SYSTEM MODELLING AND SYNTHESIS WITH HDL

DTEK0078
2023 Lecture 7

UNIVERSITY OF TURKU

Our research website: https://tiers.utu.fi/

Storage

UNIVERSITY OF TURKU

Edge Sensitive Storage

- In VHDL code, clock signals are no different than any other signal
- A register is inferred when a signal or variable is updated on a clock edge
  - in other words, when we do an assignment inside a region of a process that is subject to a clock edge

Synchronous Process: Flip-Flop

```
process (clk)
begin
    if rising_edge(clk) then
        q <= d ;
    end if;
end process;
```

```
process
begin
    wait until rising_edge(clk);
    q <= d ;
end process;
```

UNIVERSITY OF TURKU

Sync and Async Reset Signal

```
process (clk)
begin
  if rising_edge(clk) then
    if rst = ‘1’ then
      q <= 0;
    else
      q <= d ;
    end if;
  end if;
end process;
```

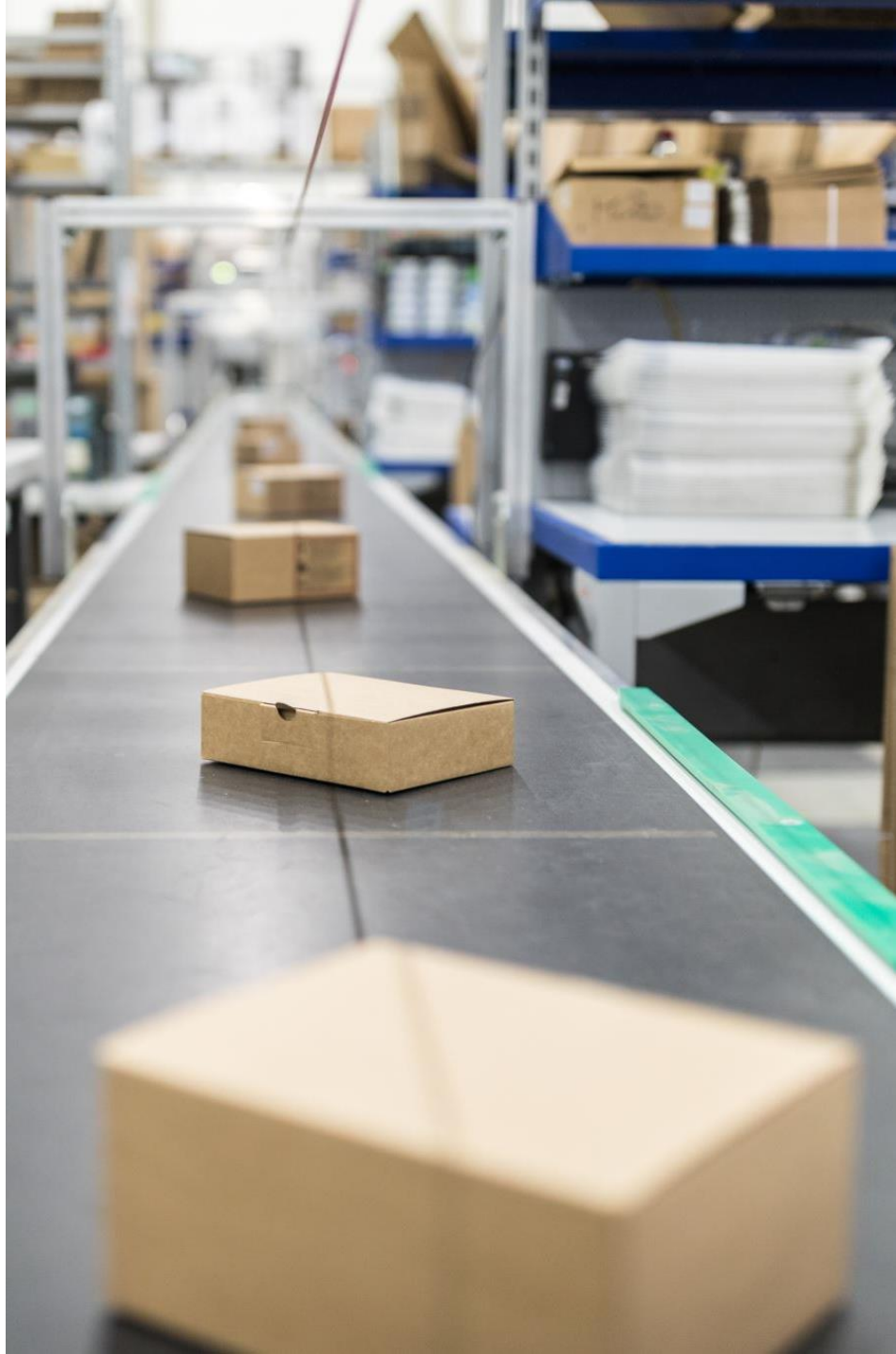
```
process (clk,rst)
begin
    if rst = ‘1’ then
        q <= 0;
    elsif rising_edge(clk) then
        q <= d ;
    end if;
end process;
```

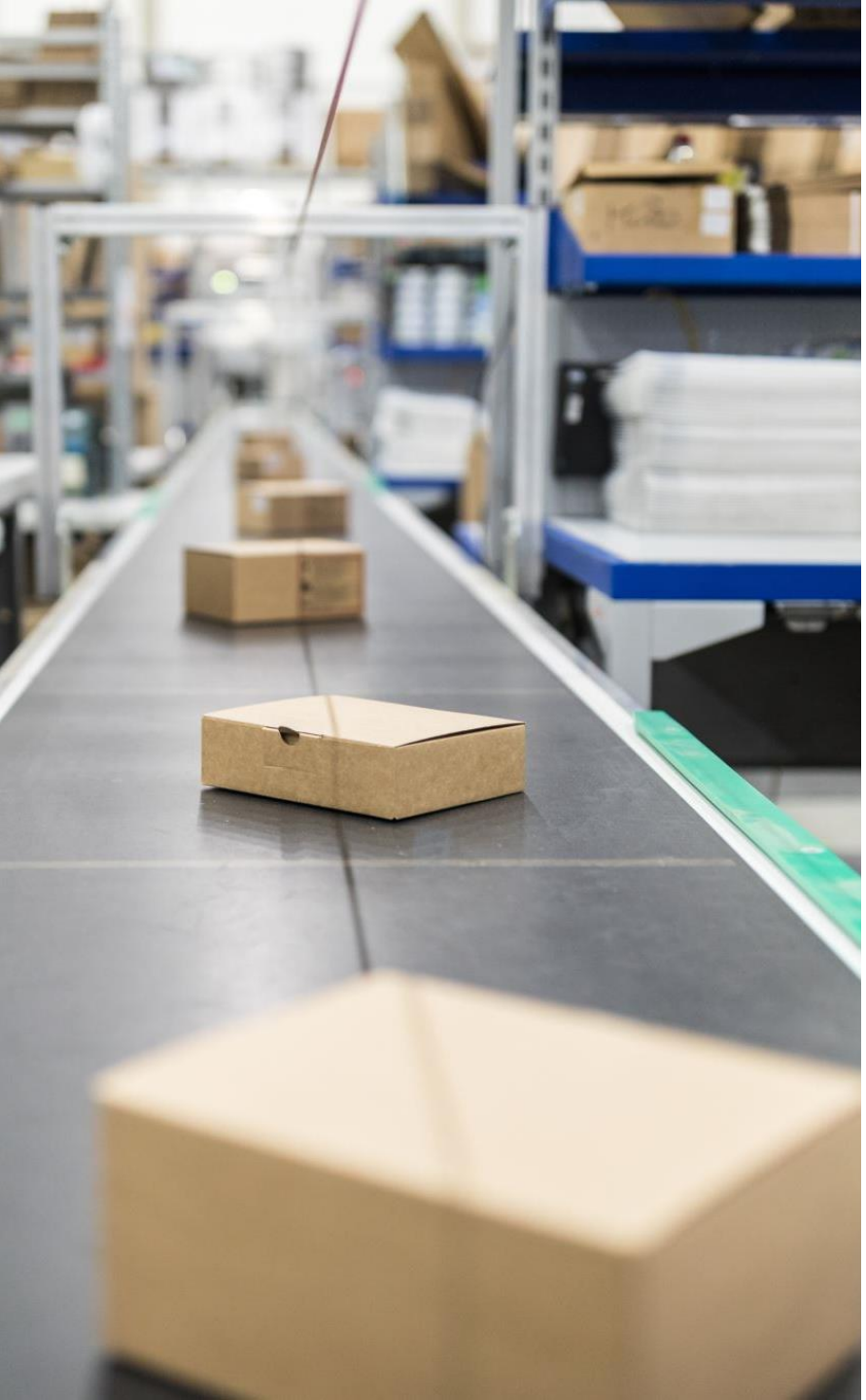
UNIVERSITY OF TURKU

In a synchronous design, only edge-sensitive storage elements should be used

Level-sensitive Storage

- Level-sensitive storage may happen unintentionally, sometimes because of coding styles that lead to the inference of latches
- It is important to know how to model latches to avoid them
- Latches can only be inferred in assignments that are not under control of a clock edge
- Use latches only if you know what you are doing – and then with comments explaining why





Synchronous Process: Latch

```
process (clk,d)
begin
  if clk= ‘1’ then
    q <= d ;
  end if;
end process;
```

```
process
begin
   wait until clk=‘1’;
   q <= d ;
end process;
```

UNIVERSITY OF TURKU

Inferring Latches

- A latch is inferred whenever a signal is assigned in one execution path and is not assigned in some other path

Inferring Latches

- A latch is inferred whenever a signal is assigned in one execution path and is not assigned in some other path

```
-- implies a latch
1 <= d when enable;
-- implies a combinational function (d and enable)
q2 <= d when enable else '0';
-- imples a register
q3 <= d when rising_edge(clock);
```

Inferring Latches

- A latch is inferred whenever a signal is assigned in one execution path and is not assigned in some other path

- The inference of latches in a combinational process as:

For a signal, a latch is inferred whenever the signal is assigned in a combinational process but not in all possible execution paths

For a variable, a latch is inferred whenever the variable is read before it is assigned a value and the assignment is not under control of a clock edge

Inference of Storage Element

For a signal, a register is inferred whenever the signal has an assignment under control of a clock edge

For a variable, a register is inferred whenever the variable has an assignment under control of a clock edge and is read before it is assigned a value

For a signal, a latch is inferred whenever the signal is assigned in a combinational process but not in all possible execution paths

For a variable, a latch is inferred whenever the variable is read before it is assigned a value and the assignment is not under control of a clock edge

Inference of Storage Element

```
process (all) begin
    if condition then
        q1 <= d;
    end if;
end process;
```

(a) Latch inferred due to incomplete assignment.

Inference of Storage Element

```
process (all) begin
    if condition then
        q1 <= d;
    end if;
end process;
```

(a) Latch inferred due to incomplete assignment.

```
process (all) begin
    if condition then
        q2 <= d;
    else
        q2 <= '0';
    end if;
end process;
```

(b) Latch prevented with an *else* clause.

```
process (all) begin
    q3 <= '0';
    if condition then
        q3 <= d;
    end if;
end process;
```

(c) Latch prevented with a default, unconditional assignment.

Inference of Storage Element

```
process (current_state) begin
    case current_state is
        when red =>
            red_light <= '1';
            green_light <= '0';
            yellow_light <= '0';
        when green =>
            red_light <= '0';
            green_light <= '1';
            -- Oops, missing assignment
            -- to signal yellow_light
        when yellow =>
            red_light <= '0';
            green_light <= '0';
            yellow_light <= '1';
    end case;
end process;
```

(a) A latch is inferred because of a missing assignment.

```
process (current_state) begin
    -- Default assignments prevent
    -- the inference of latches
    red_light <= '0';
    green_light <= '0';
    yellow_light <= '0';

    case current_state is
        when red =>
            red_light <= '1';
        when green =>
            green_light <= '1';
        when yellow =>
            yellow_light <= '1';
    end case;
end process;
```

(b) Latch prevented with default initial assignments.

UNIVERSITY OF TURKU

Finite State Machines

UNIVERSITY OF TURKU

Mealy and Moore

Moore machine’s outputs are a function of the present state only

Mealy machine’s outputs are a function of the present state and present inputs

UNIVERSITY OF TURKU

Mealy and Moore

- Can be functionaly equivalent
- Mealy usually requires smaller number of states
- Mealy reacts one clock cycle sooner than Moore
- Moore does not have a combinational path from input to output

Finite State Machine

- A finite state machine (FSM) is an abstraction of a sequential circuit whose possible internal states are enumerated by the designer
- The FSM is always in exactly one state (called its current state) and can transition to a different state in response to an event
- To design an FSM, we must specify its
  - states, input signals, output signals, next-state function, and output function
- The outputs of an FSM may depend on its current state and current inputs

Finite State Machine

- Can be implemented as single process or two process ways
  - single-process FSM: all the blocks are implemented in the same process statement
  - two-process FSM: the most common division is to code the state register as a clocked process and the next state and output logic blocks as a combinational process

```
architecture single_proc of turnstile_fsm is
begin
  process (clock, reset)
    type state_type is (locked, unlocked);
    variable state: state_type;
  begin
    -- Single process for state register,
    -- next-state logic, and output logic:
    if reset then
      state := locked;
      lock <= '1';
    elsif rising_edge(clock) then
      case state is
        when locked =>
          if ticket_accepted then
            lock <= '0';
            state := unlocked;
          else
            lock <= '1';
            state := locked;
          end if;
        when unlocked =>
          if pushed_through then
            lock <= '1';
            state := locked;
          else
            lock <= '0';
            state := unlocked;
          end if;
```

```
architecture two_proc of turnstile_fsm is
  type state_type is (locked, unlocked);
  signal current_state, next_state:
    state_type;
begin
  -- State register:
  process (clock, reset) begin
    if reset then
      current_state <= locked;
    elsif rising_edge(clock) then
      current_state <= next_state;
    end if;
  end process;

  -- Next-state and output logic:
  process (all) begin
    case current_state is
      when locked =>
        if ticket_accepted then
          lock <= '0';
          next_state <= unlocked;
        else
          lock <= '1';
          next_state <= locked;
        end if;
      when unlocked =>
        if pushed_through then
          lock <= '1';
          next_state <= locked;
```

Single

+ entirely encapsulated implementation within the process
+ a single object to keep the state
+ no risk of accidental latches

- all outputs are registered by default
- may become too complicated

```
architecture single_proc of turnstile_fsm is
begin
  process (clock, reset)
    type state_type is (locked, unlocked);
    variable state: state_type;
  begin
    -- Single process for state register,
    -- next-state logic, and output logic:
    if reset then
      state := locked;
      lock <= '1';
    elsif rising_edge(clock) then
      case state is
        when locked =>
          if ticket_accepted then
            lock <= '0';
            state := unlocked;
          else
            lock <= '1';
            state := locked;
          end if;
        when unlocked =>
          if pushed_through then
            lock <= '1';
            state := locked;
          else
            lock <= '0';
            state := unlocked;
          end if;
```

```
architecture two_proc of turnstile_fsm is
  type state_type is (locked, unlocked);
  signal current_state, next_state:
    state_type;
begin
  -- State register:
  process (clock, reset) begin
    if reset then
      current_state <= locked;
    elsif rising_edge(clock) then
      current_state <= next_state;
    end if;
  end process;

  -- Next-state and output logic:
  process (all) begin
    case current_state is
      when locked =>
        if ticket_accepted then
          lock <= '0';
          next_state <= unlocked;
        else
          lock <= '1';
          next_state <= locked;
        end if;
      when unlocked =>
        if pushed_through then
          lock <= '1';
          next_state <= locked;
```

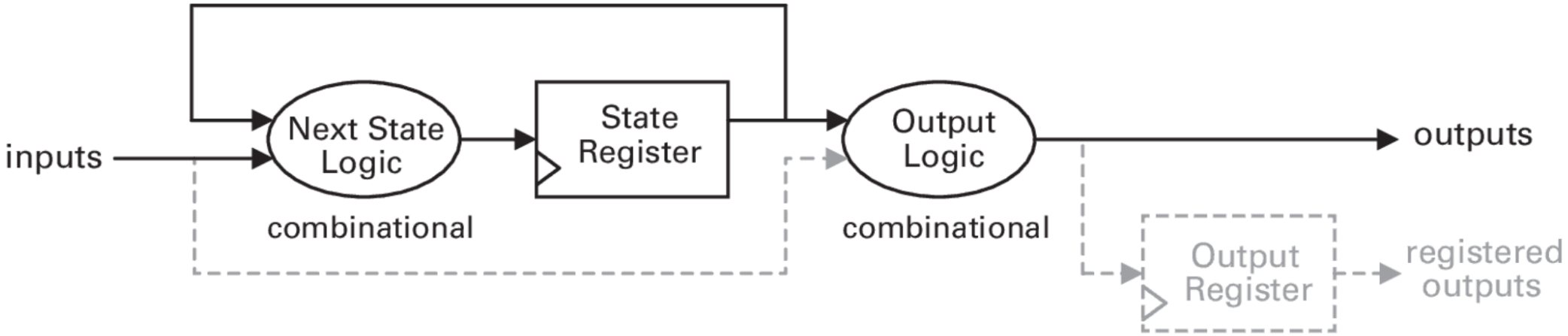
Two

+ Output are registered or not
+ Each process has a clear task

- Bad design leads unwanted latches
- may become too complicated

Finite State Machine

- An FSM can be implemented as a synchronous digital circuit using a register to keep its current state, and combinational logic to implement the next-state and output functions



VHDL outline for Moore Machine

```
type statemachine is (S0, S1, … , SX);
signal state: statemachine;

Moore: process (clock, reset)
begin
  if (reset = ‘1’) then state <= S0;
  elsif (clock = ‘1’ and clock’event) then
    case state is
    when S0 =>
      if input = ‘1’ then   state <= S1;
      else  state <= S0;
      end if;
    when S1 =>
    ...
    end case;
    end if;
end process;

output <= ‘1’ when state = SX else ‘0’;
```

UNIVERSITY OF TURKU

VHDL outline for Mealy Machine

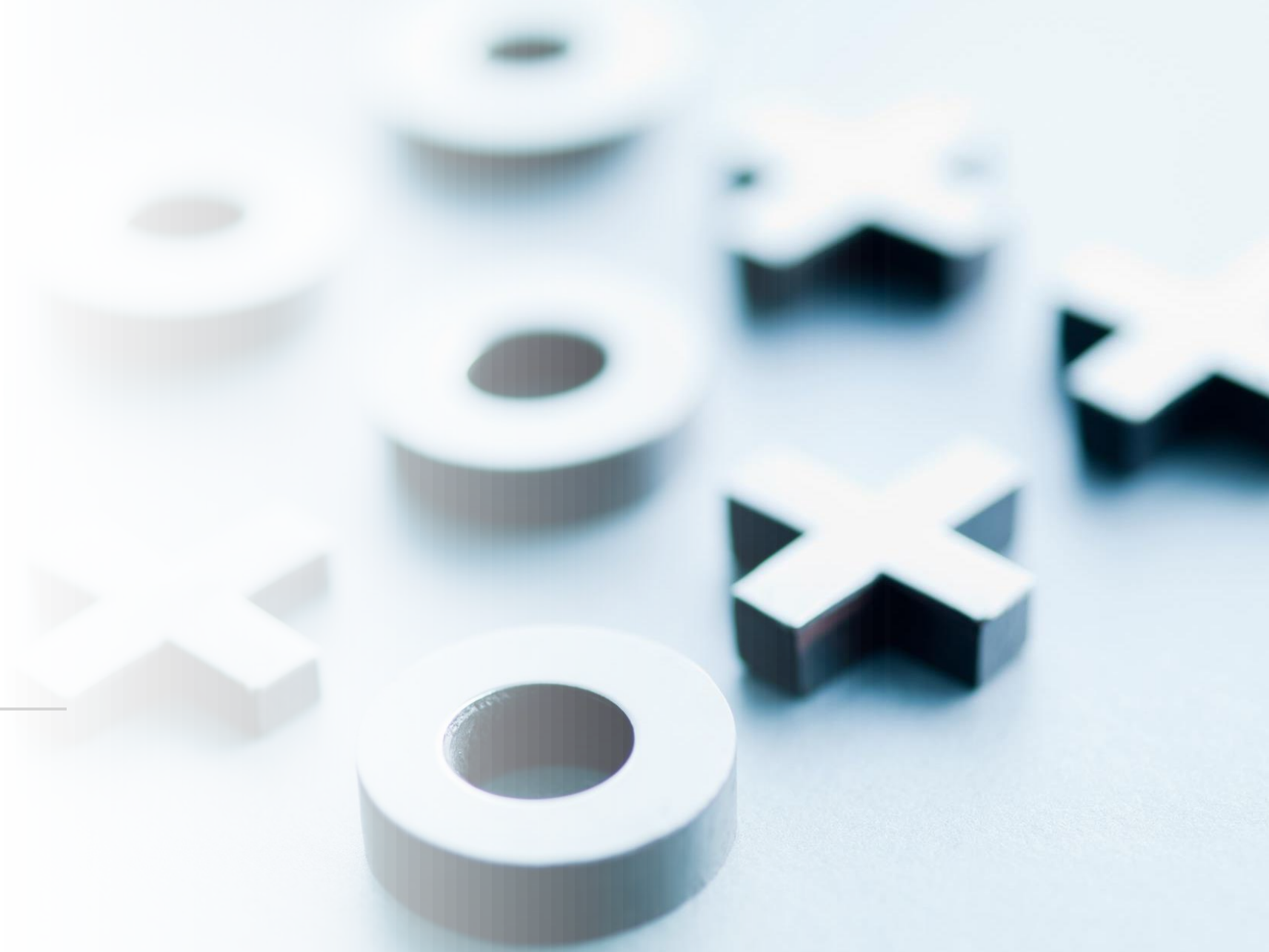
```
type statemachine is (S0, S1, … ,  SX);
signal state: statemachine;

Mealy: process(clock, reset)
begin
  if (reset = ‘1’) then                state <= S0;
  elsif (clock = ‘1’ and clock’event) then
    case state is
      when S0 =>
        if input = ‘1’ then state <= S1;
        else                state <= S0;
        end if;
      when S1 =>
      ...
    end case;
  end if;
end process;

output <= ‘1’ when (state = SX-1 and input = ‘0|1’) else ‘0’;
```

UNIVERSITY OF TURKU

Routines



UNIVERSITY OF TURKU

Routines

A sequence of statements that have been wrapped together and given a name

- routines, subroutines, subprograms, functions, procedures, and methods mean essentially the same

The term *subprogram* is given in the VHDL LRM for both *functions* and *procedures*

Used for organizing our source code

- It makes sense to create them even when they have a fixed behavior or when they are used only once in the code

Why Use Routines

- To Reduce Code Complexity
- To Avoid Duplicating Code
- To Make the Code More Readable
- To Make the Code Easier to Test
- To Make the Code Easier to Change

```
-- sub bytes
for i in 0 to 3 loop
    for j in 0 to 3 loop
        state_2(i,j) :=
            sbox(state_1(i,j));
    end loop;
end loop;

-- shift rows
for i in 0 to 3 loop
    for j in 0 to 3 loop
        state_3(i,j) :=
            state_2(i,(j+i) mod 4);
    end loop;
end loop;

-- add round key
for i in 0 to 3 loop
    for j in 0 to 3 loop
        state_4(i,j) := state_3(i,j) xor
            round_key(i,j);
    end loop;
end loop;
```

(a) A complex operation performed in a big block of sequential code.

```
function sub_bytes(state: state_type)
    return state_type is
begin
    -- Code for operation 'sub bytes'
end;

function shift_rows(state: state_type)
    return state_type is
begin
    -- Code for operation 'shift rows'
end;

function add_round_key(state: state_type)
    return state_type is
begin
    -- Code for operation 'add round key'
end;

...

state_2 := sub_bytes(state_1);
state_3 := shift_rows(state_2);
state_4 := add_round_key(state_3);
```

(b) The same operation broken down into three simpler routines.

To Reduce Code Complexity

UNIVERSITY OF TURKU

```
function sub_bytes(state: state_type)
    return state_type is
begin
    -- Code for operation 'sub bytes'
end;

function shift_rows(state: state_type)
    return state_type is
begin
    -- Code for operation 'shift rows'
end;

function add_round_key(state: state_type)
    return state_type is
begin
    -- Code for operation 'add round key'
end;

...

state_2 := sub_bytes(state_1);
state_3 := shift_rows(state_2);
state_4 := add_round_key(state_3);
```

(b) The same operation broken down into three simpler routines.

Two main benefits

1. each routine can be understood in isolation. Because each routine is smaller, is more focused, and has less variables, each of them is easier to understand than the original task

2. the code using the routines provides a clear overview of the main task

To Reduce Code Complexity

```
-- Convert x and y coordinates to memory addresses in the framebuffer
write_address <= x_in(9 downto 2) & y_in(9 downto 2);
read_address <= x_out(9 downto 2) & y_out(9 downto 2);
next_address <= x_next(9 downto 2) & y_next(9 downto 2);
```

```
function memory_address_from_x_y(x, y: in bit_vector) return bit_vector is
begin
    return x(9 downto 2) & y(9 downto 2);
end;
...
write_address <= memory_address_from_x_y(x_in, y_in);
read_address <= memory_address_from_x_y(x_out, y_out);

next_address <= memory_address_from_x_y(x_next, y_next);
```

To Avoid Duplicating Code

```
function memory_address_from_x_y(x, y: in bit_vector) return bit_vector is
begin
    return x(9 downto 2) & y(9 downto 2);
end;
...
write_address <= memory_address_from_x_y(x_in, y_in);
read_address <= memory_address_from_x_y(x_out, y_out);

next_address <= memory_address_from_x_y(x_next, y_next);
```

- The knowledge about how to calculate an address from a pair of coordinates is located in a single place
- The function name is self-explanatory making the code self-documenting and easier to understand
- You can ignore the details of how to transform a pair of coordinates to a memory address and concentrate on higher level functionality

To Avoid Duplicating Code

```
-- Check whether my_array contains a 0
number_to_check := 0;
element_found := false;
for i in my_array'range loop
    if my_array(i) = number_to_check then
        element_found := true;
    end if;
end loop;
...
```

To Make the Code More Readable

```
-- Check whether my_array contains a 0
number_to_check := 0;
element_found := false;
for i in my_array'range loop
    if my_array(i) = number_to_check then
        element_found := true;
    end if;
end loop;
...
```

- Sometimes the best way to make a sequence of statements easier to read is to hide them behind a routine with a descriptive name

```
array_contains_zero := array_contains_element(my_array, 0);
```

To Make the Code More Readable

UNIVERSITY OF TURKU

Functions vs. Procedures

Function	Procedure
is similar to a function in mathematics: it processes one or more input values and produces exactly one output value	encapsulates a group of statements that communicate with the calling code in two possible ways: via input and output parameters or by direct manipulation of objects declared outside the procedure

Functions vs. Procedures

```
function sum_function(vect: integer_vector) return integer is
    variable sum: integer := 0;
begin
    for i in vect'range loop
        sum := sum + vect(i);
    end loop;
    return sum;
end;
```

(a) A function.

```
procedure sum_procedure(vect: in integer_vector; sum: out integer) is
begin
    sum := 0;
    for i in vect'range loop
        sum := sum + vect(i);
    end loop;
end;
```

(b) A procedure.

UNIVERSITY OF TURKU

Functions

A function encloses a sequence of statements that operate on zero or more input values and produce exactly one output value

What really matters is the value that the function calculates and returns

A function call is used as part of an expression; it cannot be used as a stand-alone statement

Functions

Cannot use wait statements

Cannot be used as a stand-alone statement

Must produce its result in a single simulation cycle

Function parameters can only be of mode in, and their classes must be constant, signal, or file (variables are not allowed).

Procedures

A procedure encloses a sequence of statements that do not return a value

A procedure call cannot be used in an expression; it must be a stand-alone statement

A procedure can calculate as many output values as we wish, as long as they are communicated via output parameters

Procedures

Procedures do not have as many limitations as functions

Procedures can be used to implement modules with flexible interfaces

Procedure parameters can be of mode *in*, *out*, or *inout*, and their classes can be constant, variable, signal, or file

Because of their flexibility, procedures can be used almost like an entity or component

Functions vs. Procedures

Characteristic	Functions	Procedures
Where and how it is used	In concurrent or sequential code. Always in an expression.	In concurrent or sequential code. Used as a standalone statement.
How to send values to the subprogram	Via *input* parameters.	Via *input* or *inout* parameters; via other objects visible to the subprogram.
How to return values from the subprogram	Via a single return value (required).	Via one or more *output* or *inout* parameters; via visible objects writable by the subprogram.
Designator (name)	An identifier or an operator (e.g., `"+"`, `"-"`, `"and"`, `"or"`).	An identifier.
Timing issues	Cannot include *wait* statements; must be evaluated in a single simulation cycle.	Can include *wait* statements, which cause the calling process to suspend.

Functions vs. Procedures

Use a function ...	Use a procedure ...
When the main goal of the subprogram is to produce a value.	When the main goal of the subprogram is to execute statements or change the values of existing objects.
When the subprogram must return a single value.	When the subprogram does not produce a value, or when it produces more than one value.
When the subprogram call is part of an expression.	When the subprogram call is a standalone statement.
For data conversion, logic, and arithmetic operations.	For operations involving the passing of time (using a *wait* statement).
To overload predefined operators (e.g., `and`, `or`, `"+"`, `"-"`).	When you need side effects (e.g., file operations).
To implement mathematical formulas.	To implement algorithms involving the passage of time, or to group a sequence of statements.
Whenever you can, to keep the client code simple.	Whenever you need more flexibility than provided by pure functions.*

Make It Small

Short routines are naturally less complex

THEY ARE EASIER TO UNDERSTAND

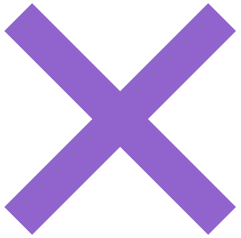
EASIER TO TEST

MORE LIKELY TO FOCUS ON A SINGLE PURPOSE

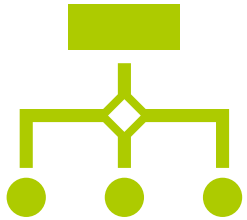
UNIVERSITY OF TURKU

Make It Small

Prevent routines from growing too large in the first place



Remove any duplicate code



Decompose it into a series of shorter subprograms

Use abstraction: find a sequence of statements that have a clear purpose and move them to their own routine

Give the Routine a Good Name:

Your goal is to make the use of the routine self-evident

UNIVERSITY OF TURKU

Warning

Formal parameters and local variables in subprograms are always initialized anew every time the subprogram is called

Implication for synthesis

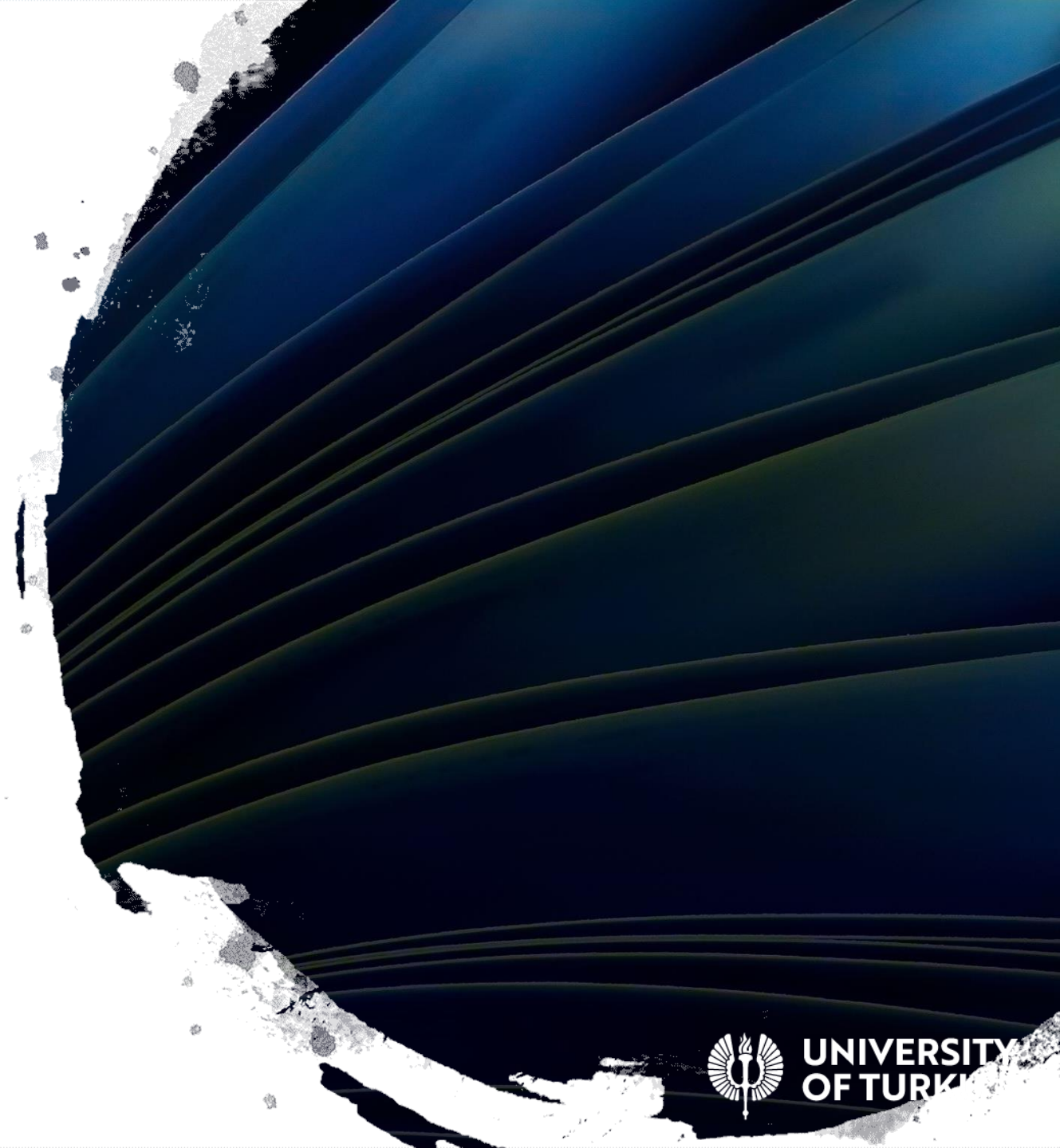
variables declared in subprograms never model storage elements because they are always assigned an initial value on every call

Where Should Routines Go?

Subprograms can be declared

- in a package
- in an entity
- in an architecture
- in a process,
- even inside another subprogram

In practice, most subprograms are declared in a package or in the declarative region of a process or an architecture




UNIVERSITY
OF TURKU