SYSTEM MODELLING AND SYNTHESIS WITH HDL

DTEK0078

2023 Lecture 5

UNIVERSITY OF TURKU

Processes

Processes

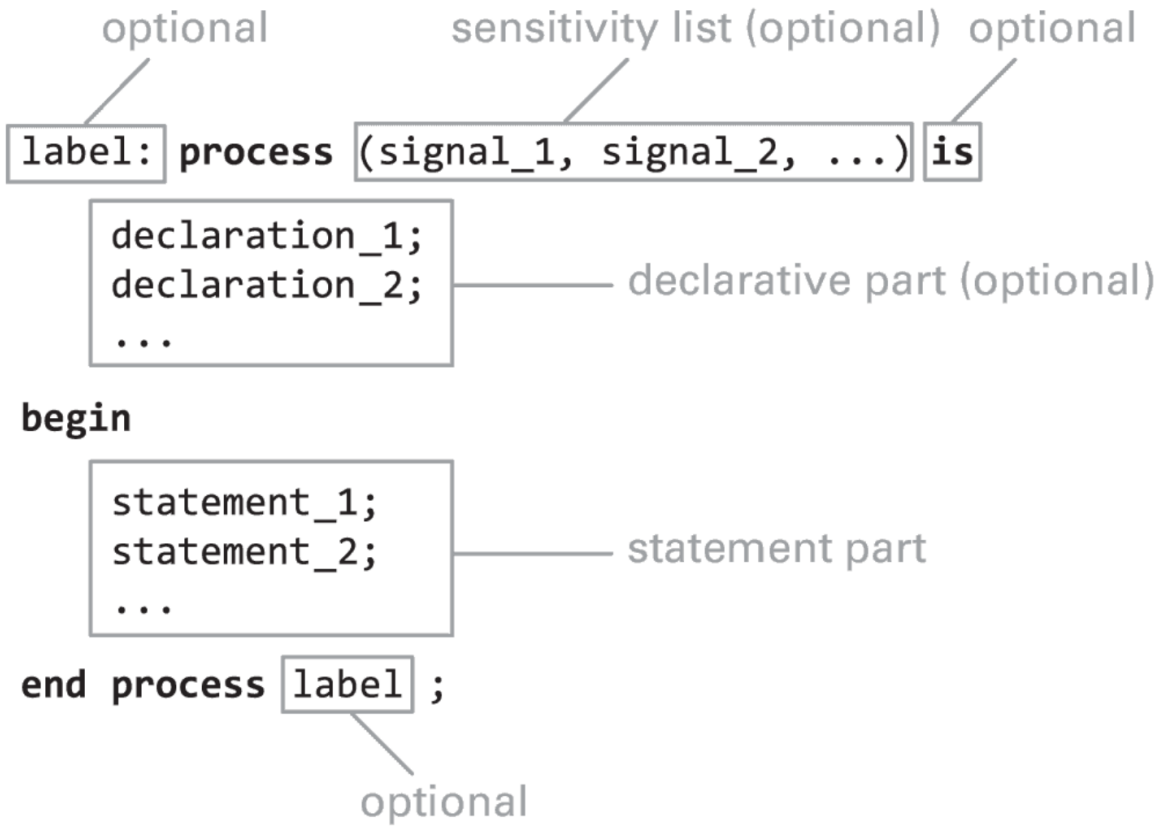
- Processes are executed

  in parallel

- Statements within a process are executed

  in sequece

- After all process's statements are executed, the process starts again from the beginning
  - Signals in the sensitity list awakes the process for executions



UNIVERSITY OF TURKU

Sensitivity List

- Two ways to write the sensitivity list for a combinational process

```
next_state_logic: process (all)
begin
    case current_state is
        when accepting_coins =>
            ...
    end case;
end process;
```

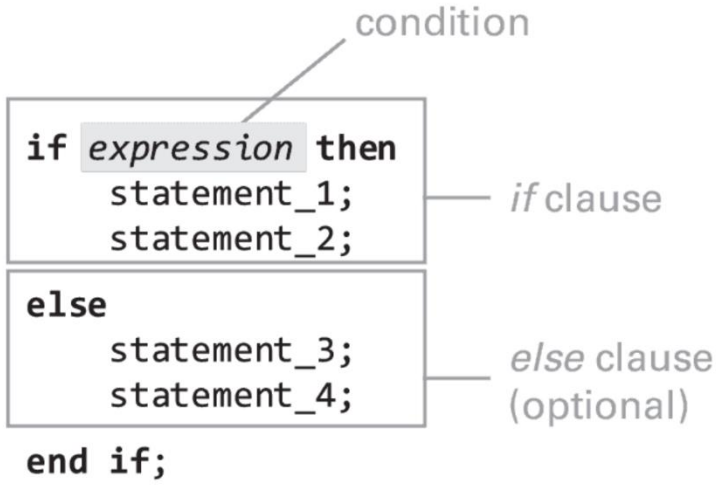
```
next_state_logic: process (current_state, nickel_in, dime_in, quarter_in, coin_return)
begin
    case current_state is
        when accepting_coins =>
            ...
    end case;
end process;
```

UNIVERSITY OF TURKU

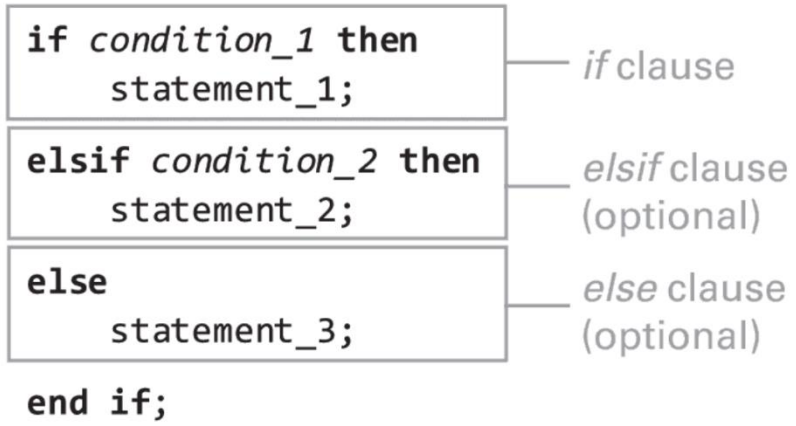
Control Structures

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

If Statement

- Choose the order of the if and else clauses consciously
- Replace complicated tests with a boolean variable

```
if (sync_counter = 63 and (data_in_sync(7 downto 0) = "01000111"
    or  data_in_sync(7 downto 0) = "00100111")
    and (current_state =  acquiring_lock or current_state = recovering_lock) )
then  locked <= '1';  ... end if;
```

If Statement

```
if (sync_counter = 63 and (data_in_sync(7 downto 0) = "01000111"
    or  data_in_sync(7 downto 0) = "00100111")
    and (current_state =  acquiring_lock or current_state = recovering_lock) )
then  locked <= '1';  ... end if;

alias current_byte is data_in_sync( 7 downto 0);
...
waiting_lock := (current_state = acquiring_lock) or (current_state = recovering_lock);
last_sync_bit := (sync_counter = 63);
preamble_detected := (current_byte = "01000111") or (current_byte = "00100111");

if waiting_lock and last_sync_bit and preamble_detected
then  locked <= '1';  .. . end if;
```

If Statement

- Replace complicated tests with a boolean variable

```
if chip_sel = '1' and enable = '1' and write = '1' then ...

-- In VHDL2008 use instead
if chip_sel and enable and write then ...
```

Conditional Signal Assignments

- Shorthand notation for the IF statement

```
conditional_signal_assignment ::=
target <= { value_expression when condition else }
          value_expression [ when condition ];
```

- Conditions are evaluated in the order of appearance
- Last values must NOT have an associated condition
- Only available in VHDL 2008 onwards
  - VHDL 2008 provides a conditional variable assignment as well (Older versions must use the if statement)

Conditional Signal Assignments

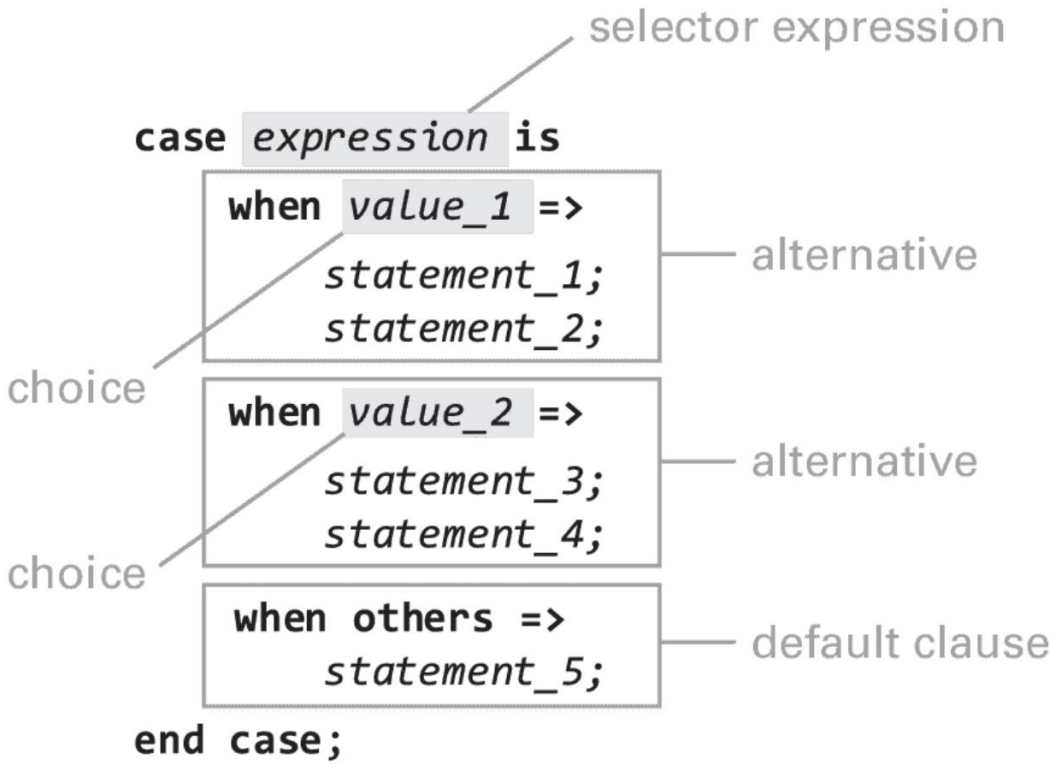
- Shorthand notation for the IF statement

```
if error_detected then
    next_state <= locking_error;
else
    next_state <= locked;
end if;
```

```
next_state <= locking_error when error_detected else locked;
```

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.

Case Statement

```
entity simple is
port (x1: in std_logic_vector(1 downto 0);
      x2: out std_logic_vector(7 downto 0) );
end;

architecture behaviour of simple is
begin
  case :Process (x1)
  begin
    case x1 is
        when ”00” => x2 <= X”01”;
        when ”01” => x2 <= X”10”;
        when ”10” => x2 <= X”11”;
        when ”11” => x2 <= X”11”;
    end case;
  end process;
end architecture;
```

UNIVERSITY OF TURKU

Case Statement

```
entity simple is
port (x1: in std_logic_vector(1 downto 0);
      x2: out std_logic_vector(7 downto 0) );
end;

architecture behaviour of simple is
begin
  case :Process (x1)
  begin
    case x1 is
        when ”00” => x2 <= X”01”;
        when ”10” => x2 <= X”10”;
        when others => x2 <= X”11”;
      end case;

  end process;
end architecture;
```

UNIVERSITY OF TURKU

Don’t Care Inputs and Outpus

- Don’t care indicates that we are not interested is the value ’1’ or ’0’
- Don’t care value is represented with the character ’-’
- NOTE, don’t care value is treated differently by simulators and synthesisers
  - Synthesisers take advantage of the don’t care value to mimise the logic it synthesises

```
x1   <= ”00” when s = ”0---” else
        ”10” when s = ”1---” else
        ”11”;

x1  <=  ”0” when s = ”1000” else
        ”1” when s = ”1100” else
        ”-”  when others;
```

UNIVERSITY OF TURKU

Matching case Statement

- case which allows don't care behaviour

```
case? data_word is
    when "1---"    =>  leading_zeros_count := 0;
    when "01--"    =>  leading_zeros_count := 1;
    when "001-"    =>  leading_zeros_count := 2;
    when "0001"    =>  leading_zeros_count := 3;
    when "0000"    =>  leading_zeros_count := 4;
    when others    =>  null;
end case?;
```

- The don’t care terms ('-') match any value at their corresponding positions. Thus, the first choice in the earlier example ("1---") would match the values "1000", "1111", "1010", and so on

Selected Signal Assignment

- Shorthand notation for the CASE statement

```
selected_signal_assignment ::=
with expression select
  target <= { value_expression when choices , }
            value_expression  when choices ;
```

- The value of *expression* is first evaluated after which the value is simultaneously compared againts all the *choices*
- Only available in VHDL 2008
  - VHDL 2008 provides a selected variable assignement as well (Older versions must use the if statement)

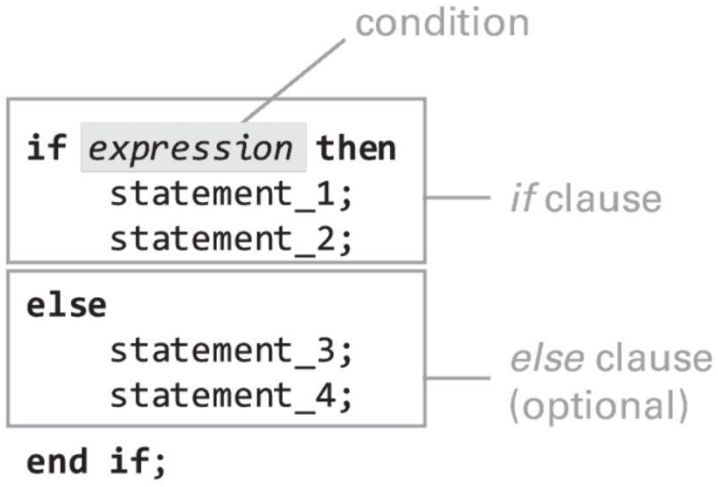
Null Statement

- Null statement explicitly indicates that there is nothing to be done
- The null statement can be used anywhere where a sequential statement is required
  - Used mainly in case statements
  - For example, in processor models, one may use *null* when *nop* (*no operation)* is performed

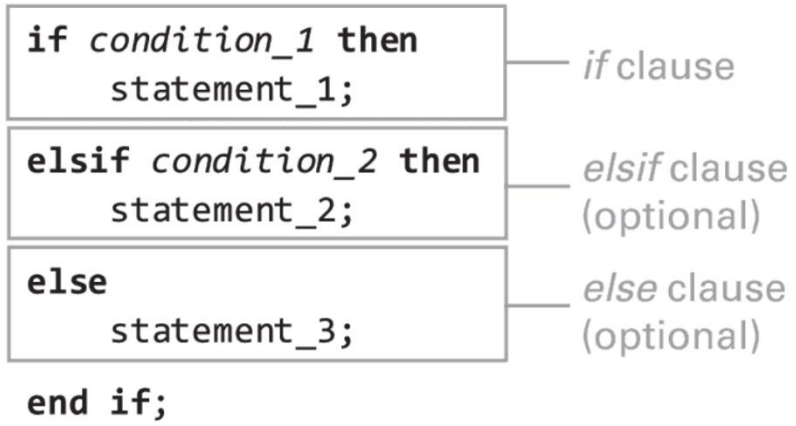
```
case x1 is
    when ”00” => x2 <= X”01”;
    when ”10” => x2 <= X”10”;
    when others => null;
end case;
```

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.

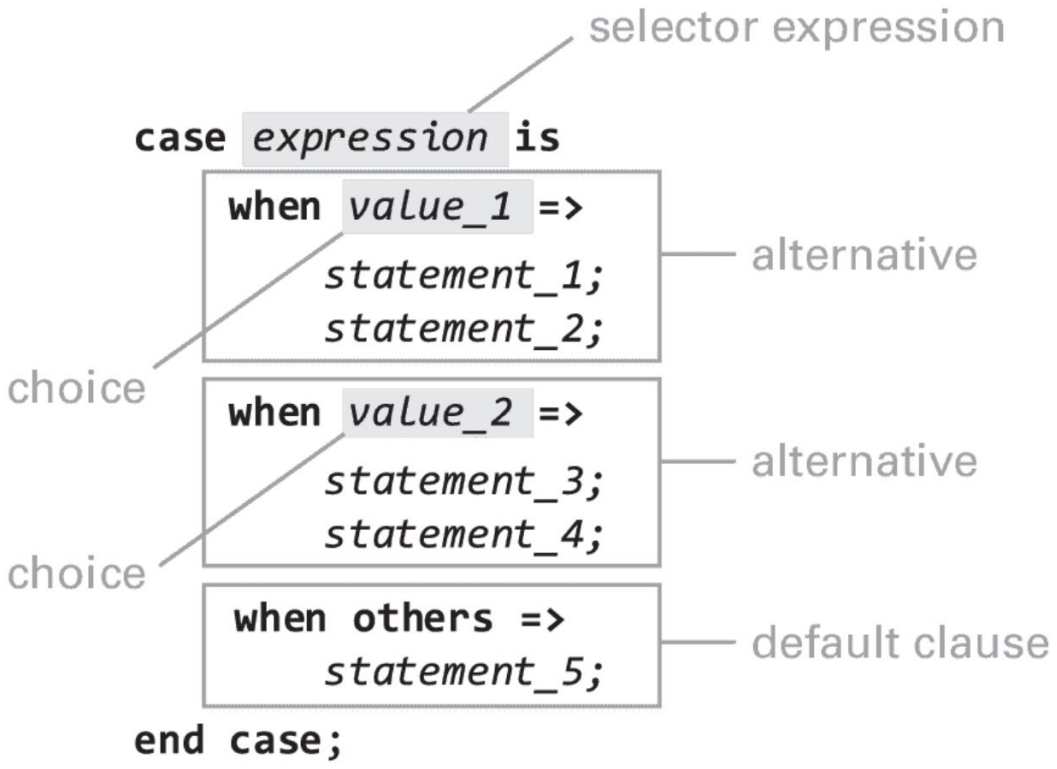


(b) An *if-elsif-else* statement.

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.

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.

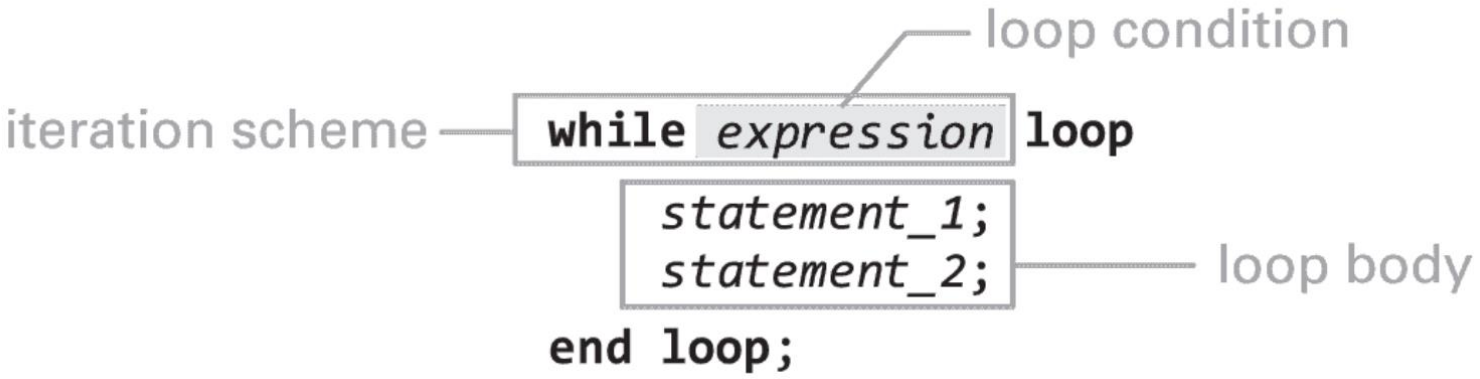
```
for i in 1 to 10 loop
    statements;
end loop;
```

(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

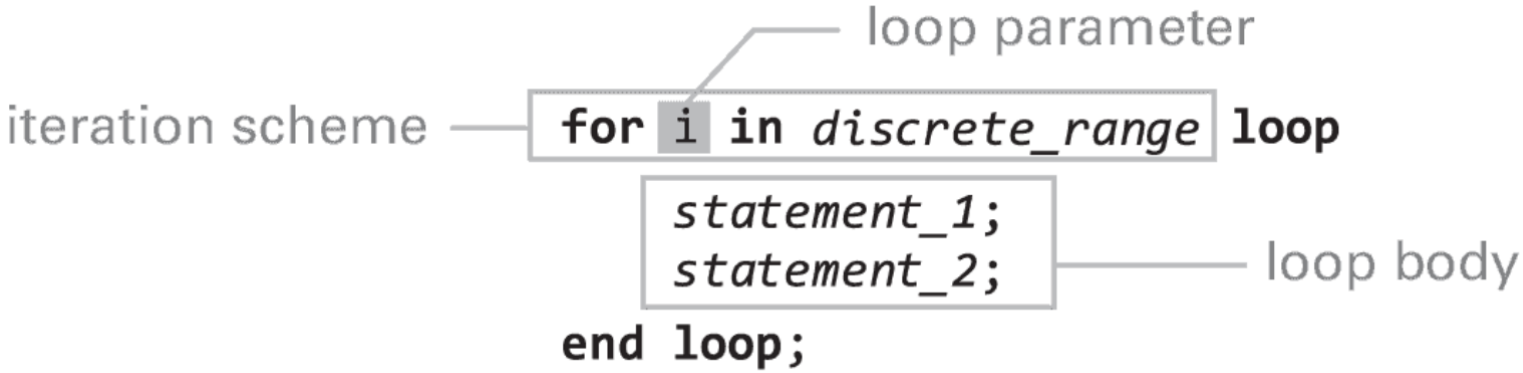
While Loop

- 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



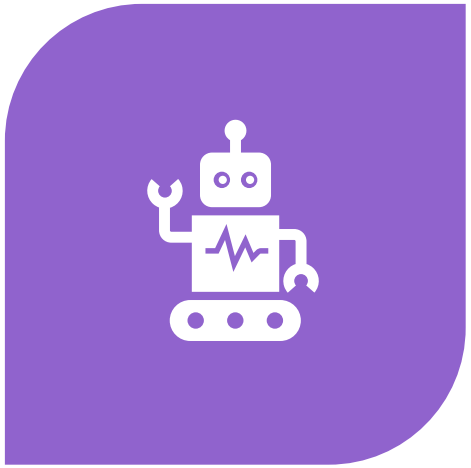
UNIVERSITY OF TURKU

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;
```

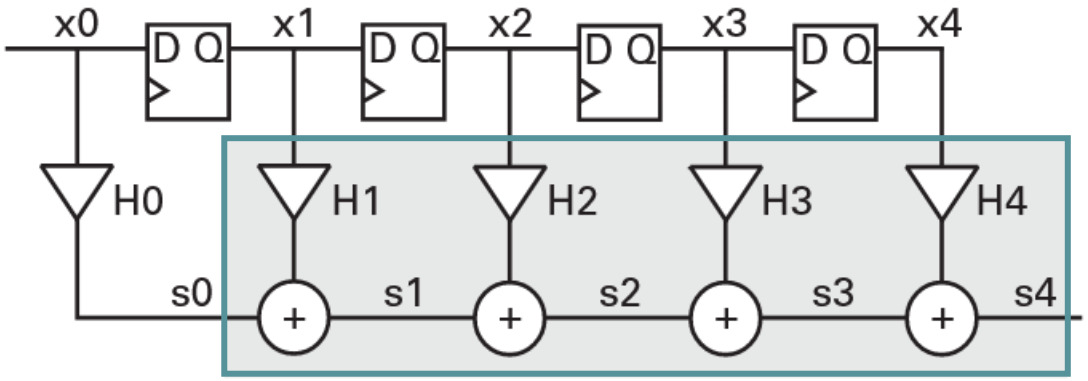
Loops in Hardware

- Each iteration will originate its own set of hardware elements

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

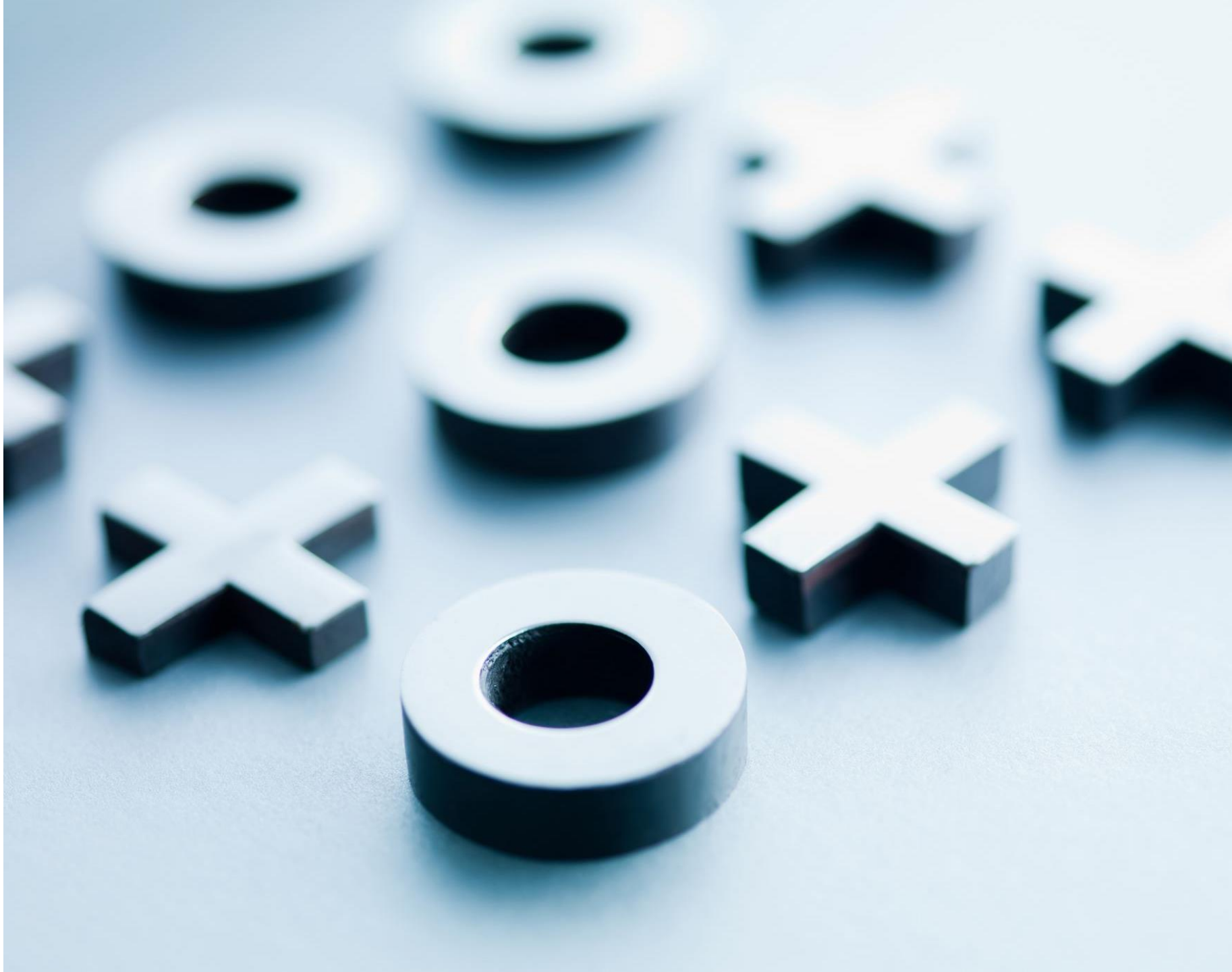
Loops in Hardware

- Each iteration will originate its own set of hardware elements
  - NOTE that the code does not implement the registers shown in the figure




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

Operators



Operators

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

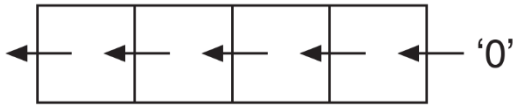
Operator Category	Operators
Miscellaneous	`**` `abs` `not`
	Unary version of `and` `or` `nand` `nor` `xor` `xnor` [(1)]
Multiplying operators	`*` `/` `mod` `rem`
Sign operators	`+` `-`
Adding operators	`+` `-` `&`
Shift operators	`sll` `srl` `sla` `sra` `rol` `ror`
Relational operators	`=` `/=` `<` `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

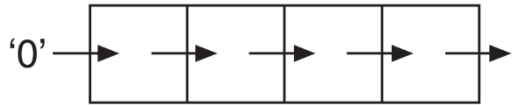
UNIVERSITY OF TURKU

Shift Operators

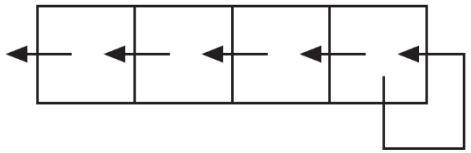
sll



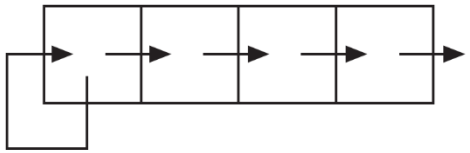
srl



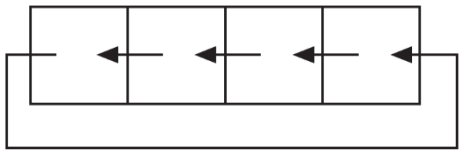
sla



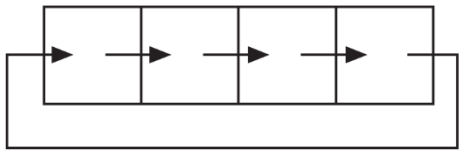
sra



rol

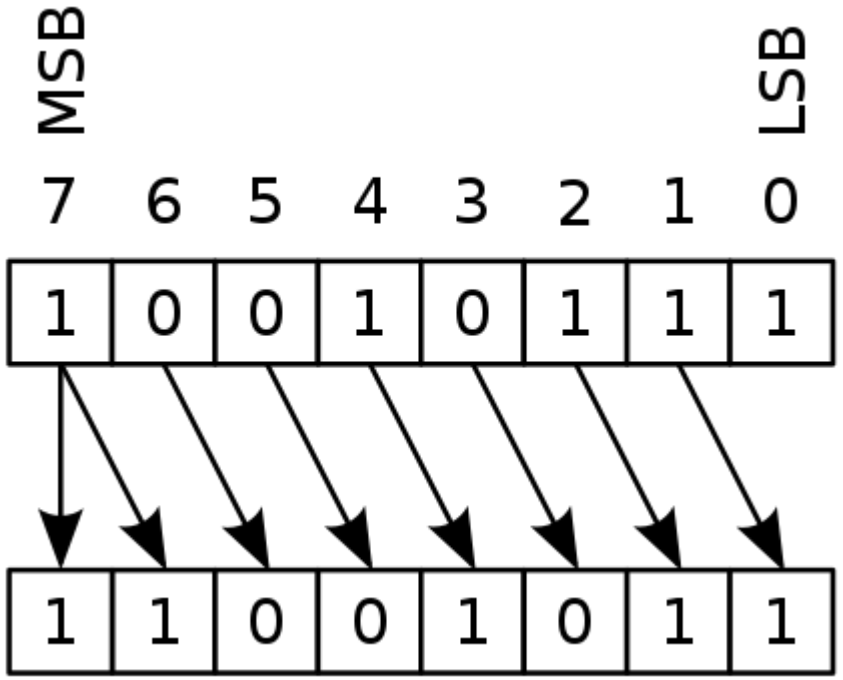


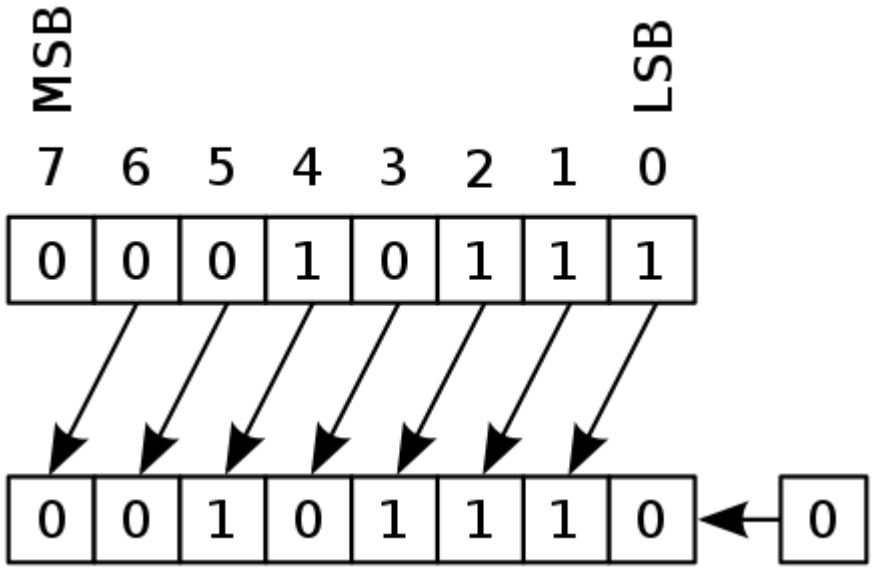
ror



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

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

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

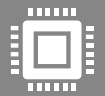
```
signal signal_1 : type_name := default_value_expression;
signal signal_2 : type_name := default_value_expression;
signal  ...     : type_name := default_value_expression;
```

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.

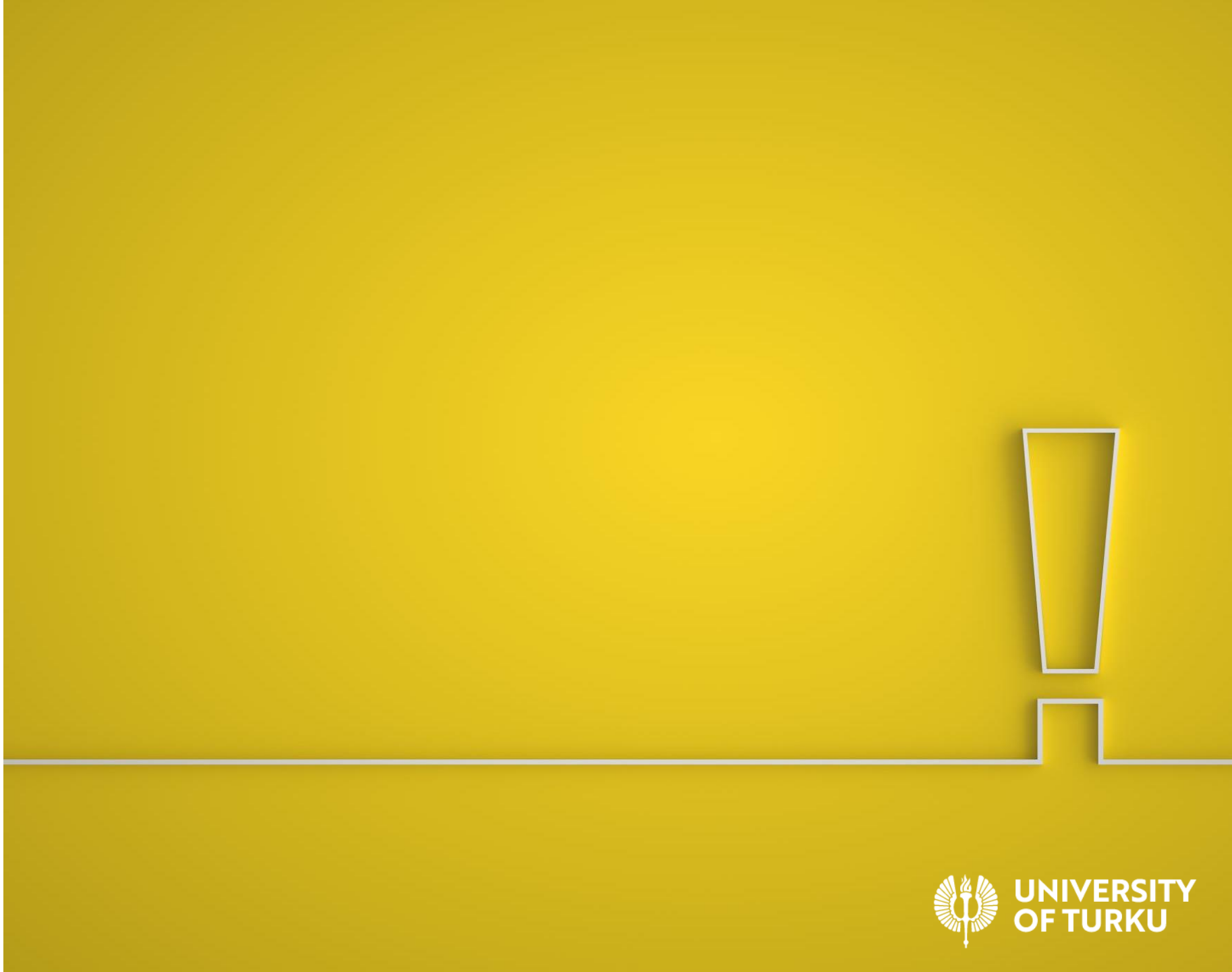


Every port in an entity is a signal

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



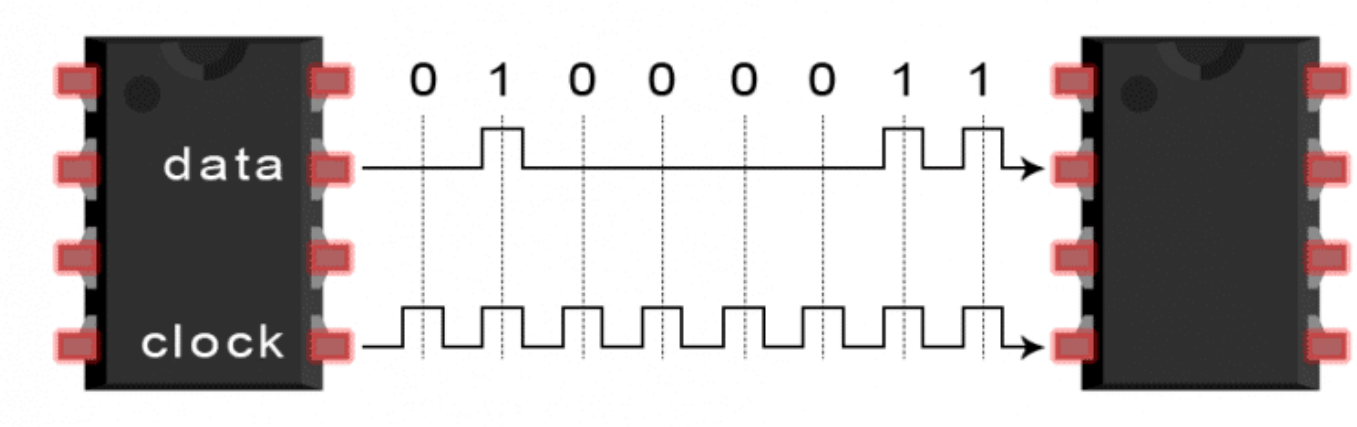
Facts: Signals and Processes

- 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

Signals between Modules



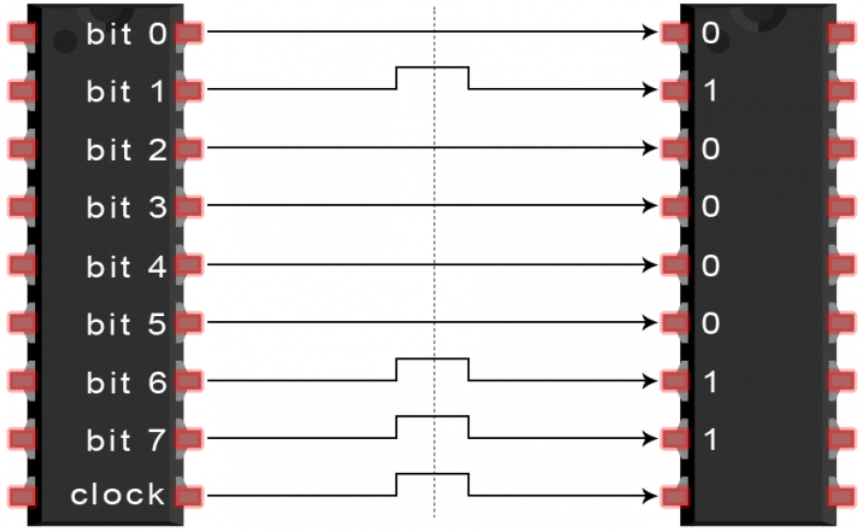

```
entity Module_1 is
    port(Dout : out std_logic;
         clock: in std_logic);
end half_adder;
```

→

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

UNIVERSITY OF TURKU

Signals between Modules




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

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

UNIVERSITY OF TURKU

Signals between Modules

- Signals can also be of type integer

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

How many

bits this is?

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

UNIVERSITY OF TURKU

Signals between Modules

- Signals can also be of type integer

Vivado synthesis implements an integer on 32 bits by default. For a more compact implementation, define the exact range of applicable values

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

How many bits this is?

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

```
entity Module_1 is
    port(Dout: out integer range 0 to 15);
end half_adder;
```

```
entity Module_1 is
    port(Din: in integer range 0 to 15);
end half_adder;
```

UNIVERSITY OF TURKU

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**

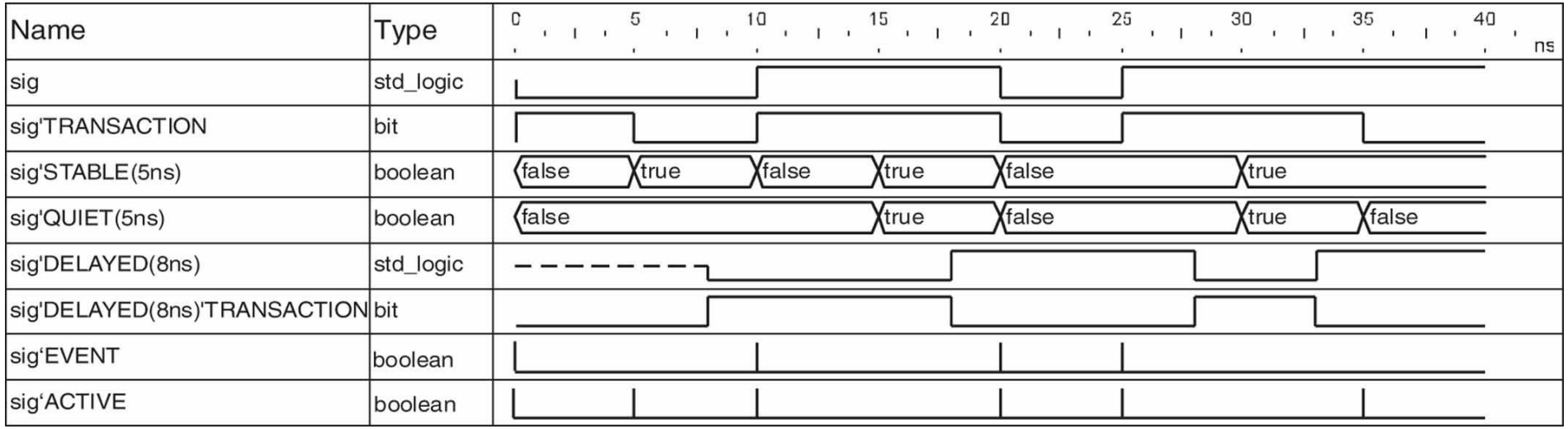
- All the above signal types are defined in libraries

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.



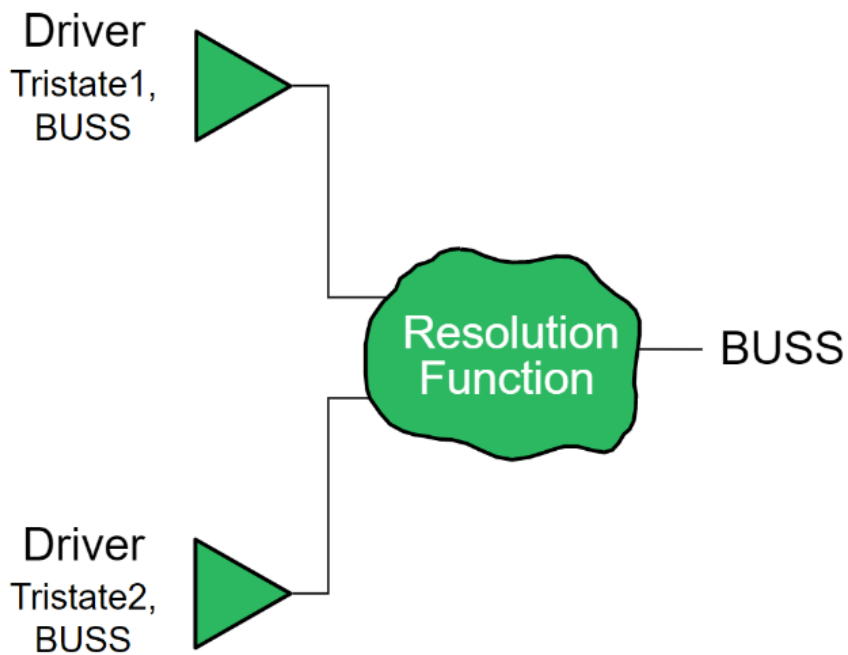
UNIVERSITY OF TURKU

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