SYSTEM MODELLING AND SYNTHESIS WITH HDL

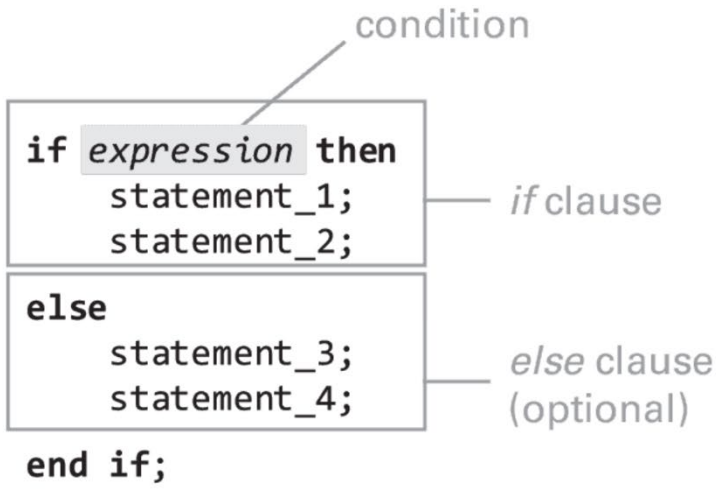
DTEK0078

2022 Lecture 5

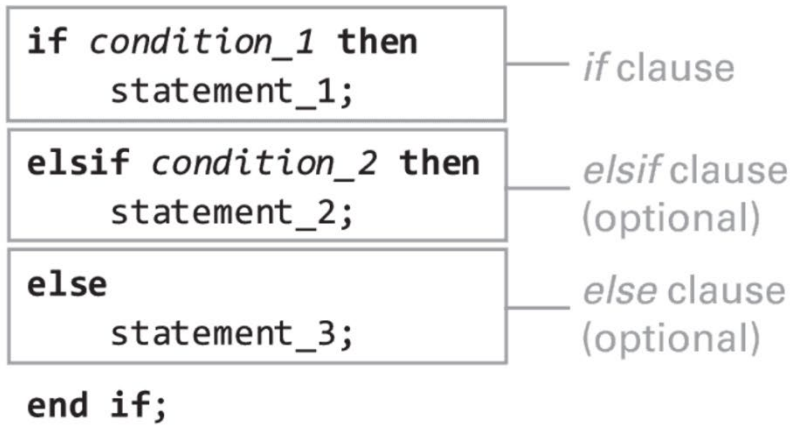
UNIVERSITY OF TURKU

If Statement

- Chooses one of the given statements
- Chooses the first matching condition for execution
- If none of the conditions are met, the construct is terminated and not statements are executed



(a) A simple *if-else* statement.

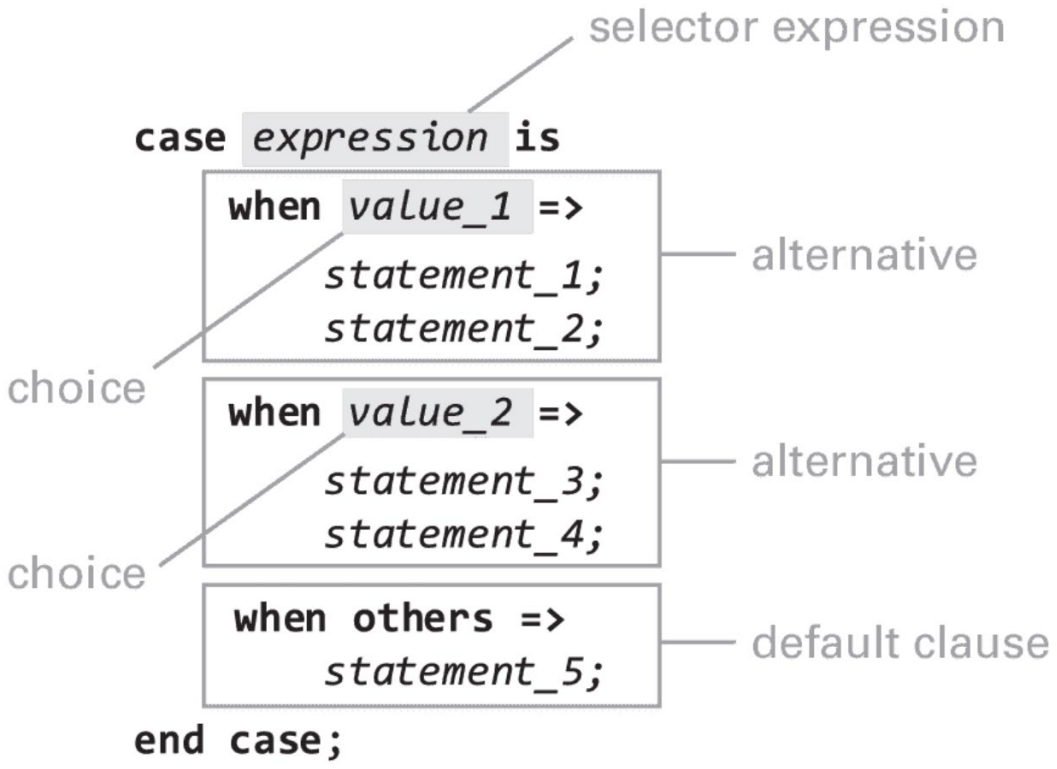


(b) An *if-elsif-else* statement.

UNIVERSITY OF TURKU

Case Statement

- Chooses one of the given statements
- *One and only one* choice can and must match
- Choices must be *mutually exclusive,* that is, must cover ALL possible choices
- *Others* can be used to denote unspecified choices



(a) Elements of a *case* statement.

UNIVERSITY OF TURKU

Preventing Latches

```
process (all) begin
    if sel = "01" then
        output <= mem(0);
    elsif sel = "10" then
        output <= mem(1);
     end if;
end process;
```

Loops

Loop Statement

- Repeatedly executes a sequence of sequential statements
- Continues until one of the following happens:
  - Completion or the iteration scheme or the execution of *exit, next* or *return* statement

```
variable i: integer := 0;
...
loop
    statements;
    i := i + 1;
    exit when i = 10;
end loop;
```

(a) A simple loop.

```
variable i: integer := 0;
...
while i < 10 loop
    statements;
    i := i + 1;
end loop;
```

(b) A *while* loop.

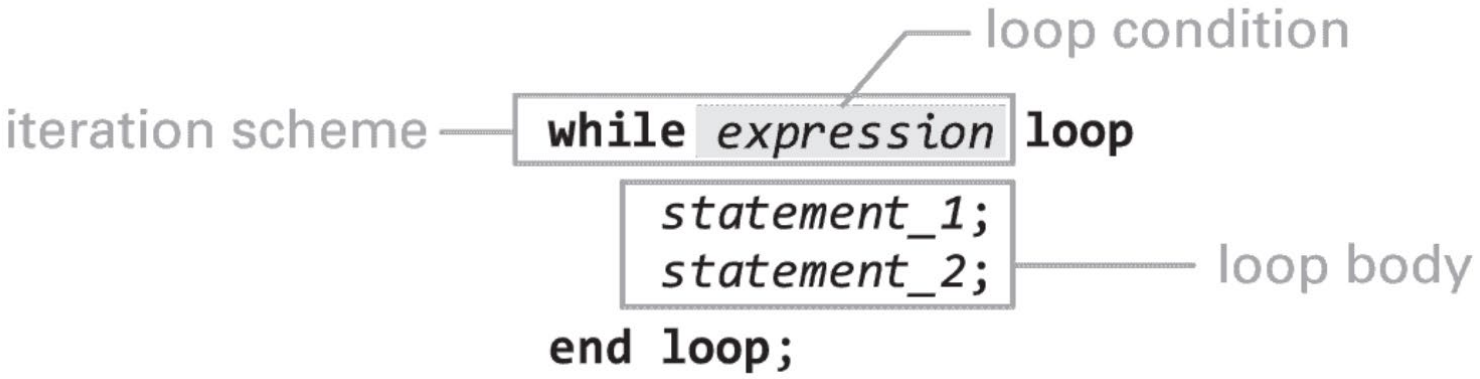
```
for i in 1 to 10 loop
    statements;
end loop;
```

(c) A *for* loop.

UNIVERSITY OF TURKU

While Loop

- Boolean iteration scheme
- The condition is evaluated before the execution of sequential statements
- One must ensure that the loop condition will eventually become false
- The execution continues with the next statement after the loop if the condition does not hold



UNIVERSITY OF TURKU

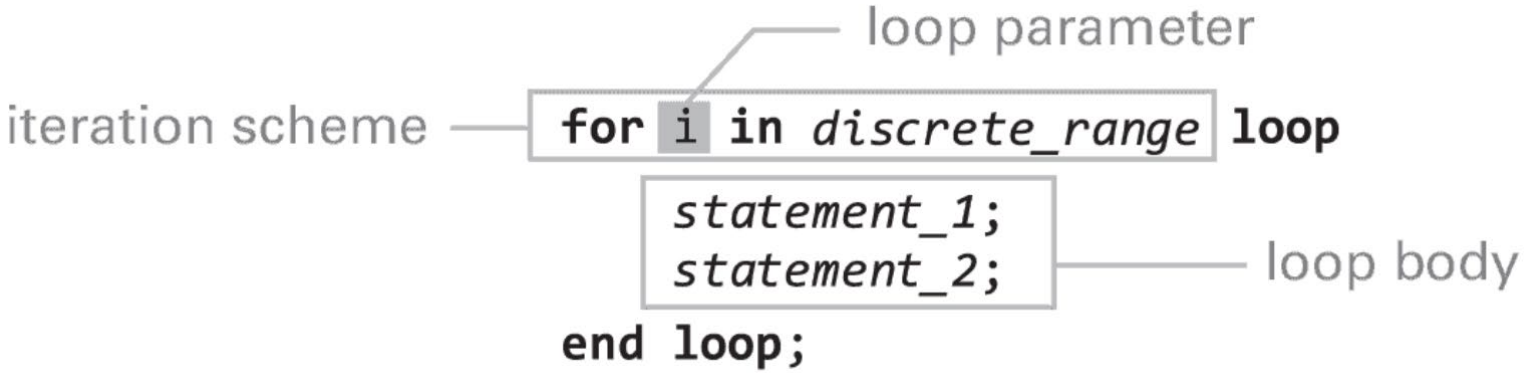
While Loop

- Boolean iteration scheme
- The condition is evaluated before the execution of sequential statements
- One must ensure that the loop condition will eventually become false
- The execution continues with the next statement after the loop if the condition does not hold

```
while loop_index > 0 loop
  sequential statements;
  update loop_index;
end loop;
sequential statements;
```

For Loop

- *identifier* takes successive values of the discrete range in each iteration of the loop
  - The number of repetitions is determined by an integer range
  - The counting starts from the left element
- After the last value in the iteration range is reached, the loop is skipped, and execution continues with the next statement after the loop



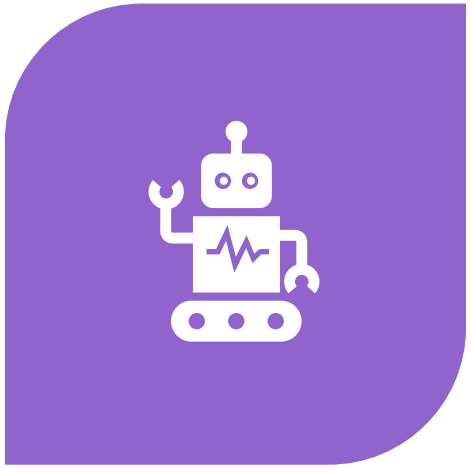
UNIVERSITY OF TURKU

For Loop

- *identifier* takes successive values of the discrete range in each iteration of the loop
  - The number of repetitions is determined by an integer range
  - The counting starts from the left element
- After the last value in the iteration range is reached, the loop is skipped, and execution continues with the next statement after the loop

```
for loop_counter in 0 to 15 loop
    counter_out <= loop_counter;
    wait for 10 ns;
end loop;
```

For loop Has Two Main Advantages



IT PUTS ALL THE CONTROL LOGIC IN ONE PLACE

—AT THE TOP

ALL THE HOUSEKEEPING IS DONE AUTOMATICALLY

Loop Parameter

- Inside the loop body, it is effectively a constant
  - it can be read but not modified
- The type of the loop parameter is determined from the discrete range specified in the loop statement
  - It is often numeric, but it does not need to be

```
type pipeline_stage_type is (fetch, decode, execute, memory, writeback);
...
for stage in pipeline_stage_type loop
    report "Current stage: " & to_string(stage);
    ...
end loop;
```

Loop Parameter

- Inside the loop body, it is effectively a constant
  - it can be read but not modified
- The type of the loop parameter is determined from the discrete range specified in the loop statement
  - It is often numeric, but it does not need to be

```
for i in stored_values'range loop
    if stored_values(i) = search_value then
        match_found := true;
        match_index := i;
        exit;
    end if;
end loop;
```

Infinite Loop

- NO *while* or *for* iteration scheme
- Enclosed statements are executed repeatedly forever until either of the auxiliary loop control statements *exit* or *next* is encountered

- Infinite loops are NOT synthesisable

Auxiliary Loop Control Statements

- Any of the three kinds of loop may use auxiliary control statements to override the normal flow of control
- Brings flexibility but reduces readability

- The *exit* statement finishes the loop immediately
  - skips the remainder of the current loop and continues with the next statement after the exited loop
- The *next* statement advance to the next iteration round
  - skips the remainder of the current loop and continues with the next loop iteration

Exit Statement

```
variable instruction: instruction_type;
...
loop
    fetch(instruction);
    exit when instruction = abort;
    execute(instruction);
end loop;
```

```
for i in stored_values'range loop
    if stored_values(i) = search_value then
        match_found := true;
        match_index := i;
        exit;
    end if;
end loop;
```

Exit Statement

```
variable instruction: instruction_type;
...
loop
    fetch(instruction);
    exit when instruction = abort;
    execute(instruction);
end loop;
```

Next Statement

```
for i in entries'range loop
    -- Invalid entries shouldn't contribute to running statistics
    next when entries(i).blank;

    -- Update running statistics
    sum := sum + entries(i).value;
    count := count + 1;
end loop;
```

```
for i in stored_values'range loop
    if stored_values(i) = search_value then
        match_found := true;
        match_index := i;
        exit;
    end if;
end loop;
```

Loops in Hardware

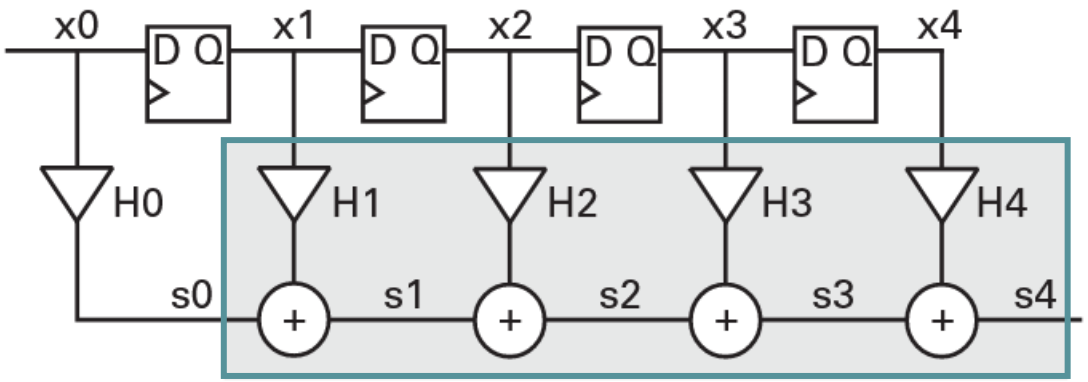
- Each iteration will originate its own set of hardware elements

```
sum := 0;
for i in 0 to 4 loop
    sum := sum + x(i);
end loop;
y <= sum;
```

(b) Implementation using a *for* loop.

Loops in Hardware

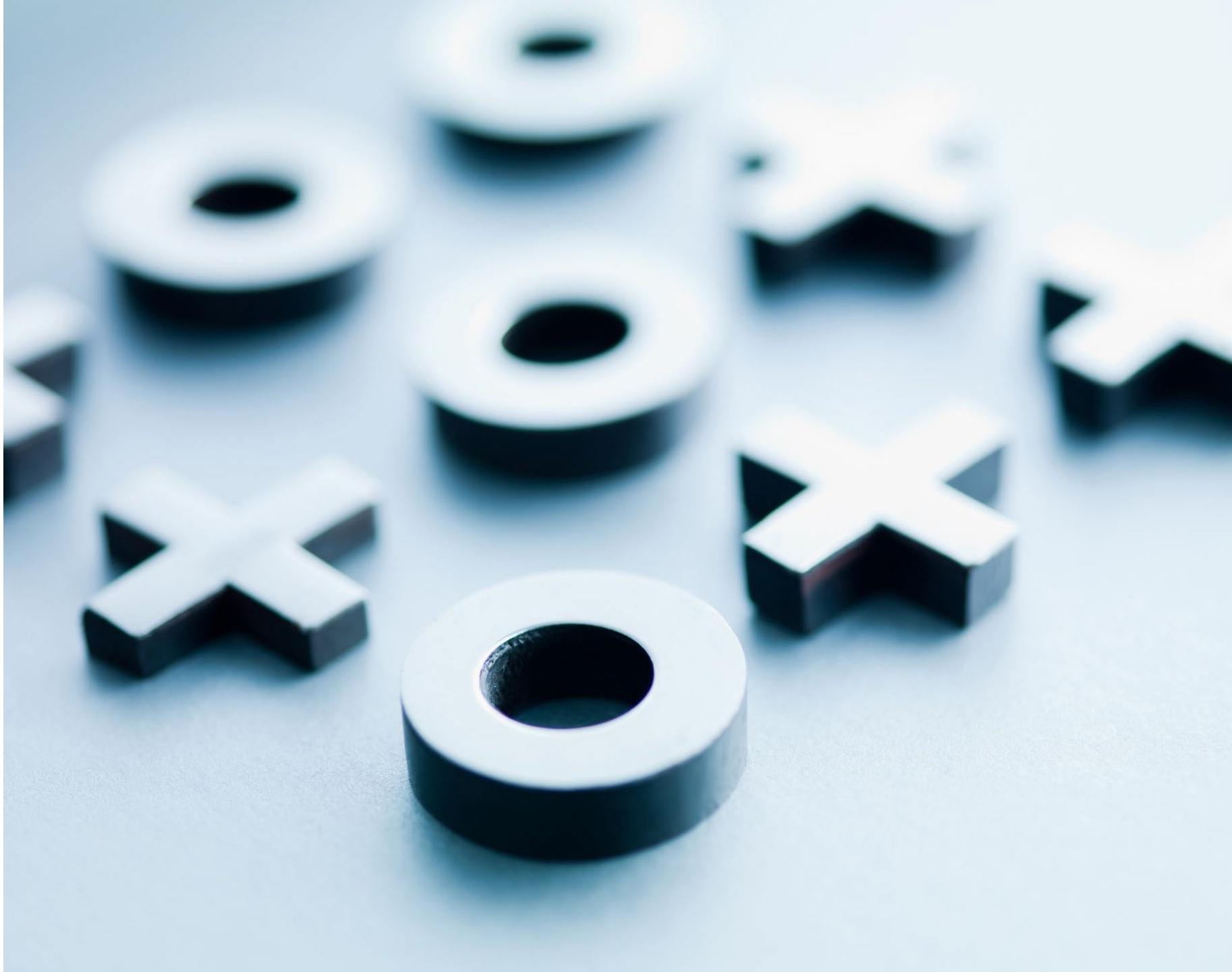
- Each iteration will originate its own set of hardware elements



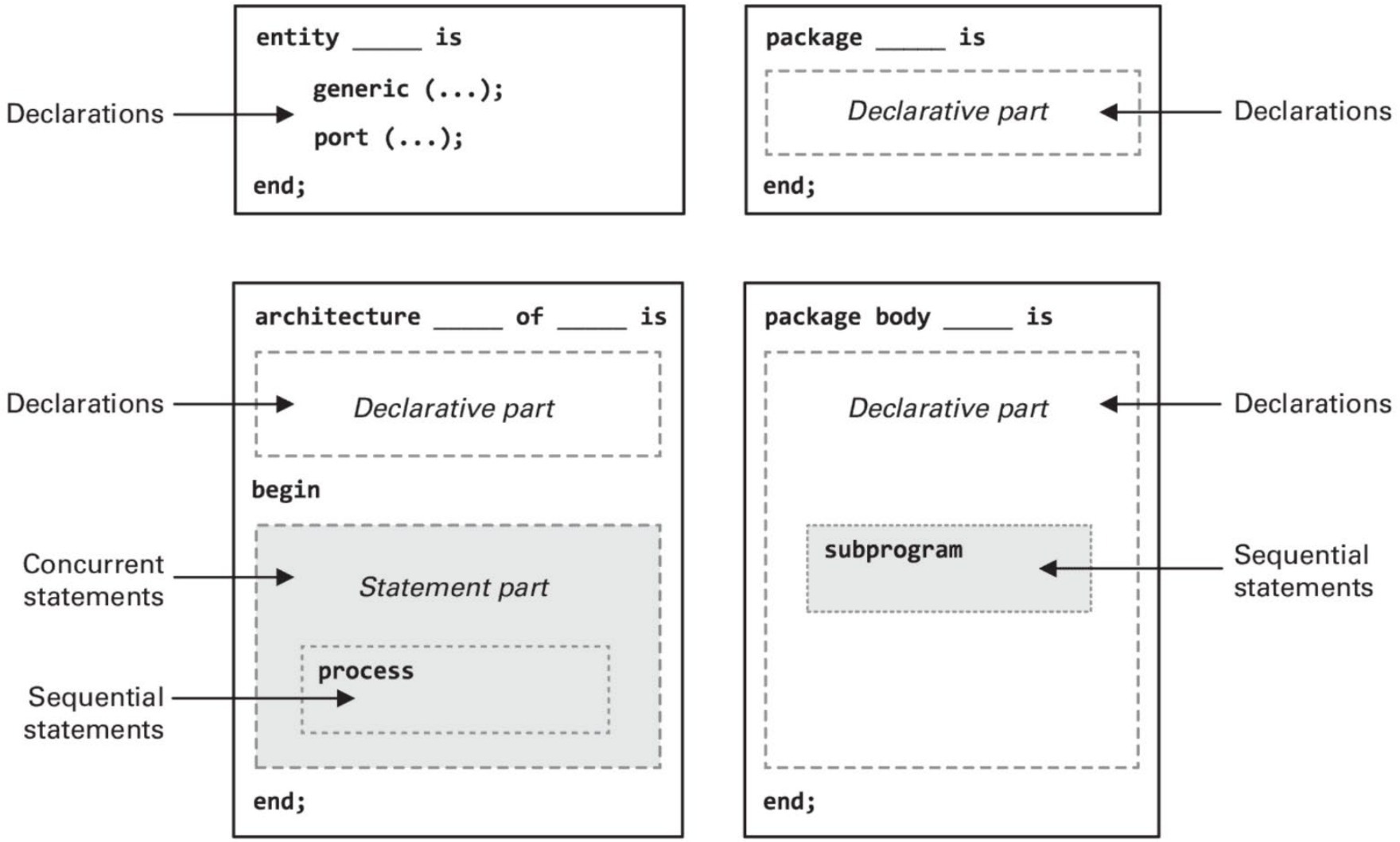

```
for i in 1 to 4 loop
    s(i) <= x(i) * H(i) + s(i-1);
end loop;
```

UNIVERSITY OF TURKU

Statements



Statements



UNIVERSITY OF TURKU

Statements

- Sequential
  - are executed one after another in the order they appear in the source code
  - can only exist in sequential code regions, namely, inside *processes* or *subprograms*
- Concurrent
  - they execute side by side, conceptually in parallel
  - they execute continuously
  - no predefined order of execution

Assert Statement

- Assertions are statements that allow VHDL code to check itself as it is simulated, compiled, or synthesized
- The assert statement checks a condition that is expected to be true

assert condition report message severity level;

Assert Statement

```
assert ( 1 <= x) and (x <= 127);
assert x > 0 severity error;

assert x /= 0 report "This should never happen";

assert fractional_value(4 downto 0) = "00000"
report "Precision loss - value will be truncated"
severity warning;
```

When to Use an Assertion?

- To Receive Precise Information When Something Goes Wrong
- To Ensure That the Code Is Working as Expected
- To Check Assumptions
- To Document the Code

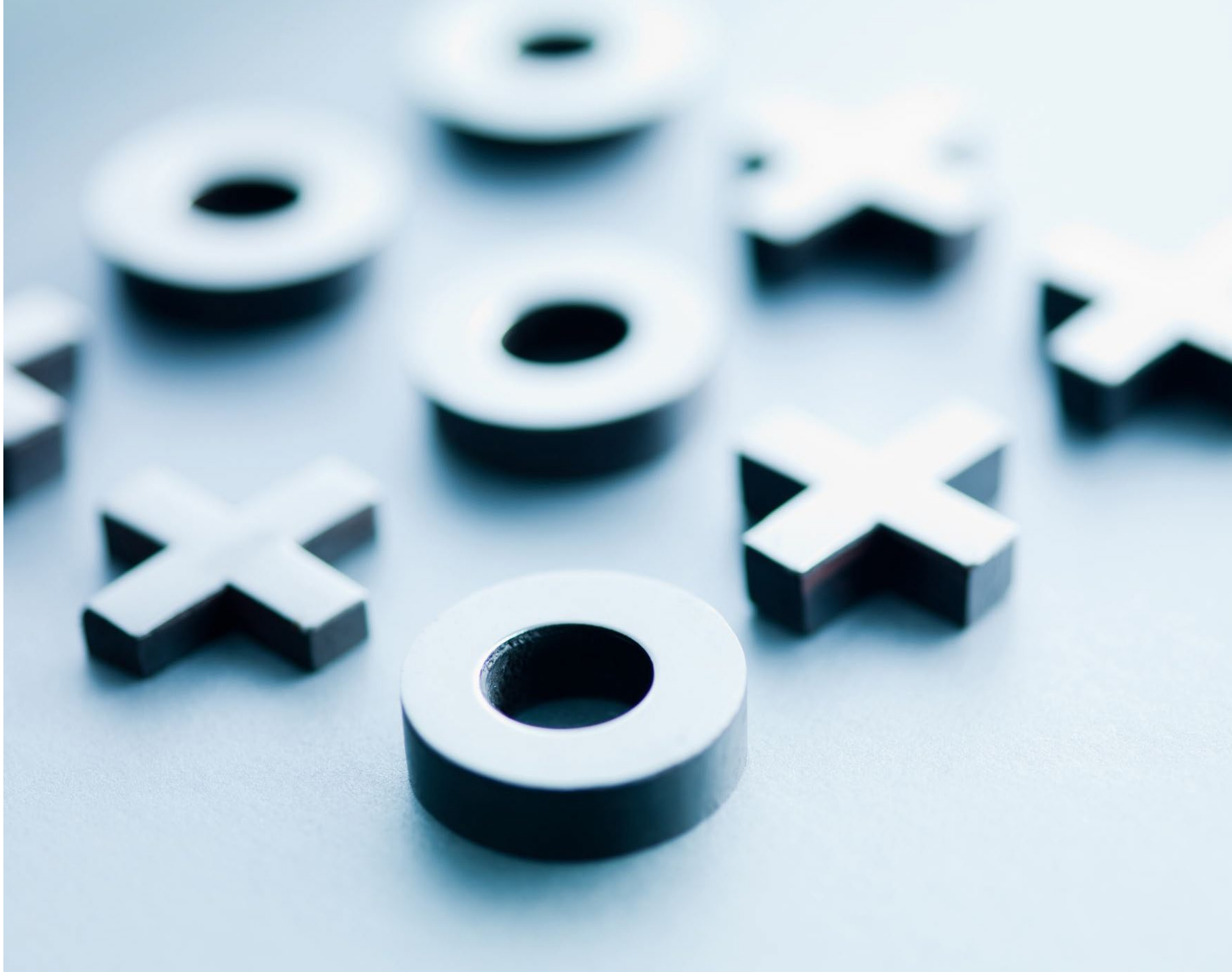
- To Detect Model Failures in Testbench Code

Severity Levels

- The VHDL standard recommends that a simulation should be stopped when an assertion of severity failure is violated
  - for any other level, the simulation should continue running

Severity Level	When to Use
note	To output debug information when there are no design problems. This level is not very useful for assertions; it is the default level for the *report* statement.
warning	To indicate unexpected conditions that *do not affect the state of the model*, but could affect its behavior. Usually means that the model can continue to run.
error	To indicate unexpected conditions that *affect the state of the model*. Usually associated with unrecoverable error conditions.
failure	To indicate a problem *inside the module* that is reporting the violation. Used in self-checking components to indicate that a bug has been detected and the code needs to be fixed.

Operators



Operators

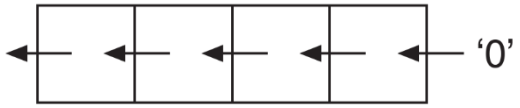
Table 8.1 Categories of operators (listed in order of decreasing precedence)

Operator Category	Operators
Miscellaneous	** abs not
	Unary version of and or nand nor xor xnor [1]
Multiplying operators	* / mod rem
Sign operators	+ -
Adding operators	+ - &
Shift operators	sll srl sla sra rol ror
Relational operators	= /= < <= > >=
	?= ?/= ?< ?<= ?> ?>=
Logical operators	Binary version of and or nand nor xor xnor
Condition operator	??

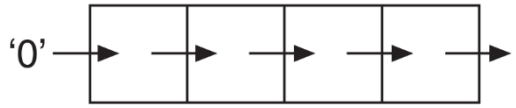
[1] In VHDL-2008, the new unary logic operators (the single-operand versions of and, or, nand, nor, xor, and xnor) have the highest precedence level).

Shift Operators

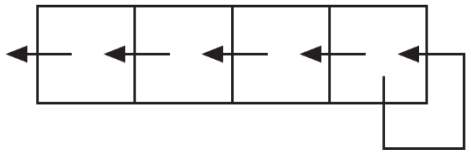
sll



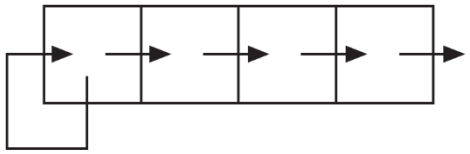
srl



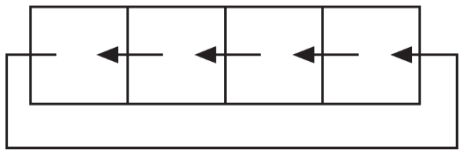
sla



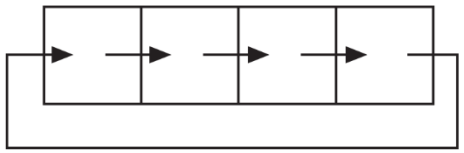
sra



rol

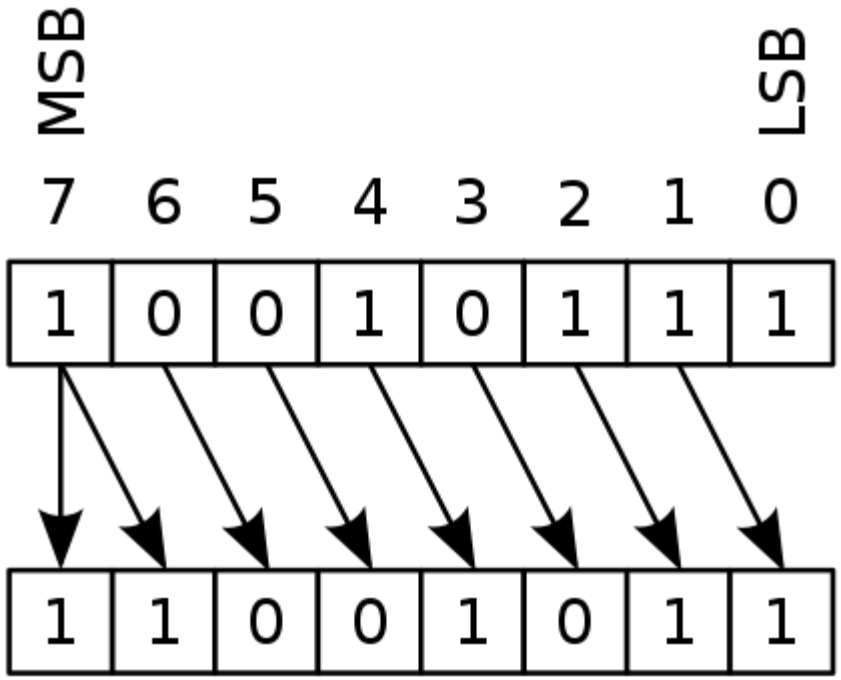


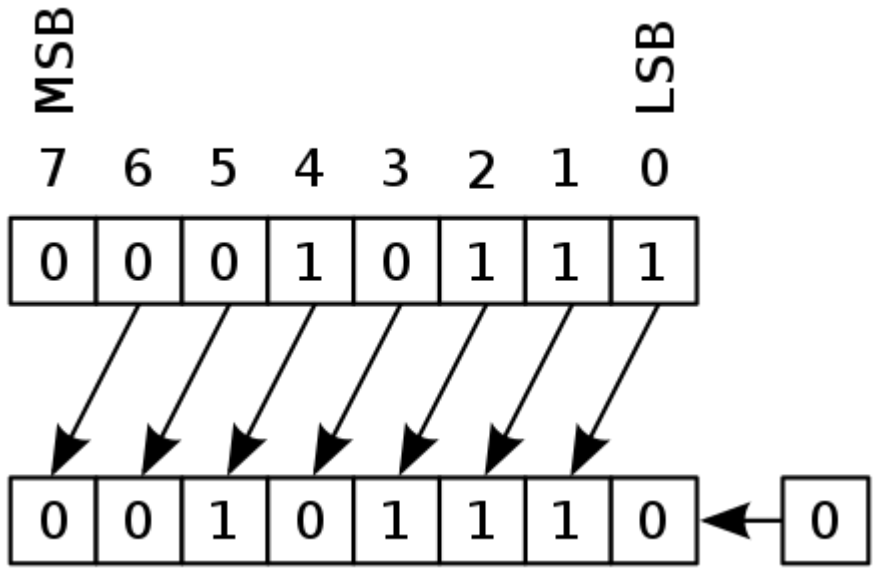
ror



UNIVERSITY OF TURKU

Shift Operators SRA and SLL





UNIVERSITY OF TURKU

Logical Operators

- Logical operators in VHDL can be classified into unary or binary
  - Operators that take a single operand are called *unary* operators
  - Operators that take two operands are called *binary* operators

```
'1' xor '0' --> '1' Use bit × bit version (called scalar-scalar operator)
'1' xor "010" --> "101" Use bit × array version (called array-scalar operator)
```

Logical Operators

```
'1' xor '0' --> '1' Use bit × bit version (called scalar-scalar operator)
'1' xor "010" --> "101" Use bit × array version (called array-scalar operator)
```

```
-- Using array × array version
"110" and "011" --> "010"
-- No left operand
and "110" --> '0'
or "000" --> '0'
xor "010" --> '1'
and "111" --> '1'
or "001" --> '1'
xor "011" --> '0'
```

Logical Operators

Operator	Left Operand	Right Operand	Result	VHDL Version
`and nand or nor xor xnor`	Single bit	Single bit	Single bit	Any version
`and nand or nor xor xnor`	Single bit	1D array	1D array	VHDL-2008
`and nand or nor xor xnor`	1D array	Single bit	1D array	VHDL-2008
`and nand or nor xor xnor`	1D array	1D array	1D array	Any version
`and nand or nor xor xnor`	none (unary)	1D array	Single bit	VHDL-2008
`not`	none (unary)	Single bit	Single bit	Any version
`not`	none (unary)	1D array	1D array	Any version

Relational Operators

- Relational operators compare two operands, testing them for equality, inequality, and relative order

Operator	Operation	Result Type	VHDL Version
=	Equality	boolean	Any version
/=	Inequality	boolean	Any version
< <= > >=	Ordering	boolean	Any version
?=	Matching equality	Operand type or array element type	VHDL-2008
?/=	Matching inequality	Operand type or array element type	VHDL-2008
?< ?<= ?> ?>=	Matching ordering	Operand type or array element type	VHDL-2008

Relational Operators

Operator	Operation	Result Type	VHDL Version
=	Equality	boolean	Any version
/=	Inequality	boolean	Any version
< <= > >=	Ordering	boolean	Any version
?=	Matching equality	Operand type or array element type	VHDL-2008
?/=	Matching inequality	Operand type or array element type	VHDL-2008
?< ?<= ?> ?>=	Matching ordering	Operand type or array element type	VHDL-2008

```
-- Match any address from 0x070000 to 0x07ffff:
-- Before VHDL-2008
ram_sel <= '1' when address(23 downto 16) = x"07" else '0';
-- After VHDL-2008
ram_sel <= (address ?= x"07----");
```

UNIVERSITY OF TURKU

Matching Equality Operator ?=

	U	X	0	1	Z	W	L	H	-
U	U	U	U	U	U	U	U	U	1
X	U	X	X	X	X	X	X	X	1
0	U	X	1	0	X	X	1	0	1
1	U	X	0	1	X	X	0	1	1
Z	U	X	X	X	X	X	X	X	1
W	U	X	X	X	X	X	X	X	1
L	U	X	1	0	X	X	1	0	1
H	U	X	0	1	X	X	0	1	1
-	1	1	1	1	1	1	1	1	1

Rules for the *matching equality* operator (?=) used with std_ulogic operands:

- A '-' matches any other symbol and returns '1'.
- A 'U' with any other symbol (except '-') returns 'U'.
- In the remaining cases, if any operand is 'X', 'W', or 'Z', the result is 'X'.
- The remaining symbols ('0', '1', 'L', 'H') are matched according to their equivalent logic levels and return a '1' or a '0'.

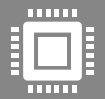
Signals

The main characteristics of signals in VHDL are

Signals are intended for interprocess communication and connections between design entities

Signals can be used in sequential or concurrent code

Signals can be declared only in the declarative part of concurrent code regions or inside packages.



Every port in an entity is a signal

Signals can be monitored for changes with events

IMPORTANT

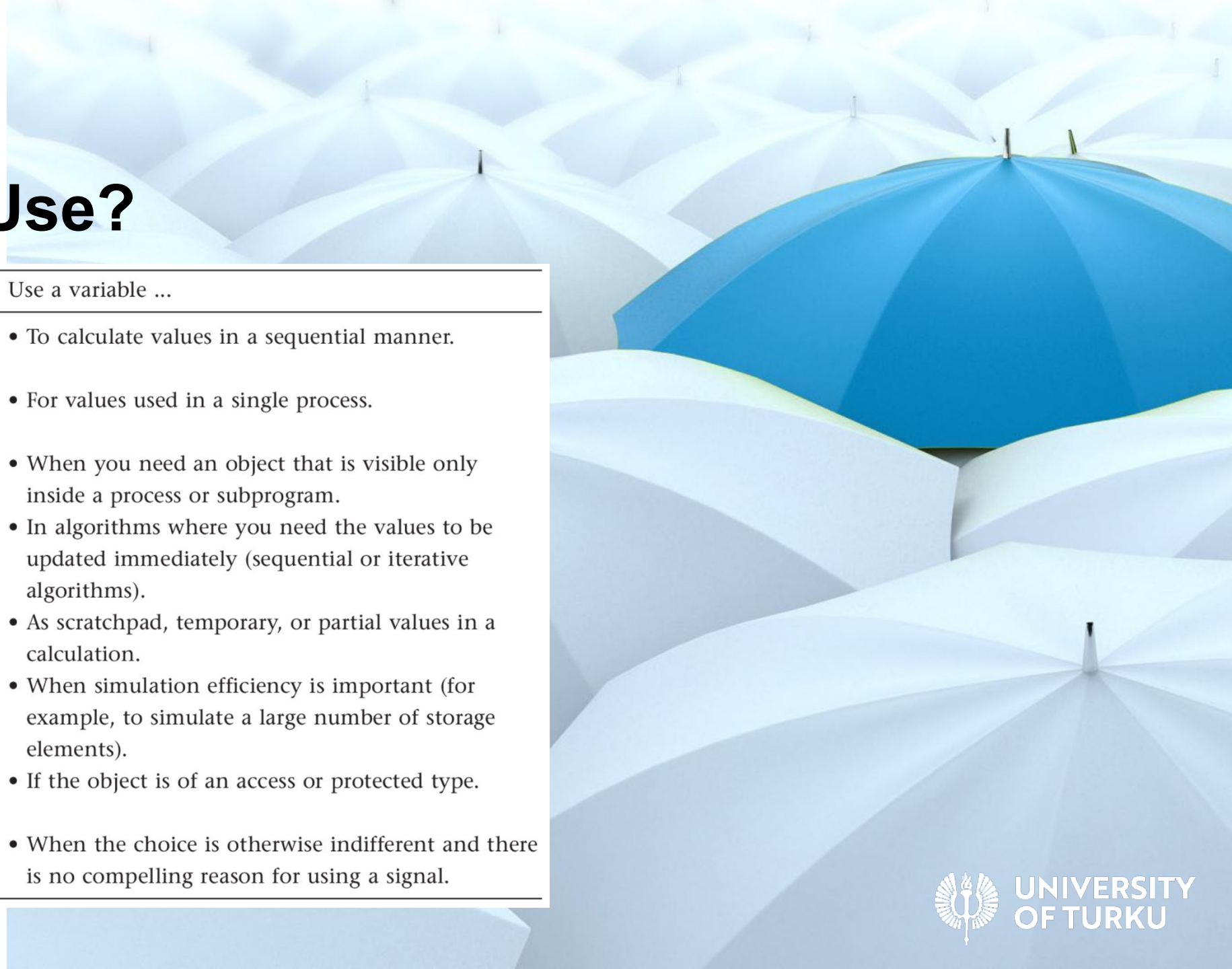
- In a process, a signal is never updated immediately on assignment
- In synthesizable code, multiple assignments to the same signal in a process behave as if only the last assignment were effective

Difference

Signal	Variable
• Intended for communication between processes or to transport values between design entities.	• Intended for local use in a process or subprogram.
• In an assignment, the new value is always scheduled for some future time (may be an infinitesimal time).	• In an assignment, the variable assumes the new value instantly.
• The value read in a process is the value that the signal had when the process resumed. Any changes are postponed until the process suspends.	• The value read in a process is the current value of the variable. Any changes can be read in the next statement.
• In synthesizable code, multiple assignments in a process behave as if only the last assignment were effective.	• In synthesizable code, the result is as if every assignment updated the variable.
• Can be used in both sequential and concurrent code.	• Can be used only in sequential code.
• Changes in the value of a signal can be monitored with events.	• Changes in the value of a variable have no observable side effects.

Which on to Use?

Use a signal ...	Use a variable ...
• To transport values between processes or between design entities.	• To calculate values in a sequential manner.
• For values that need to be used in more than one process.	• For values used in a single process.
• When you need an object that is visible in an entire architecture.	• When you need an object that is visible only inside a process or subprogram.
• For values that need to be used in concurrent statements or in concurrent code regions.	• In algorithms where you need the values to be updated immediately (sequential or iterative algorithms).
• To model the wires of a circuit diagram.	• As scratchpad, temporary, or partial values in a calculation.
• When you need to monitor an object for events.	• When simulation efficiency is important (for example, to simulate a large number of storage elements).
• To control when a process gets executed in a simulation.	• If the object is of an access or protected type.
	• When the choice is otherwise indifferent and there is no compelling reason for using a signal.



UNIVERSITY OF TURKU

Facts: Signals and Processes

- Signal values do not change while a process is running
- If a process makes multiple assignments to a signal, only the last assignment before the process suspends is effective.

```
process
begin
        integer_signal <= 1;
        integer_signal <= 2;
        integer_signal <= 3;
        if some_condition then
                integer_signal <= 4;
        end if;
        wait for 1 ms;
end process;
```

UNIVERSITY OF TURKU

Signal

- A signal is an object intended for communicating values between processes or design entities
- A signal includes the time dimension that is absent in variables: it has (past,) present, and future values

```
signal signal_1, signal_2, ... : type_name := default_value_expression;
```

Difference between Variables and Signals

- Variables are sequential statements
  - That is, they are used in processes
- Signals are concurrent statements

- **The value of a variable is updated immediately whereas the value of a signal is updated after a delay**
  - Very important to remember

Signals

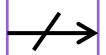
- Signals join component instances

```
entity Module_1 is
port(Dout: out std_logic);
end half_adder;
```



```
entity Module_2 is
port(Din: in std_logic);
end half_adder;
```

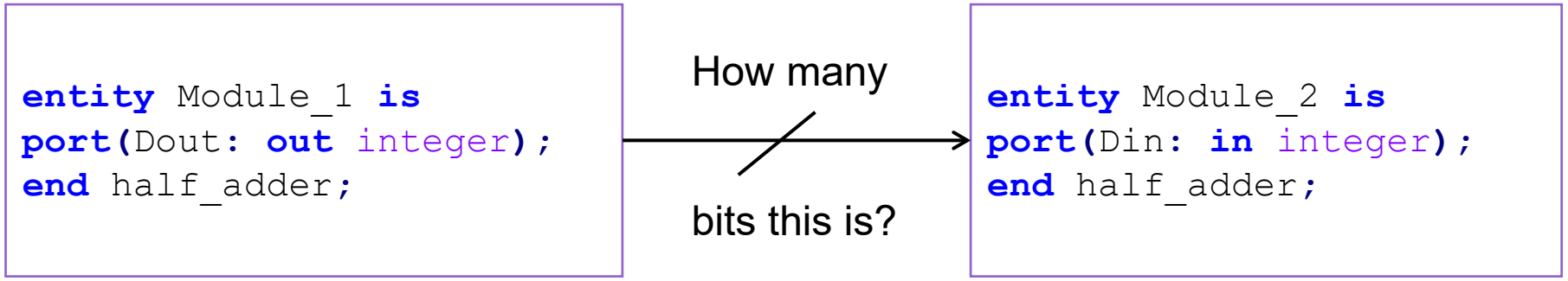
```
entity Module_1 is
port(Dout: out std_logic_vector(7 downto 0);
end half_adder;
```



```
entity Module_1 is
port(Din: in std_logic_vector(7 downto 0);
end half_adder;
```

Signals

- Signals can also be of type integer



Signals within a Module

- Signals are used within a component as well
- The *signal declaration within* an architecture defines internal signals for the module

```
architecture structural of my_module is
  signal x1 : std_logic;
  signal x2 : std_logic_vector(3 downto 0);
begin
…
end architecture;
```

Signal Types

- ~~**bit** and **bit_vector**~~
- **std_logic** and **std_logic_vector**
- **std_ulogic** and **std_ulogic_vector**
- **boolean**
- **signed** and **unsigned**

- All the above signal types are defined in libraries

Assigning Values to Signals

UNIVERSITY OF TURKU

Literals

Base Category	Subcategory	Examples
Numeric literal	Integer, decimal literal	`51, 33, 33e1`
Numeric literal	Integer, based literal	`16#33#, 2#0011_0011#`
Numeric literal	Real, decimal literal	`33.0, 33.0e1`
Numeric literal	Real, based literal	`16#33.0#, 2#0011_0011.0011#`
Numeric literal	Physical literal	`15 ns, 15.0 ns, 15.0e3 ns`
Enumeration literal	Identifier	`true, ESC, warning, failure`
Enumeration literal	Character literal	`'a', ' ', '0', '1', '½', '@', '''`
String literal	String literal	`"1100", "abc", "a", "0", "1", "123"`
Bit-string literal	Bit-string literal	`b"1100", X"7FFF", x"7fff", 16d"123"`
Null literal	Null literal	`null`

Literals

```
8d"123"          -- Expands to "01111011"
8b"111_1011"     -- Expands to "01111011"
8ub"111_1011"    -- Expands to "01111011" (unsigned, no sign extension)
8sb"111_1011"    -- Expands to "11111011" (signed, perform sign extension)
x"7b"            -- Expands to "01111011"
ux"7b"           -- Expands to "01111011" (unsigned, no sign extension)
8sx"b"           -- Expands to "11111011" (signed, perform sign extension)
```

Assigning Values to Signals

```
signal x1: std_logic;
signal x2: std_logic_vector(3 downto 0);
signal x3: std_logic_vector(15 downto 0);
signal x4: std_logic_vector(9 downto 0);

x1 <= '1';
x2 <= "0101"; -- Binary base assumed by default (B"0000")
x3 <= X"FF67"; -- Hexadecimal base
x4 <= O"713"; -- Octal base
```

Assigning Values to Signals (Vectors)

```
signal x1: std_logic;
signal x2: std_logic_vector(3 downto 0);
signal x3: std_logic_vector(15 downto 0);
signal x4: std_logic_vector(9 downto 0);

x2(0)     <= '1'; -- assigns value to the rightmost element of x2
x2        <= x3(3 downto 0); -- assigns the 4 rightmost values to x2
x2        <= (3 => '1', 0 => '1', 2 => '1', 1 => '0');
x2        <= (3|0 |2 => '1', 1 => '0');
x2        <= (others => '1', 1 => '0');
x2        <= (others => '0');
```

Assigning Values to Signals - VHDL-2008

- An array aggregate forms an array from a collection of elements

```
('0', "1001", '1')
```

- Can be used for the target of an assignment statement as well

```
signal a, b, sum : unsigned(7 downto 0);
signal carry : std_ulogic;
...
(carry, sum) <= ('0' & a) + ('0' & b);
```

```
(13 downto 8 => opcode,
 6 downto 0 =>register_address,
 7 => destination) := instruction;
```

UNIVERSITY OF TURKU

Assigning Values to Signals – VHDL 2008

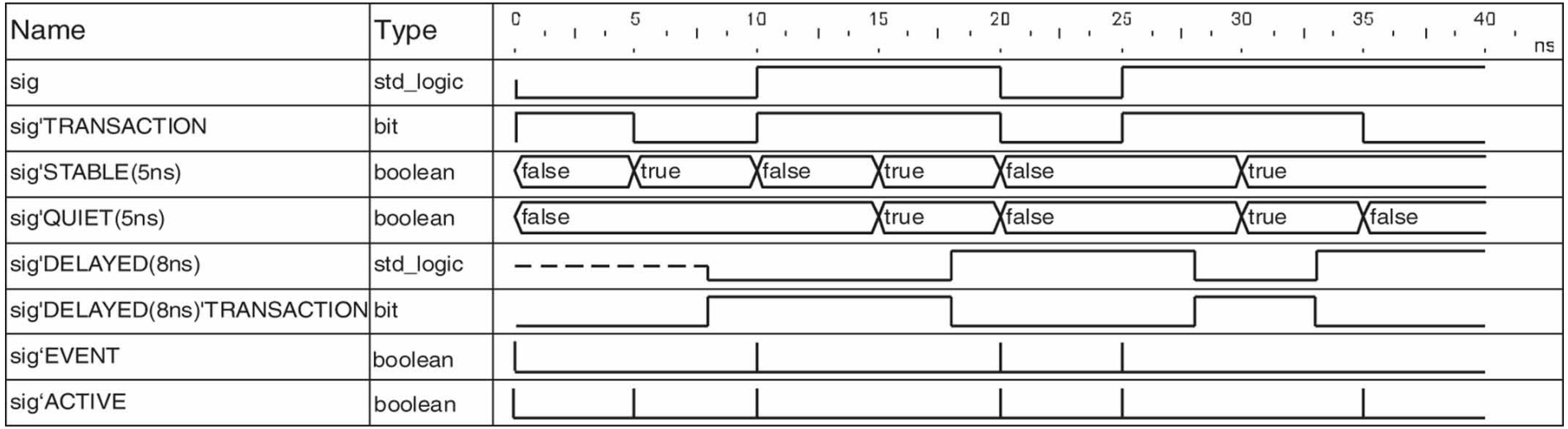
```
signal x1: std_logic_vector(5 downto 0);

x1 <= 6x"0f";
x1 <= 6SX"F";  (sign extension)
x1 <= 6Ux"f";  (zero extension)
x1 <= 6sb"11"; (binary format)
x1 <= 6uO"7";  (octal format)
```

Signal Attributes

Attribute	Result
S'Transaction	Implicit bit signal whose value is changed in each simulation cycle in which a transaction occurs on S (signal S becomes active).
S'Stable(t)	Implicit boolean signal. True when no event has occurred on S for t time units up to the current time, False otherwise.
S'Quiet(t)	Implicit boolean signal. True when no transaction has occurred on S for t time units up to the current time, False otherwise.
S'Delayed(t)	Implicit signal equivalent to S, but delayed t units of time.
S'Event	A boolean value. True if an event has occurred on S in the current simulation cycle, False otherwise.
S'Active	A boolean value. True if a transaction occurred on S in the current simulation cycle, False otherwise.
S'Last_event	Amount of elapsed time since last event on S, if no event has yet occurred it returns TIME'HIGH.
S'Last_active	Amount of time elapsed since last transaction on S, if no transaction has yet occurred it returns TIME'HIGH.
S'Last_value	Previous value of S immediately before last event on S.
S'Driving	True if the process is driving S or every element of a composite S, or False if the current value of the driver for S or any element of S in the process is determined by the null transaction.
S'Driving_value	Current value of the driver for S in the process containing the assignment statement to S.

Signal Attributes

Attribute	Result
S'Transaction	Implicit bit signal whose value is changed in each simulation cycle in which a transaction occurs on S (signal S becomes active).
S'Stable(t)	Implicit boolean signal. True when no event has occurred on S for t time units up to the current time, False otherwise.
S'Quiet(t)	Implicit boolean signal. True when no transaction has occurred on S for t time units up to the current time, False otherwise.
S'Delayed(t)	Implicit signal equivalent to S, but delayed t units of time.
S'Event	A boolean value. True if an event has occurred on S in the current simulation cycle, False otherwise.
S'Active	A boolean value. True if a transaction occurred on S in the current simulation cycle, False otherwise.

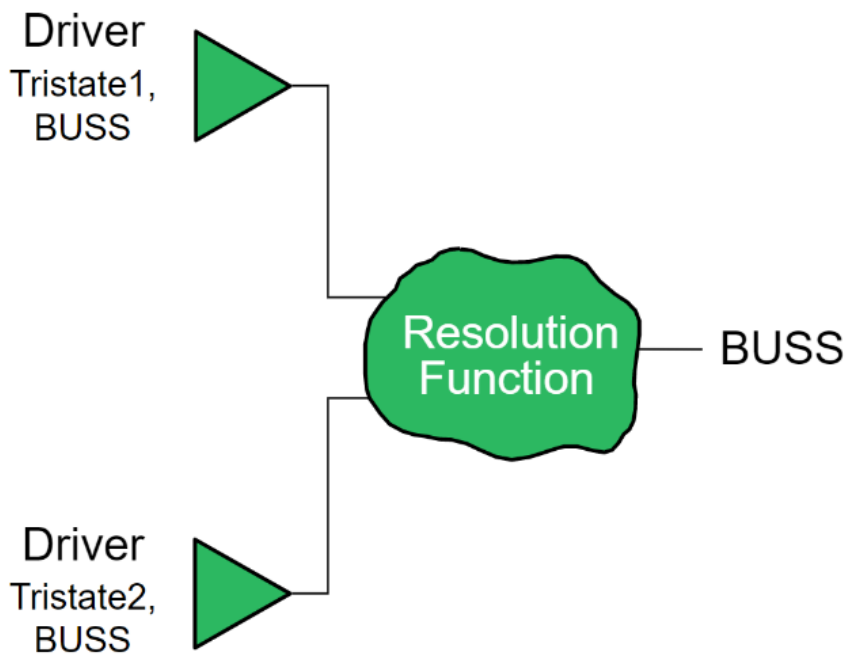


UNIVERSITY OF TURKU

Tristate Drivers

```
subtype STD_LOGIC is RESOLVED STD_ULOGIC;
```

```
signal BUSS, ENB1, ENB2, D1, D2: STD_LOGIC;
...
TRISTATE1: process (ENB1, D1)
begin
  if ENB1 = '1' then
    BUSS <= D1;
  else
    BUSS <= 'Z';
  end if;
end process;

TRISTATE2: process (ENB2, D2)
begin
  if ENB2 = '1' then
    BUSS <= D2;
  else
    BUSS <= 'Z';
  end if;
end process;
```




UNIVERSITY
OF TURKU