements can be written to file (file is OUT mode)

- Built-in file type declaration:

```
TYPE <type_name> IS FILE OF <object_type>;
```

- User's file object declaration

```
FILE file_identifier : <type_name> IS
 MODE <file_name>;
```

- Procedure READ(file,data), reads data from file
- Procedure WRITE(file,data), writes data to file
- Function ENDFILE(file) checks EOF
- Package STD.TEXTIO contains functions for text file manipulation





### 3. File example in VHDL '87

```
access_files : process (clk, rst_n)
 -- vhd'87 syntax
 file my_in_file : text is in "input.txt";
 file my_out_file : text is out "output.txt";
 variable in_line_v, out_line_v : line;
 variable tmp_v : integer := 0;
begin
 if rst_n = '0' then
 -- ...
 elsif (clk'event and clk = '1') then
 while (not ENDFILE(my_in_file)) loop

 READLINE(my_in_file, in_line_v);
 READ(in_line_v, tmp_v);
 -- many flavors of read() available

 WRITE(out_line_v, tmp_v);
 WRITELINE(my_out_file, out_line_v);
 end loop;
 end if;
end process access_files;
```

### 3. Whole '87 example (2)

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

entity test_file87 is
end test_file87;
architecture behav of test_file87 is
 signal clk : std_logic := '0';
 signal rst_n : std_logic;
begin -- behav
 rst_n <= '0', '1' after 50 ns;
 clk <= not clk after 10 ns;
 access_files : process (clk, rst_n)
 -- vhd'87 syntax
 file my_in_file : text is in "input.txt";
 file my_out_file : text is out "output.txt";
 variable in_line_v, out_line_v : line; -- type "LINE" is a pointer to a string
 variable tmp_v : integer := 0;
 begin
 if rst_n = '0' then -- asynchronous reset (active low)

 elsif (clk'event and clk = '1') then -- rising clock edge

 while (not ENDFILE(my_in_file)) loop
 -- This loop reads the whole file in single clk cycle.
 -- Only the first string from each line is converted to
 -- integer and the rest are ignored.
 -- Runtime error occurs if a line does not start with integer.
 READLINE(my_in_file, in_line_v);
 READ(in_line_v, tmp_v); -- many flavors of read() available
 WRITE(out_line_v, tmp_v);
 WRITELINE(my_out_file, out_line_v);
 end loop;

 end if;
 end process access_files;
end behav;
```

### 3. File types: VHDL '93

- Quite similar to '87 but files can be opened and closed as needed.

- File operation functions in VHDL '93:

```
FILE_OPEN(<file_identfier>, <file_name>, <file_mode>);
```

```
FILE_OPEN(<status>, <file_identfier> , <file_name>,
 <file_mode>);
```

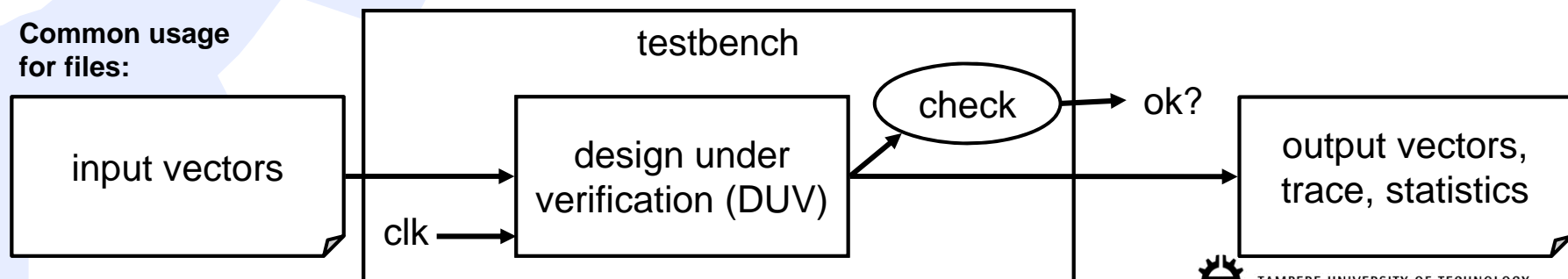
```
FILE_CLOSE(<file_identfier>);
```

- File modes are:

- READ\_MODE (file is read-only)
- WRITE\_MODE (file is write-only, initially empty)
- APPEND\_MODE (file is write-only, output will be added to the end of the file)



Common usage  
for files:





# More complex example in VHDL'93

```
access_files : process (clk, rst_n)
 -- vhd'93 syntax, only these 2 lines differ from previous in minimal case
 file my_in_file : text open read_mode is "input.txt";
 file my_out_file : text open write_mode is "output.txt";
 variable in_line_v, out_line_v : line;
 variable tmp_v : integer;
 variable valid_number : boolean := false;
 variable curr_line : integer := 0;
begin
 if rst_n = '0' then -- asynchronous reset (active low)
 ...
 elsif (clk'event and clk = '1') then -- rising clock edge
 valid_number := false;
 -- Loop until finding a line that is not a comment.
 while valid_number = false and not(endfile(my_in_file)) loop

 READLINE(my_in_file, in_line_v); -- a) read from file, b) from terminal: READLINE(input, in_line_v);
 READ (in_line_v, tmp_v, valid_number); -- 3rd param tells if ok
 curr_line := curr_line+1; -- just for reporting

 if valid_number = false then
 report "Skipped the line " & integer'image(curr_line) & " (it's comment or malformed)" severity note;
 next; -- start new loop interation
 end if;

 -- Another way for debug printing, LF = line feed = new line
 write (output,string'("Got value " & integer'image(tmp_v)& " at t:" & time'image(now) & LF));

 WRITE(out_line_v, tmp_v);
 WRITELINE(my_out_file, out_line_v); -- a) write to file, b) to terminal: WRITELINE(output, out_line_v);
 end loop;
 end if;
end process access_files;
```

## 4. Access types

- Very similar to C pointers (suitable for LIFO/FIFO modelling).
- Two predefined functions NEW and DEALLOCATE.
- Only variables can be declared as access type
- Very rare. Not synthesizable.
- Example declarations and usage of new and deallocate:

```
TYPE point_loc IS ACCESS LOCATION;
VARIABLE pl1_v, pl2_v, pl3_v: point_loc;
```

```
pl1_v := NEW location; -- new object is created
pl2_v := pl1; -- pl1 points to same obj as pl2
pl3_v := NEW location;
pl1_v := pl3_v;
DEALLOCATE(pl2_v);
```



# Summary of VHDL types

most common

| Type name                      | Sim.<br>only | Note                                                                                                       |
|--------------------------------|--------------|------------------------------------------------------------------------------------------------------------|
| std_logic,<br>std_logic_vector |              | Actually enumeration, you'll need pkg ieee_1164, use these instead of bit/bit_vector, use downto indexing  |
| integer                        |              | Limit range for synthesis                                                                                  |
| unsigned, signed               |              | Similar to std_logic_vector, but safer for arithmetic                                                      |
| array                          |              | E.g. std_logic_vector is array. Define the array type first and then signal/constant/variable of that type |
| enumeration                    |              | bit and std_logic are actually enumerations, use this at least for states of an FSM                        |
| record                         |              | Synthesizable, but not very common                                                                         |
| file                           | x            | For reading input data and storing trace/log during simulation-based verification                          |
| physical                       | x            | For detailed gate-level simulation with timing                                                             |
| real                           | x            | Quite rare because cannot be (always) synthesized                                                          |
| access                         | x            | Very rare                                                                                                  |



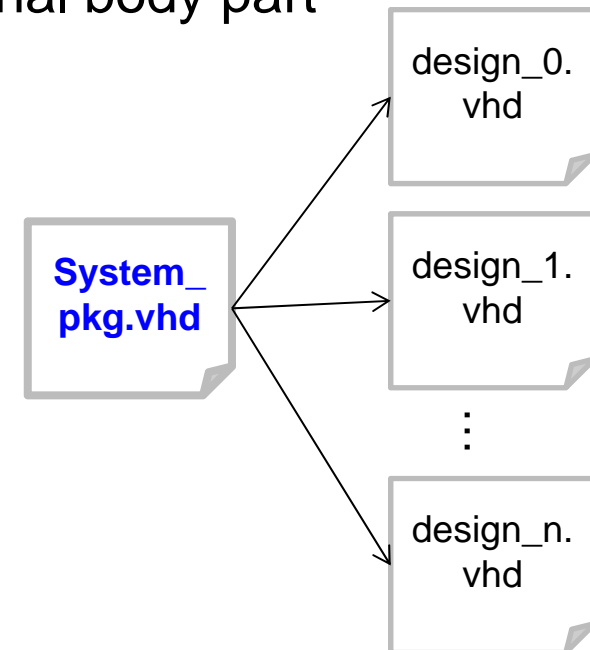


## 2e. Packages



# Packages

- Meant for encapsulating data which can be shared globally among several design units.
- Consists of declaration part and optional body part
- Package declaration can contain:
  - **type and subtype declarations**
  - **subprograms**
  - **constants, alias declarations**
  - file declarations
  - global signal declarations
  - component declarations
- Package body consists of
  - type and subtype declarations
  - subprogram declarations and bodies
  - deferred constants (avoids some re-compilation, rare concept)
  - file declarations





# Package example

```
PACKAGE example_pkg IS
```

```
 CONSTANT example_c : STD_LOGIC := '1';
```

```
 FUNCTION integer_to_vector
```

```
 (size : INTEGER; number : INTEGER)
```

```
 RETURN STD_LOGIC_VECTOR;
```

```
END example_pkg;
```

```
PACKAGE BODY example_pkg IS
```

```
 FUNCTION integer_to_vector
```

```
 (size : INTEGER; number : INTEGER)
```

```
 RETURN STD_LOGIC_VECTOR IS
```

```
 ... --insert the implementation here
```

```
 END integer_to_vector;
```

```
END example_pkg;
```

# Package example (2)

```
LIBRARY IEEE;
```

```
USE IEEE.STD_LOGIC_1164.all;
```

```
USE IEEE.STD_NUMERIC.all;
```

```
PACKAGE io_pkg IS
```

```
 CONSTANT addr_width_c : NATURAL := 16;
```

```
 CONSTANT data_width_c : NATURAL := 16;
```

```
 CONSTANT stat_c : NATURAL := 1;
```

```
 CONSTANT total_out_c : NATURAL := 10;
```

```
 TYPE o_bits_arr IS ARRAY (0 to total_out-1)
 OF NATURAL;
```

```
 FUNCTION inmux(
 data : STD_LOGIC_VECTOR(data_width_c-1 downto 0);
 sel : NATURAL)
 RETURN STD_LOGIC_VECTOR;
```

```
END io_pkg;
```

```
-- Function inmux will be defined in package body
```

# Libraries

- Collection of *compiled* VHDL design units (database)
  - 1. Packages
    - package declaration
    - package body
  - 2. Entities (entity declaration)
  - 3. Architectures (architecture body)
  - 4. Configurations (configuration declarations)
- Some pre-defined, e.g. STD and IEEE
- One can also create own libraries
  - at least a library called `work`
- To use e.g. package, it must be compiled to some library first (typically to *work*)

## Libraries (2)

- All entity names etc. must be unique within a library
- If you two different entitites with the same name, e.g. `fifo`, you must compile them into separate libraries
- You must define which `fifo` to use for each component instance
  - Either during instantiation or with separate configuration

# Using packages and components

1. Packages and entities are first compiled into some *library* (subdirectory on hard disk)
  - Compilation command is `vcom`
2. Command `vlib` tells the path to the simulator
  - VHDL file can refer to that library with symbolic name like `ieee` or `work`
3. In VHDL file, introduce first what *libraries* are used
  - `work` is the default name, no need to introduce
4. Then, tell what *packages* are used
5. Then, tell what stuff is used from the package
  - Function names, types etc, usually "all"

```
library <libname>;
```

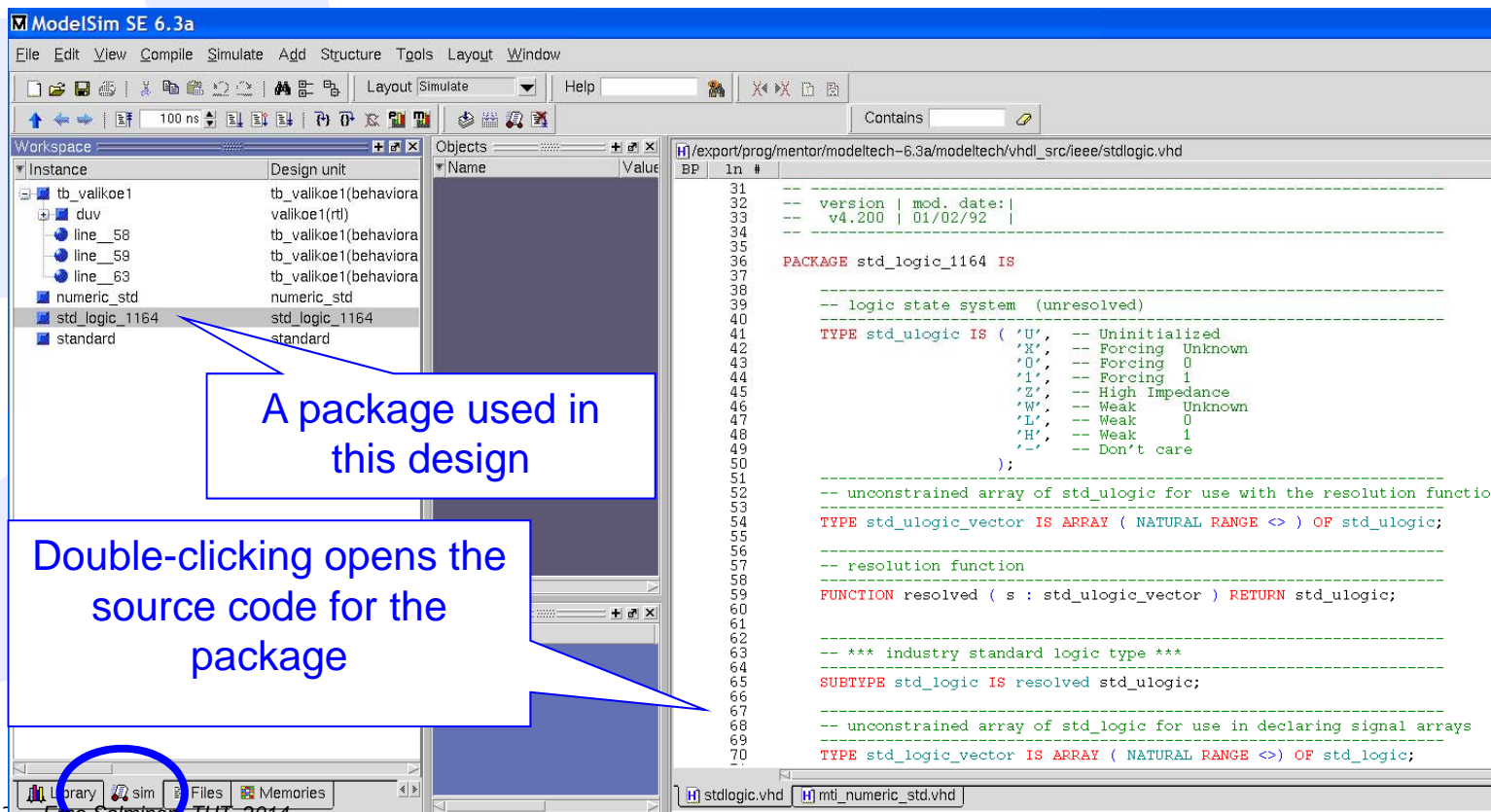
```
use <libname>.<pkg_name>.<stuff>;
```

## Using packages and components (2)

- Standard IEEE packages are the most common
  - Compiled automatically during the simulator installation
  - Referred using symbolic name `ieee`
  - Most common package is `std_logic_1164`
- Sometimes, you need others
  - E.g. special simulation models for FPGA-related primitive components

# Browsing the package contents

- a) Read the files directly from installation directory
  - Something like: /export/prog/mentor/modeltech-6.3a/modeltech/vhdl\_src/ieee/stdlogic.vhd
- b) Start simulation of a design, open either tab "sim" or "Files", and double-click some package





## 2f. Standard packages





# Data types of standard VHDL

- integer:

- Minimal range:  $-(2^{31}-1)$  to  $2^{31}-1$
- Two subtypes: natural, positive

- boolean: (false, true)

- bit: ('0', '1')

- Not capable enough, but we'll return to that...

- bit\_vector: a one-dimensional array of bit

# Operators in standard VHDL

| operator               | description    | data type<br>of operand a   | data type<br>of operand b | data type<br>of result      |
|------------------------|----------------|-----------------------------|---------------------------|-----------------------------|
| <code>a ** b</code>    | exponentiation | integer                     | integer                   | integer                     |
| <code>abs a</code>     | absolute value | integer                     |                           | integer                     |
| <code>not a</code>     | negation       | boolean, bit,<br>bit_vector |                           | boolean, bit,<br>bit_vector |
| <code>a * b</code>     | multiplication | integer                     | integer                   | integer                     |
| <code>a / b</code>     | division       |                             |                           |                             |
| <code>a mod b</code>   | modulo         |                             |                           |                             |
| <code>a rem b</code>   | remainder      |                             |                           |                             |
| <code>+ a</code>       | identity       | integer                     |                           | integer                     |
| <code>- a</code>       | negation       |                             |                           |                             |
| <code>a + b</code>     | addition       | integer                     | integer                   | integer                     |
| <code>a - b</code>     | subtraction    |                             |                           |                             |
| <code>a &amp; b</code> | concatenation  | 1-D array,<br>element       | 1-D array,<br>element     | 1-D array                   |

# Operators (2)

|                 |                          |                          |           |                                                            |
|-----------------|--------------------------|--------------------------|-----------|------------------------------------------------------------|
| a <b>sll</b> b  | shift left logical       | bit_vector               | integer   | bit_vector                                                 |
| a <b>srl</b> b  | shift right logical      |                          |           |                                                            |
| a <b>sla</b> b  | shift left arithmetic    |                          |           | Note that shift is <b>not</b> defined for std_logic_vector |
| a <b>sra</b> b  | shift right arithmetic   |                          |           |                                                            |
| a <b>rol</b> b  | rotate left              |                          |           |                                                            |
| a <b>ror</b> b  | rotate right             |                          |           |                                                            |
| a = b           | equal to                 | any                      | same as a | boolean                                                    |
| a /= b          | not equal to             |                          |           |                                                            |
| a < b           | less than                | scalar or 1-D array      | same as a | boolean                                                    |
| a <= b          | less than or equal to    |                          |           |                                                            |
| a > b           | greater than             |                          |           |                                                            |
| a >= b          | greater than or equal to |                          |           |                                                            |
| a <b>and</b> b  | and                      | boolean, bit, bit_vector | same as a | boolean, bit, bit_vector                                   |
| a <b>or</b> b   | or                       |                          |           |                                                            |
| a <b>xor</b> b  | xor                      |                          |           |                                                            |
| a <b>nand</b> b | nand                     |                          |           |                                                            |
| a <b>nor</b> b  | nor                      |                          |           |                                                            |
| a <b>xnor</b> b | xnor                     |                          |           |                                                            |

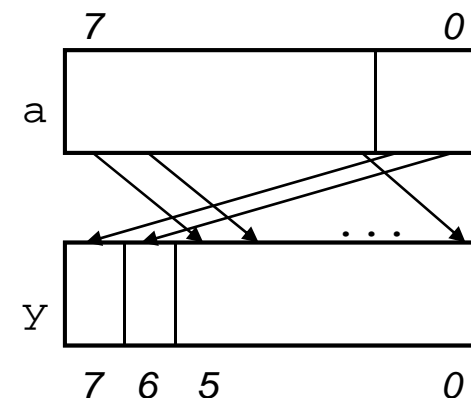
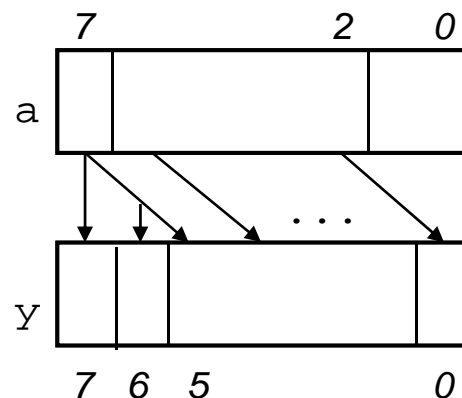
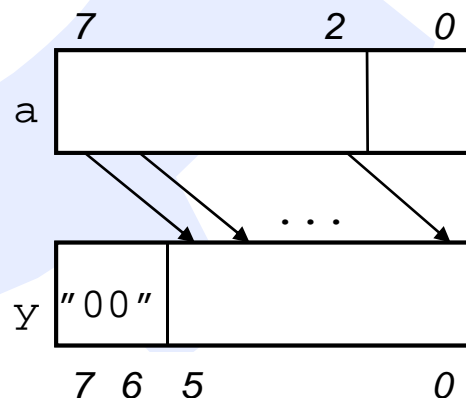
# Concatenation

- Concatenation operator (&)
- Attaches multiple signals together into array

```
y <= "00" & a(7 DOWNTO 2);
```

```
y <= a(7) & a(7) & a(7 DOWNTO 2);
```

```
y <= a(1 DOWNTO 0) & a(7 DOWNTO 2);
```



# Array aggregate

- Aggregate is a VHDL construct to assign a value to an array-typed object

- Different types supported, E.g.,

```
a <= "10100000"; --direct
```

```
a <= (7=>'1', 6=>'0', 0=>'0', 1=>'0',
 5=>'1', 4=>'0', 3=>'0', 2=>'1');
```

```
a <= (7|5=>'1', 6|4|3|2|1|0=>'0');
```

```
a <= (7|5=>'1', others=>'0');
```

- E.g., setting all elements at the same time

```
a <= "00000000" -- Size of a has to be
 -- known
```

```
a <= (others=>'0'); -- Size not needed,
 -- Flexible, Good for
 -- reset. Superb!
```

# IEEE std\_logic\_1164 package

- 'Bit' is too limited having only 2 possible values
- Introduce extended data types
  - std\_logic
  - std\_logic\_vector
- std\_logic: 9 values: ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-')
  - '0', '1': forcing logic 0 and forcing logic 1
  - 'Z': high-impedance, as in a tri-state buffer.
  - 'L', 'H': weak logic 0 and weak logic 1,
    - As in wired-OR and wired-AND logic (pull-down/pull-up resistors)
  - 'X', 'W': “unknown” and “weak unknown”
  - 'U': for uninitialized
  - '-': don't-care

## IEEE std\_logic\_1164 package (2)

### ■ std\_logic\_vector

- an array of elements with std\_logic data type
- Implies a *bus* (=set of signals)

### ■ Recommended form is descending range

```
signal a : std_logic_vector(7 downto 0);
```

### ■ Another form (less desired, do not use)

```
signal b : std_logic_vector(0 to 7);
```

- Always be consistent within in a design
- Assigning  $a \leq b$  or  $b \leq a$  will be confusing

### ■ Need to invoke package to use the data type:

```
library ieee;
```

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

# Overloaded operator

## IEEE std\_logic\_1164 package

- Which standard VHDL operators can be applied to std\_logic and std\_logic\_vector?
- Overloading: same operator of different data types
- Overloaded operators in std\_logic\_1164 package

| overloaded operator | data type of operand a        | data type of operand b | data type of result |
|---------------------|-------------------------------|------------------------|---------------------|
| <b>not</b> a        | std_logic_vector<br>std_logic |                        | same as a           |
| a <b>and</b> b      |                               |                        |                     |
| a <b>or</b> b       |                               |                        |                     |
| a <b>xor</b> b      | std_logic_vector              | same as a              | same as a           |
| a <b>nand</b> b     | std_logic                     |                        |                     |
| a <b>nor</b> b      |                               |                        |                     |
| a <b>xnor</b> b     |                               |                        |                     |

Note: that shift is not defined for std\_logic\_vector. Use slicing and concatenation.



# Type conversion

- Type conversion is crucial in strongly typed language, such as VHDL
- Type conversion function in std\_logic\_1164 package:

| function             | data type<br>of operand a | data type<br>of result |
|----------------------|---------------------------|------------------------|
| to_bit(a)            | std_logic                 | bit                    |
| to_stdulogic(a)      | bit                       | std_logic              |
| to_bit_vector(a)     | std_logic_vector          | bit_vector             |
| to_stdlogicvector(a) | bit_vector                | std_logic_vector       |

# Examples of type conversions

## ■ E.g.

```
signal s1, s2, s3: std_logic_vector(7 downto 0);
signal b1, b2: bit_vector(7 downto 0);
```

The following statements are wrong because of data type mismatch:

```
s1 <= b1; -- bit_vector assigned to std_logic_vector
b2 <= s1 and s2; -- std_logic_vector assigned to bit_vector
s3 <= b1 or s2; -- or is undefined between bit_vector
 -- and std_logic_vector
```

We can use the conversion functions to correct these problems:

```
s1 <= to_stdlogicvector(b1);
b2 <= to_bitvector(s1 and s2);
s3 <= to_stdlogicvector(b1 or s2);
```

The last statement can also be written as:

```
s3 <= to_stdlogicvector(b1 or to_bitvector(s2));
```

# IEEE numeric\_std package

## ■ How to infer arithmetic operators?

## ■ In standard VHDL:

```
signal a, b, sum: integer;
```

```
. . .
```

```
sum <= a + b;
```

## ■ What's wrong with integer data type?

- Negative or positive representation of the number
- Integer is typically 32-bit
  - Default range is also 32-bit, synthesis tools may not optimize
  - Note the range  $-(2^{31}-1)$  to  $2^{31}-1$ ,
  - i.e. 0 to  $2^{32}-1$  not supported!

## IEEE numeric\_std package (2)

- IEEE numeric\_std package: define integer as an array of elements of std\_logic
- Two new data types: unsigned, signed
- The array interpreted as an unsigned or signed binary number, respectively
  - Unsigned are represented as standard binary
  - Signed vectors are represented using two's complement

■ E.g.,

```
signal x, y: signed(15 downto 0);
```

- Need invoke package to use the data type

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

# Overloaded operators in IEEE numeric\_std package

| overloaded operator                                                  | description                | data type of operand a                                     | data type of operand b                                     | data type of result                      |
|----------------------------------------------------------------------|----------------------------|------------------------------------------------------------|------------------------------------------------------------|------------------------------------------|
| <b>abs</b> a<br>- a                                                  | absolute value<br>negation | signed                                                     |                                                            | signed                                   |
| a * b<br>a / b<br>a <b>mod</b> b<br>a <b>rem</b> b<br>a + b<br>a - b | arithmetic<br>operation    | unsigned<br>unsigned, natural<br>signed<br>signed, integer | unsigned, natural<br>unsigned<br>signed, integer<br>signed | unsigned<br>unsigned<br>signed<br>signed |
| a = b<br>a /= b<br>a < b<br>a <= b<br>a > b<br>a >= b                | relational<br>operation    | unsigned<br>unsigned, natural<br>signed<br>signed, integer | unsigned, natural<br>unsigned<br>signed, integer<br>signed | boolean<br>boolean<br>boolean<br>boolean |

## New functions in IEEE numeric\_std package

| function          | description  | data type of<br>operand a                          | data type of<br>operand b | data type of<br>result                               |
|-------------------|--------------|----------------------------------------------------|---------------------------|------------------------------------------------------|
| shift_left(a,b)   | shift left   | unsigned, signed                                   | natural                   | same as a                                            |
| shift_right(a,b)  | shift right  |                                                    |                           | Note: that these are<br>functions, not<br>operators. |
| rotate_left(a,b)  | rotate left  |                                                    |                           |                                                      |
| rotate_right(a,b) | rotate right |                                                    |                           |                                                      |
| resize(a,b)       | resize array | unsigned, signed                                   | natural                   | same as a                                            |
| std_match(a,b)    | compare '-'  | unsigned, signed<br>std_logic_vector,<br>std_logic | same as a                 | boolean                                              |
| to_integer(a)     | data type    | unsigned, signed                                   |                           | integer                                              |
| to_unsigned(a,b)  | conversion   | natural                                            | natural                   | unsigned                                             |
| to_signed(a,b)    |              | integer                                            | natural                   | signed                                               |

# Operator overloading example

- Operator overloading is a declaration of a function whose designator is an operator symbol

- Note the double quotes around the symbol

```
package body NUMERIC_STD is
. . .
-- Result subtype: UNSIGNED(MAX(L'LENGTH, --
-- R'LENGTH)-1 downto 0).
-- Result: Adds two UNSIGNED vectors that may be of
-- different lengths.
→ function "+" (L, R: UNSIGNED) return UNSIGNED is
 constant SIZE: NATURAL := MAX(L'LENGTH, R'LENGTH);
 variable L01 : UNSIGNED(SIZE-1 downto 0);
 variable R01 : UNSIGNED(SIZE-1 downto 0);
begin
 if ((L'LENGTH < 1) or (R'LENGTH < 1)) then
 return NAU; end if;

 L01 := TO_01(RESIZE(L, SIZE), 'X');
 if (L01(L01'LEFT)='X') then return L01;
 end if;

 R01 := TO_01(RESIZE(R, SIZE), 'X');
 if (R01(R01'LEFT)='X') then return R01;
 end if;

 return ADD_UNSIGNED(L01, R01, '0');
end "+";
. . .
```

```
-- this internal function computes the addition
-- of two UNSIGNED with input CARRY
-- * the two arguments are of the same length
```

```
function ADD_UNSIGNED (L, R: UNSIGNED;
 C: STD_LOGIC) return UNSIGNED is
 constant L_LEFT: INTEGER := L'LENGTH-1;
 alias XL: UNSIGNED(L_LEFT downto 0) is L;
 alias XR: UNSIGNED(L_LEFT downto 0) is R;
 variable RESULT: UNSIGNED(L_LEFT downto 0);
 variable CBIT: STD_LOGIC := C;
begin
 for I in 0 to L_LEFT loop
 RESULT(I) := CBIT xor XL(I) xor XR(I);
 CBIT := (CBIT and XL(I))
 or (CBIT and XR(I))
 or (XL(I) and XR(I));
 end loop;
 return RESULT;
end ADD_UNSIGNED;
```

# Operators over an array data type

## ■ Relational operators for array

- operands must have the same element type but their lengths may differ

## ■ Two arrays are compared *element by element, starting from the left*

- If an array has less bits, it is considered smaller if compared bits are equal (std\_logic\_vector)

## ■ All the following return true

- "011" = "011"
- "011" > "010"
- "011" > "00010"
- "0110" > "011"



# Operators over an array data type

- a = 2 bits wide, b = 3 or 4 bits wide:

| a  | b    | a > b, std_logic_vector | a > b, unsigned |
|----|------|-------------------------|-----------------|
| 11 | 000  | 1                       | 1               |
| 11 | 011  | 1                       | 0               |
| 11 | 0111 | 1                       | 0               |
| 11 | 110  | 0                       | 0               |
| 00 | 001  | 0                       | 0               |
| 00 | 000  | 0                       | 0               |

- Problems: consider std\_logic\_vector 'if a = b then'
  - If a and b have different length, the expression is always false!
  - Syntactically correct so now warning/error
  - => Use always unsigned/signed data type for values (that need to be compared)
  - std\_logic\_vector only for "general" control and ports

# Type conversion

- Std\_logic\_vector, unsigned, signed are defined as an array of element of std\_logic
- They are considered as three different data types in VHDL
- Type conversion between data types:
  - a) type conversion function
  - b) type casting (for “closely related” data types)
- Sometimes operands must be resized to same size, e.g. both to 16 bits

# Example

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
.
.
.
signal s1, s2, s3, s4, s5, s6:
 std_logic_vector(3 downto 0);

signal u1, u2, u3, u4, u6, u7:
 unsigned(3 downto 0);

signal sg: signed(3 downto 0);
```

# Example (cntd)

```
-- Ok
u3 <= u2 + u1; -- ok, both operands unsigned

u4 <= u2 + 1; -- ok, unsigned and natural operands

-- Wrong
u5 <= sg; -- type mismatch
u6 <= 5; -- type mismatch, 5 is integer/natural

-- Fixed
u5 <= unsigned(sg); -- type casting
u6 <= to_unsigned(5,4); -- use conversion function,
 -- use 4 bits to represent
 -- the value 5
```

## Example (cntd2)

-- Wrong

```
u7 <= sg + u1; -- + undefined over these types
```

-- Fixed

```
u7 <= unsigned(sg) + u1; -- ok, but be careful
```

-- Wrong

```
s3 <= u3; -- type mismatch
```

```
s4 <= 5; -- type mismatch
```

-- Fixed

```
s3 <= std_logic_vector(u3); -- type casting
```

```
s4 <= std_logic_vector(to_unsigned(5,4));
```

## Example (cntd3)

-- Wrong

```
s5 <= s2 + s1; -- + undefined
 -- over std_logic_vector
```

```
s6 <= s2 + 1; -- + undefined for
 -- std_logic_vector
```

-- Fixed

```
s5 <= std_logic_vector(unsigned(s2)
 + unsigned(s1));
```

```
s6 <= std_logic_vector(unsigned(s2) + 1);
```

# Conversion example HW

```

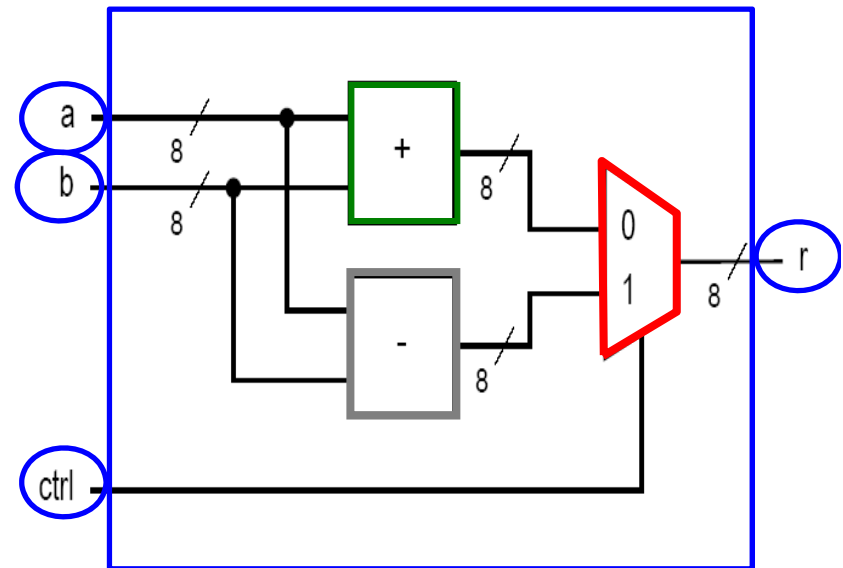
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity addsub is
 port (
 a,b: in std_logic_vector(7 downto 0);
 ctrl: in std_logic;
 r: out std_logic_vector(7 downto 0)
);
end addsub;

architecture direct_arch of addsub is
 signal src0, src1, sum:
 signed(7 downto 0);
begin
 src0 <= signed(a);
 src1 <= signed(b);
 res <= src0 + src1 when ctrl='0' else
 src0 - src1;
 r <= std_logic_vector(res);
end direct_arch;

```

| ctrl | operation   |
|------|-------------|
| 0    | $r = a + b$ |
| 1    | $r = a - b$ |



- Only a single line implements all the HW!
- No logic from the conversions!
- Exact HW for "+" and "-" operations do not have to be specified, tool will select appropriate
  - It is possible to code own implementation, also

# Resize() in numeric\_std.vhdl

```

-- RESIZE Functions

```

```
-- Id: R.1
```

```
function RESIZE (ARG: SIGNED; NEW_SIZE: NATURAL) return SIGNED;
```

```
-- Result subtype: SIGNED(NEW_SIZE-1 downto 0)
```

```
-- Result: Resizes the SIGNED vector ARG to the specified size.
```

```
-- To create a larger vector, the new [leftmost] bit positions
-- are filled with the sign bit (ARG'LEFT). When truncating,
-- the sign bit is retained along with the rightmost part.
```

```
-- Id: R.2
```

```
function RESIZE (ARG: UNSIGNED; NEW_SIZE: NATURAL) return UNSIGNED;
```

```
-- Result subtype: UNSIGNED(NEW_SIZE-1 downto 0)
```

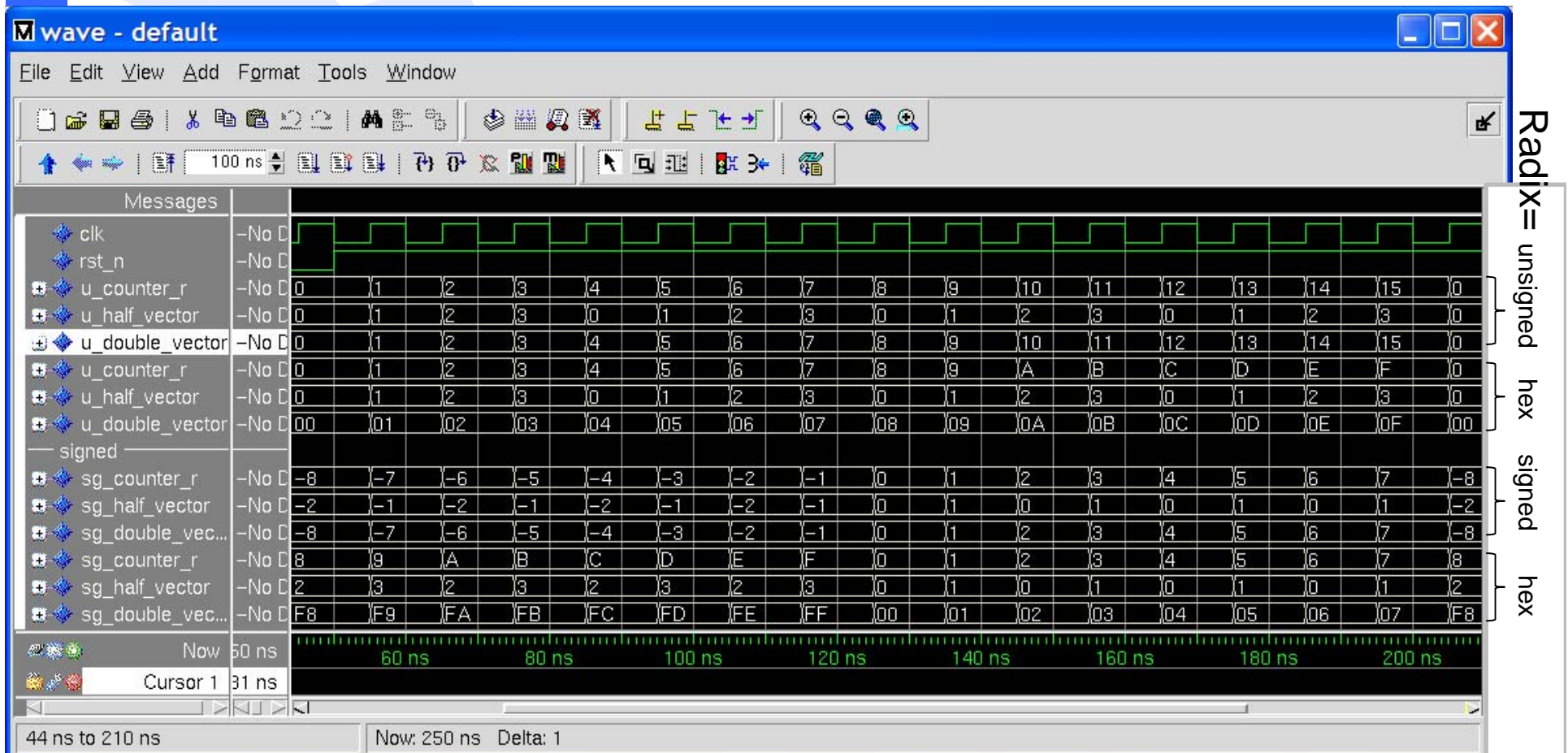
```
-- Result: Resizes the SIGNED vector ARG to the specified size.
```

```
-- To create a larger vector, the new [leftmost] bit positions
-- are filled with '0'. When truncating, the leftmost bits
-- are dropped.
```



# Resize in action

- Original data width 4b (u\_counter\_r, sg\_counter\_r) is resized to 2b and 8b



Note: showing values in hex format is bit misleading with negative numbers

# Resize() in numeric\_std.vhdl (2)

```
-- Id: R.1
function RESIZE (ARG: SIGNED; NEW_SIZE: NATURAL) return SIGNED is
 alias INVEC: SIGNED(ARG'LENGTH-1 downto 0) is ARG;
 variable RESULT: SIGNED(NEW_SIZE-1 downto 0) := (others => '0');
 constant BOUND: INTEGER := MIN(ARG'LENGTH, RESULT'LENGTH)-2;
begin
 if (NEW_SIZE < 1) then return NAS;
 end if;
 if (ARG'LENGTH = 0) then return RESULT;
 end if;
 RESULT := (others => ARG(ARG'LEFT)); -- sign extension
 if BOUND >= 0 then
 RESULT(BOUND downto 0) := INVEC(BOUND downto 0);
 end if;
 return RESULT;
end RESIZE;

-- Id: R.2
function RESIZE (ARG: UNSIGNED; NEW_SIZE: NATURAL) return UNSIGNED is
 constant ARG_LEFT: INTEGER := ARG'LENGTH-1;
 alias XARG: UNSIGNED(ARG_LEFT downto 0) is ARG;
 variable RESULT: UNSIGNED(NEW_SIZE-1 downto 0) := (others => '0');
begin
 if (NEW_SIZE < 1) then return NAU;
 end if;
 if XARG'LENGTH = 0 then return RESULT;
 end if;
 if (RESULT'LENGTH < ARG'LENGTH) then
 RESULT(RESULT'LEFT downto 0) := XARG(RESULT'LEFT downto 0);
 else
 RESULT(RESULT'LEFT downto XARG'LEFT+1) := (others => '0');
 RESULT(XARG'LEFT downto 0) := XARG;
 end if;
 return RESULT;
end RESIZE;
```

# Conversion and resize summary

| From type           | To type          | Conversion function                            |
|---------------------|------------------|------------------------------------------------|
| std_logic_vector    | unsigned         | unsigned( arg )                                |
| std_logic_vector    | signed           | signed( arg )                                  |
| unsigned            | std_logic_vector | std_logic_vector( arg )                        |
| signed              | std_logic_vector | std_logic_vector( arg )                        |
| integer             | unsigned         | to_unsigned( arg, size )                       |
| integer             | signed           | to_signed( arg, size )                         |
| unsigned            | integer          | to_integer( arg )                              |
| signed              | integer          | to_integer( arg )                              |
| integer             | std_logic_vector | integer -> unsigned/signed -> std_logic_vector |
| std_logic_vector    | integer          | std_logic_vector -> unsigned/signed -> integer |
| unsigned + unsigned | std_logic_vector | std_logic_vector( arg1 + arg2 )                |
| signed + signed     | std_logic_vector | std_logic_vector( arg1 + arg2 )                |
| Type                |                  | Resize function                                |
| unsigned            |                  | resize( arg, size )                            |
| signed              |                  | resize( arg, size )                            |

<http://www.tkt.cs.tut.fi/kurssit/50200/K14/Harjoitukset/conversion.html>



# Non-IEEE package

- Several supported non-IEEE packages exists
- Packages by Synopsys
- `std_logic_arith`:
  - Similarities with `numeric_std`
    - Cannot be used at the same time
  - New data types: unsigned, signed
  - Details are different
- `std_logic_unsigned` / `std_logic_signed`
  - Treat `std_logic_vector` as unsigned and signed numbers
  - i.e., overload `std_logic_vector` with arith operations
- USE NUMERIC\_STD
  - It is the standard and implementation is known and portable.



## 2g. Attributes



# Attributes

- A special identifier used to return or specify information about a named object.
  - Denote values, functions, types, signals, or ranges associated with various kinds of elements.
  - Predefined (a part of the VHDL'87) and user defined
  - Predefined attributes are always applied to a prefix
  - Used instead of fixed values or constants (unless the value is known and very unlikely to be modified)
- code is easier to maintain and reuse

# Attributes (2)

## ■ Predefined attributes

## ■ Notation in the examples:

- t = scalar type or subtype,
  - e.g. integer
- a = array type,
  - e.g. std\_logic\_vector (3 downto 0)
- s = signal
  - e.g. std\_logic;

### 1. Value kind attributes, *return a constant value*

- Return an explicit value and are applied to a type or subtype
- t'left, t'right, t'high, t'low,
- a'length[(n)]

### 2. Type, *return a type*: (t'base)

### 3. Range, *return a range*: (a'range, a'reverse\_range)

# Attributes (3)

## 4. Function, *call a function that returns a value:*

- Attributes that return information about a given type, signal, or array value
- $t'pos(x)$ ,  $t'val(x)$ ,  $t'succ(x)$ ,  $t'pred(x)$ ,  $t'leftof(x)$ ,  $t'rightof(x)$ ,
- $a'left[(n)]$ ,  $a'right[(n)]$ ,  $a'high[(n)]$ ,  $a'low[(n)]$ ,
- $s'event$ ,  $s'active$ ,  $s'last\_event$ ,  $s'last\_active$ ,  $s'last\_value$

## 5. Signal, *create a new implicit signal:*

- $s'delayed[(t)]$ ,  $s'stable[(t)]$ ,  $s'quiet[(t)]$ ,  $s'transaction$

### ■ User defined

- Only constants possible

### ■ Only a few are commonly needed



# Attributes: Event

- Returns value true if an event occurred (signal has changed its value) during the current delta, and otherwise returns value false.
- General form:

`S' EVENT`

- Example:

```
PROCESS(clk)
BEGIN
 IF clk'EVENT AND clk='1' THEN
 q <= d;
 END IF;
END PROCESS;
```

- NOTE:  
Typical way to model flip-flop behaviour.

- Use only for clock signal
- Can be used in synthesis
- Cannot be nested!

# Attributes: low

- Returns the lower bound of array object or type.
- General form:

`T'LOW` and `A'LOW [(N)]`

- Example:

```
...
VARIABLE c,b: BIT_VECTOR(5 DOWNT0 0);
...
FOR i IN c'LOW TO 5 LOOP
 c(i) := b(i); -- i goes from 0 to 5
END LOOP;
```

- `T'LOW` is value kind of attribute and `A'LOW` is function kind of attribute

# Attributes: left

- Returns the left-most element index of a given type or subtype.
- General form:

`T'LEFT`

- Example:

```
...
TYPE bit_array IS ARRAY (5 DOWNT0 1) OF BIT;
...
SIGNAL tmp_r : INTEGER;
...
tmp_r <= bit_array'LEFT;
-- tmp_r is assigned with a value of 5
```

# Array attributes

```
TYPE v41 IS ('X', '0', '1', 'Z');
```

```
TYPE v41_4by8 is ARRAY (3 downto 0, 0 to 7) of v41;
```

```
Signal s_4by8 : v41_4by8;
```

| Attribute      | Description          | Example                                              | Result               |
|----------------|----------------------|------------------------------------------------------|----------------------|
| 'LEFT          | Left bound           | s_4by8 'LEFT                                         | 3                    |
| 'RIGHT         | Right bound          | s_4by8 'RIGHT<br>s_4by8 'RIGHT(2)                    | 0<br>7               |
| 'HIGH          | Upper bound          | s_4by8 'HIGH(2)                                      | 7                    |
| 'LOW           | Lower bound          | s_4by8 'LOW(2)                                       | 0                    |
| 'RANGE         | Range                | s_4by8 'RANGE(2)<br>s_4by8 'RANGE(1)                 | 0 TO 7<br>3 DOWNT0 0 |
| 'REVERSE_RANGE | Reverse range        | s_4by8 'REVERSE_RANGE(2)<br>s_4by8 'REVERSE_RANGE(1) | 7 DOWNT0 0<br>0 TO 3 |
| 'LENGTH        | Length               | s_4by8 'LENGTH                                       | 4                    |
| 'ASCENDING     | TRUE<br>If Ascending | S_4by8 'ASCENDING(2)<br>s_4by8 'ASCENDING(1)         | TRUE<br>FALSE        |

Figure 6.37 Predefined Array Attributes.

Source: Zainalabedin Navabi, VHDL: Modular Design and Synthesis of Cores and Systems

More info about attributes in the extra section

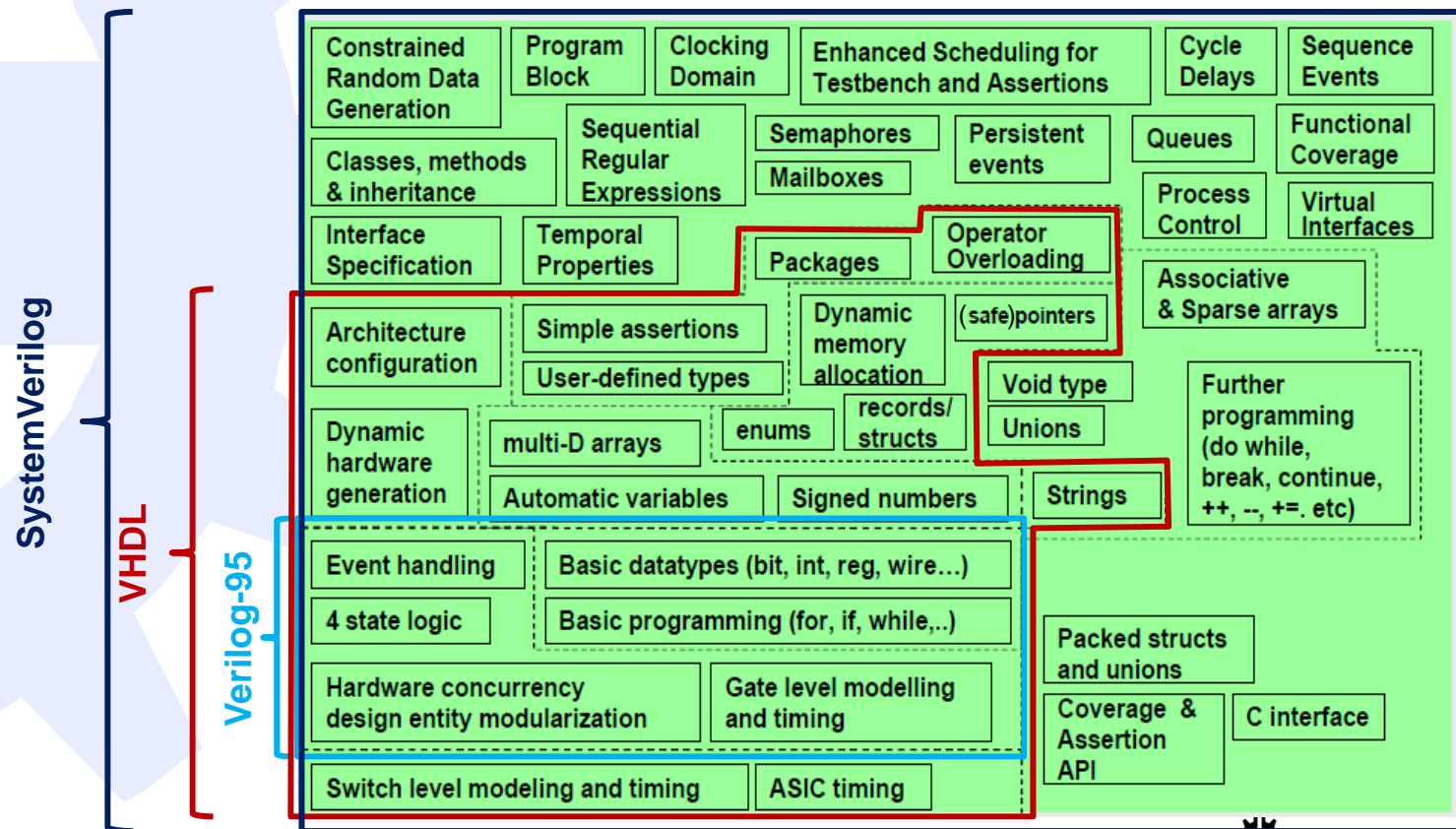
# VHDL summary

| Language constructs in VHDL | Purpose                                                                               | Other notes                                                                | C++ counterpart                          |
|-----------------------------|---------------------------------------------------------------------------------------|----------------------------------------------------------------------------|------------------------------------------|
| ENTITY                      | Defines interface. Includes generics and ports (their names, widths, and directions). | "Public interface", the actual implementation is hidden into architecture. | Class definition                         |
| GENERIC                     | Instance-specific constant value                                                      | Excellent idea in HDL!                                                     | Constant parameters, templates           |
| PORT                        | I/O pin of an entity. Defines direction and type.                                     | See also signal.                                                           | Method of a class, inter-process message |
| ARCHITECTURE                | Contains functionality.                                                               | One entity may have many architectures in the library                      | Class implementation                     |
| SIGNAL, (VARIABLE)          | Communication channel between components/processes.                                   | They are not the same! Variables only inside processes                     | Variable                                 |
| COMPONENT                   | For instantiating a sub-block                                                         | Needed for hierarchy.                                                      | Class instance, object                   |
| PROCESS                     | These capture most of the functionality.                                              | Processes are executed in parallel. Both seq. and comb.                    | Thread                                   |
| IF, FOR, CASE, ASSIGNMENT   | Control statements                                                                    | Bounds must be known for loops at compile-time                             | The same                                 |
| PACKAGE                     | Contains shared definitions.                                                          | Constants, functions, procedures, types                                    | Header file (file.h)                     |
| LIBRARY                     | Holds analyzed ('compiled') codes                                                     | Standard ieee library is practically always used                           | Compiled object codes (file.o)           |



# Capabilities Verilog-95, VHDL, SystemVerilog

- Verilog is another, but primitive HDL
  - For some reason, more popular is US and Asia than Europe
- SystemVerilog is rather new language which adds many handy features for verification



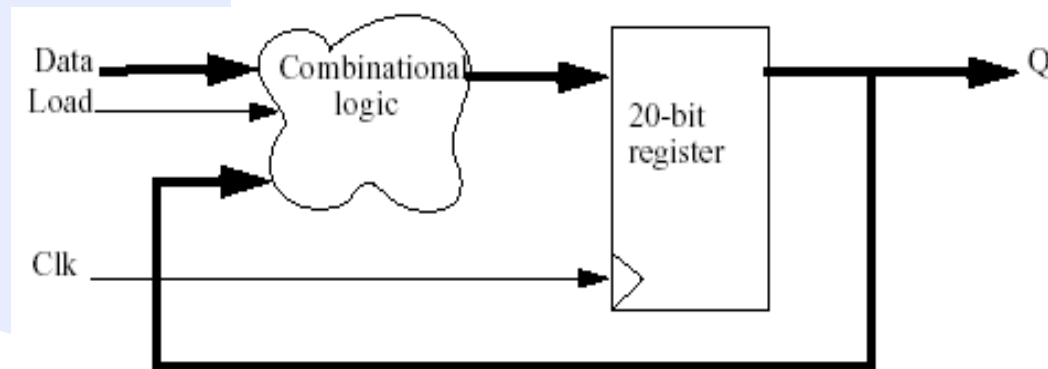


# Extra slides on VHDL



# Recap: Register transfer level (RTL)

- Circuit is described in terms of how data moves through the system.
- In the register-transfer level you describe how information flows between registers in the system.
- The combinational logic is described at a relatively high level, the placement and operation of registers is specified quite precisely.



- The behaviour of the system over the time is defined by registers.



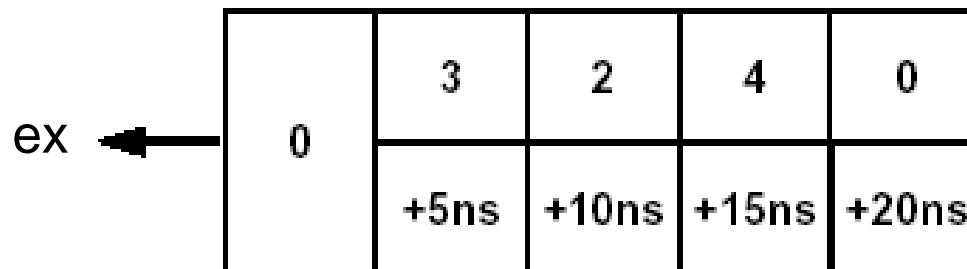
# Signal Drivers

- Every signal has at least one driver, if it is not disconnected.
- Signal assignment changes driver
- A conceptual circuit that is created for every signal driver
- Example of driver:

```
signal ex: positive;
```

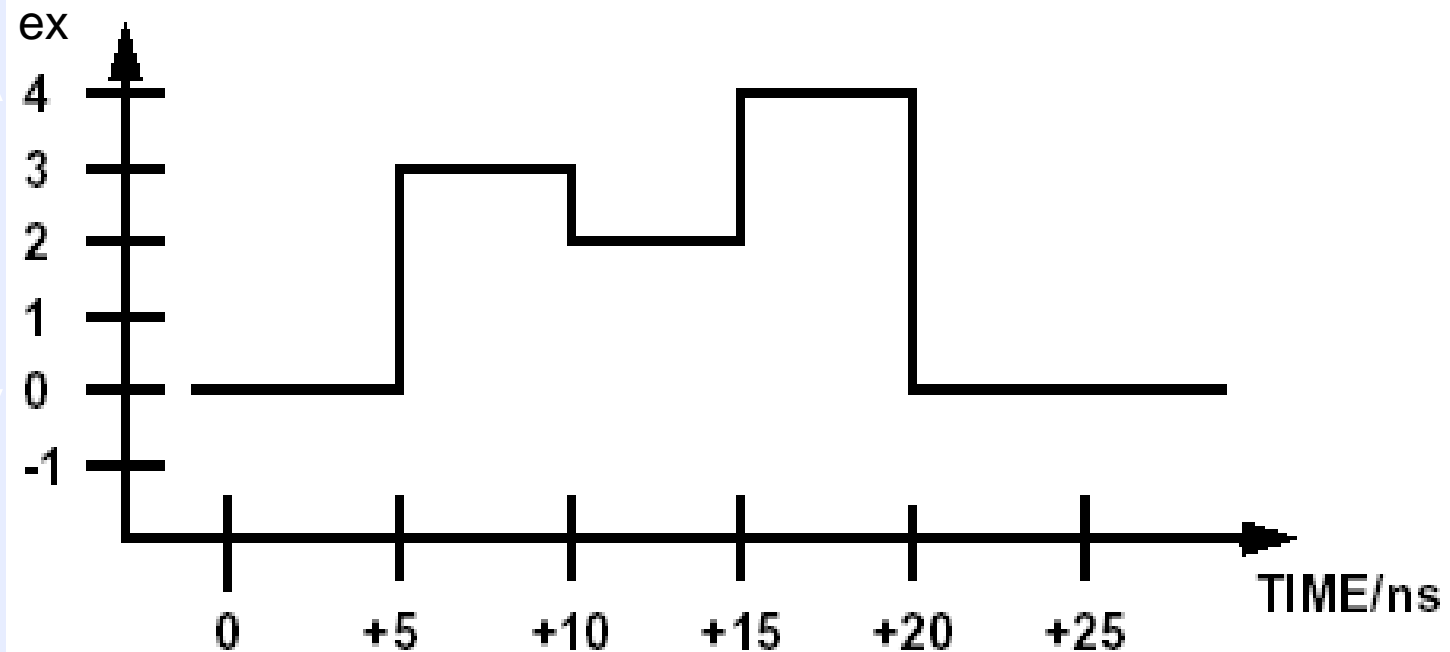
```
...
```

```
ex <= 3 AFTER 5 ns, 2 AFTER 10 ns,
 4 AFTER 15 ns, 0 AFTER 20 ns;
```



# Signal Drivers

- Signal “ex” as function of a time



# Shift...

- For bit\_vectors: operators: sla, sra, sll, srl...
- For (un)signed: functions shift\_left() and shift\_right()
- For std\_logic\_vector: no operators nor built-in functions, you should use slicing

■ **variable A: bit\_vector := "101101";**

- A sll 2 -- "110100", filled with zeros
- A sla 2 -- "110111", filled with LSB!

□ sla is rather strange operator in VHDL

- A srl 2 -- "001011", filled with zeros
- A sra 2 -- "111011", filled with MSB

grey color denotes  
inserted bits

- A rol 2 -- "110110"
- A ror 2 -- "011011"

# Configurations

- ◆ **Links entity declaration and architecture body together.**
  - ◆ **Apply to structural description (one that instantiates components)**
- ◆ **Concept of default configuration is a bit messy in VHDL '87.**
  - **Last architecture analyzed links to entity?**
- ◆ **Can be used to change simulation behaviour without re-analyzing the VHDL source.**
- ◆ **Complex configuration declarations are ignored in synthesis.**
- ◆ **Some entities can have, e.g., gate level architecture and behavioral architecture.**
- ◆ **Are always optional.**

# Configuration example

**ARCHITECTURE** configurable OF multiplexer IS

COMPONENT n2

PORT (

a: IN std\_logic;

b: IN std\_logic;

y: std\_logic

);

END COMPONENT;

SIGNAL sbar, asel, bsel : std\_logic;

**BEGIN**

U1: n2 PORT MAP (a => s, b => s, y => sbar);

U2: n2 PORT MAP (a => x, b => sbar, y => asel);

**END ARCHITECTURE** configurable;

CONFIGURATION  
is in its own section  
in VHDL file, not  
within entities or  
architectures.

Component  
"n2" used

Mapping of  
component n2

**CONFIGURATION** example\_cfg OF multiplexer IS

FOR configurable

FOR ALL : n2

USE ENTITY **WORK.nand2\_t** (arch\_2)

GENERIC MAP (cc\_delay\_g => 3);

END FOR;

END FOR;

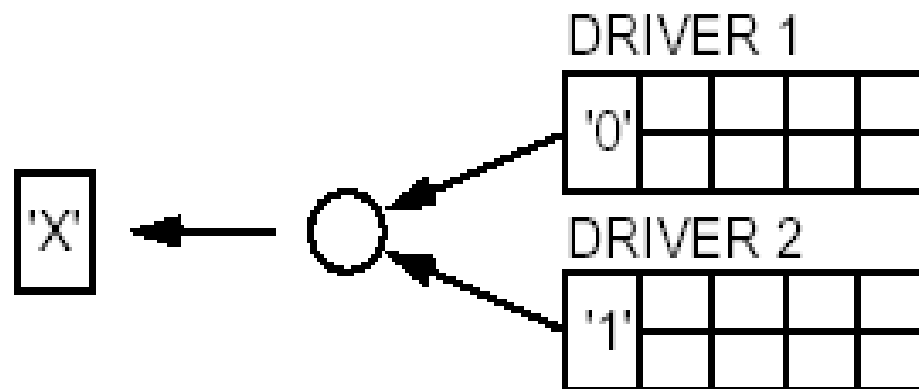
**END CONFIGURATION** example\_cfg;

Use entity  
nand2\_t with  
architecture  
arch\_2

Generic for  
the nand2\_t

# Resolution function

- Describes the resulting value when two or more different values are driven onto a signal (more than one driver exists).
- Enables resolved data types
- Multi-value logic, implementation of three-state drivers.



# Type conversion between number-related data types

| data type of a             | to data type     | conversion function / type casting |
|----------------------------|------------------|------------------------------------|
| unsigned, signed           | std_logic_vector | std_logic_vector(a)                |
| unsigned, std_logic_vector | unsigned         | unsigned(a)                        |
| unsigned, signed           | std_logic_vector | std_logic_vector(a)                |
| unsigned, signed           | integer          | to_integer(a)                      |
| natural                    | unsigned         | to_unsigned(a, size)               |
| integer                    | signed           | to_signed(a, size)                 |

# Attributes: high

- Returns the upper bound of array object or type.
- General form:

`T'HIGH` and `A'HIGH [(N)]`

- Example:

```
...
VARIABLE c,b: BIT_VECTOR(5 DOWNT0 0);
...
FOR i IN c'HIGH DOWNT0 C'LOW LOOP
 c(i) := b(i); -- i goes from 5 to 0
END LOOP;
```

- `T'HIGH` is value kind of attribute and `A'HIGH` is function kind of attribute.



# Attributes: right

- Returns the right-most bound of a given type or subtype.
- General form:

`T'RIGHT`

- Example:

```
...
TYPE bit_array IS ARRAY (5 DOWNT0 1) OF BIT;
...
SIGNAL tmp_r : INTEGER;
...
tmp_r <= bit_array'RIGHT;
-- r is assigned with a value of 1
```

# Attributes: length

- Returns the length (number of elements) of a array.
- General form:

`A' LENGTH[ (n) ]`

- Example:

```
...
TYPE bit_array IS ARRAY (31 TO 0) OF BIT;
...
SIGNAL len : INTEGER;
...
len <= bit_array'LENGTH;
-- LEN is assigned with a value of 32
```

# Attributes: range

- Returns the range of array object or array subtype.
- General form:

`A'RANGE` and `A'REVERSE_RANGE [(N)]`

- Example:

```
...
SIGNAL c,b: std_logic_vector(5 DOWNT0 0);
...
FOR i IN c'RANGE LOOP
 c(i) <= b(i); -- i goes from 5 to 0
END LOOP;
```

- NOTE:
  - T'RANGE doesn't exist.

# Attribute examples

## ■ Signal d : std\_logic\_vector (7 downto 0)

- d'LOW = 0
- d'HIGH = 7
- d'LEFT = 7
- d'RIGHT = 0
- d'LENGTH = 8
- d'RANGE =(7 downto 0)
- d'REVERSE\_RANGE=(0 to 7)

## ■ Unfortunately, these cannot be applied to e.g a specific integer

- Only information of type integer is obtainable, not about it's instantiation
- integer'high, integer'low,
- **Not:**

```
signal d : integer range 0 to 3;
d'high, d'low; -- does not work
```

# Attributes: active

- Returns value true if an any transaction occurred during the current delta, and otherwise returns value false.
- General form:

`S' ACTIVE`

- Example:

```
PROCESS(clk)
BEGIN
 IF clk'ACTIVE AND clk='1' THEN
 q <= d;
 END IF;
END PROCESS;
```

- NOTE:  
Synthesis tools may not work correctly, use  
'EVENT instead.



# Attributes: stable

- Creates a boolean signal that is true whenever the reference signal has had no events for the time specified by the optional time expression.

- General form:

```
S'STABLE [(time)]
```

- Example:

```
PROCESS(clk)
BEGIN
 IF NOT(clk'STABLE) AND clk= '1' THEN
 q <= d;
 END IF;
END PROCESS;
```

- **NOTE:**  
Used to model flip-flop behaviour. Not so efficient as event.



# Type attributes

```
TYPE v41 IS ('X', '0', '1', 'Z');
SUBTYPE v31 IS v41 RANGE '0' TO 'Z';
SUBTYPE v21 IS v41 RANGE '0' TO '1';
TYPE opcode IS (sta, lda, add, sub,
and, nop, jmp, jsr);
```

Source: Zainalabedin Navabi, VHDL: Modular Design and Synthesis of Cores and Systems

| Attribute   | Description                              | Example                              | Result       |
|-------------|------------------------------------------|--------------------------------------|--------------|
| 'BASE       | Base of type                             | v31'BASE                             | v41          |
| 'LEFT       | Left bound of type or subtype            | v31'LEFT<br>v41'LEFT                 | '0'<br>'X'   |
| 'RIGHT      | Right bound of type or subtype           | v31'RIGHT<br>v41'RIGHT               | 'Z'<br>'Z'   |
| 'HIGH       | Upper bound of type or subtype           | INTEGER'HIGH<br>v31'HIGH             | Large<br>'Z' |
| 'LOW        | Lower bound of type or subtype           | POSITIVE'LOW<br>v41'LOW              | 1<br>'X'     |
| 'POS(V)     | Position of value V in base of type.     | v41'POS('Z')<br>v31'POS('X')         | 3<br>0       |
| 'VAL(P)     | Value at Position P in base of type.     | v41'VAL(3)<br>v31'VAL(3)             | 'Z'<br>'Z'   |
| 'SUCC(V)    | Value, after value V in base of type.    | v31'SUCC('1')                        | 'Z'          |
| 'PRED(V)    | Value, before value V in base of type.   | v31'PRED('1')                        | '0'          |
| 'LEFTOF(V)  | Value, left of value V in base of type.  | v31'LEFTOF('1')<br>v31'LEFTOF('X')   | '0'<br>Error |
| 'RIGHTOF(V) | Value, right of value V in base of type. | v31'RIGHTOF('1')<br>v31'RIGHTOF('X') | 'Z'<br>'0'   |
| 'ASCENDING  | TRUE if range is ascending               | v41'ASCENDING                        | TRUE         |
| 'IMAGE (V)  | Converts value V of type to string.      | v41'IMAGE('Z')<br>opcode'IMAGE(lda)  | "Z"<br>"lda" |
| 'VALUE(S)   | Converts string S to value of type.      | opcode'VALUE("nop")                  | nop          |

Figure 6.38 Predefined Type Attributes.

# Block Statement

- ◆ **Partitioning mechanism that allows design to group logically areas of design**

```
ARCHITECTURE behav OF cpu IS
```

```
BEGIN
```

```
 alu : BLOCK
```

```
 BEGIN
```

```
 statements
```

```
 END BLOCK alu;
```

```
 regs : BLOCK
```

```
 BEGIN
```

```
 statements
```

```
 END BLOCK regs;
```

```
END behav;
```



# Guarded Block

- ◆ **A block containing boolean expression which can enable and disable driver inside block**

```
ARCHITECTURE behav OF guarded_latch IS
BEGIN
 latch : BLOCK(clk = '1')
 BEGIN
 q <= GUARDED d AFTER 3ns;
 qn <= GUARDED NOT(d) AFTER 5;
 END BLOCK latch;
END guarded_latch
```

- ◆ **Not synthesizable.**



# Guarded Signal Assignment

- Like sequential signal assignment except guarded expression
- General form:

```
target <= [guarded] [transport] expr [after t_expr{,
 expr after t_expr}
];
```

- Example, guarded signal assignment:

```
BLOCK (enable = '1')
 q <= GUARDED d AFTER 10 ns;
END BLOCK
```

- When enable = '1', d is assigned to q after 10 ns, otherwise q and d are disconnected
  - I.e. changes in d are not reflected in q

