SYSTEM MODELLING AND SYNTHESIS WITH HDL

DTEK0078
2022 Lecture 8

UNIVERSITY OF TURKU

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

Finite State Machines

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

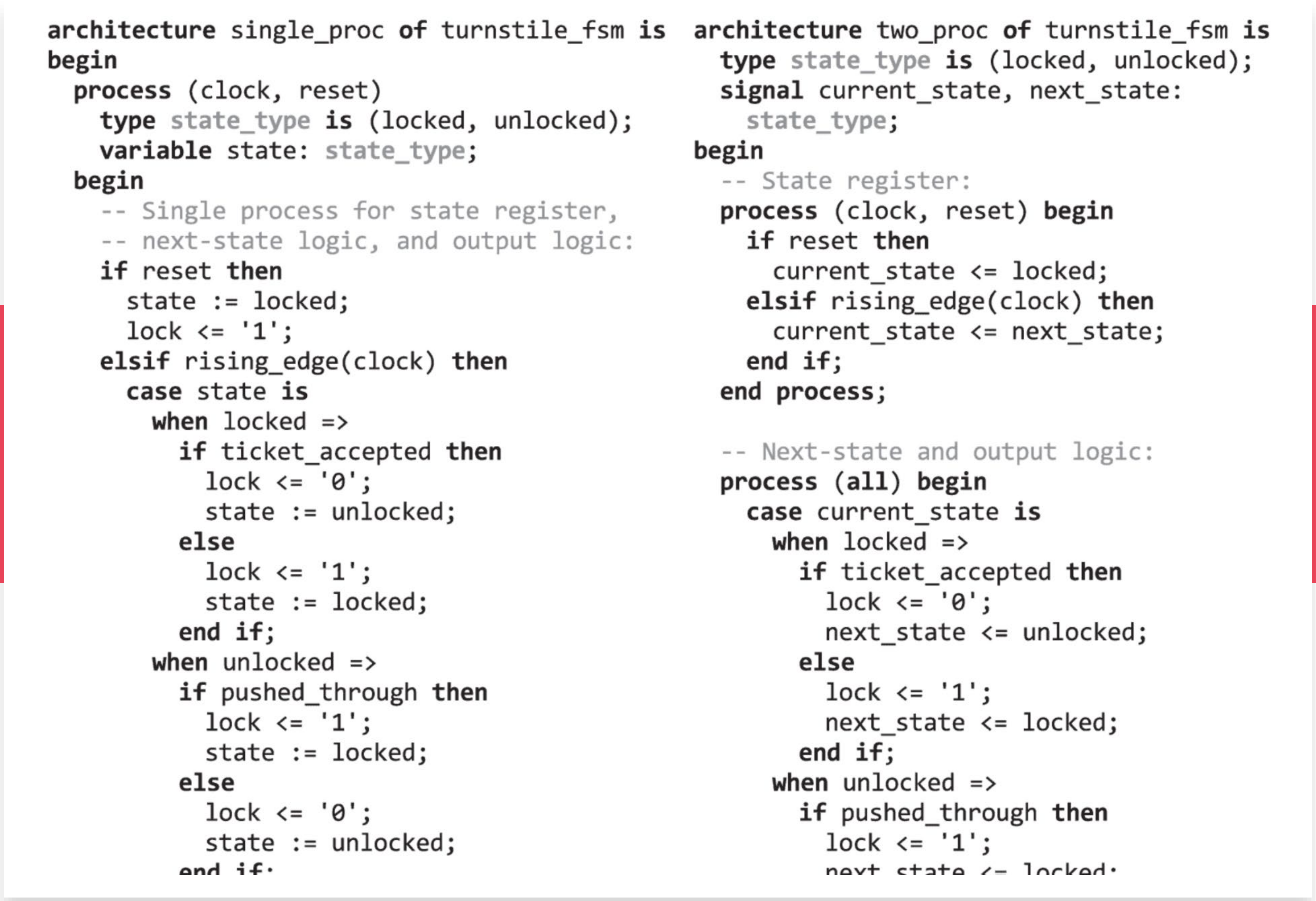
UNIVERSITY OF TURKU

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



```vhdl
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;
```

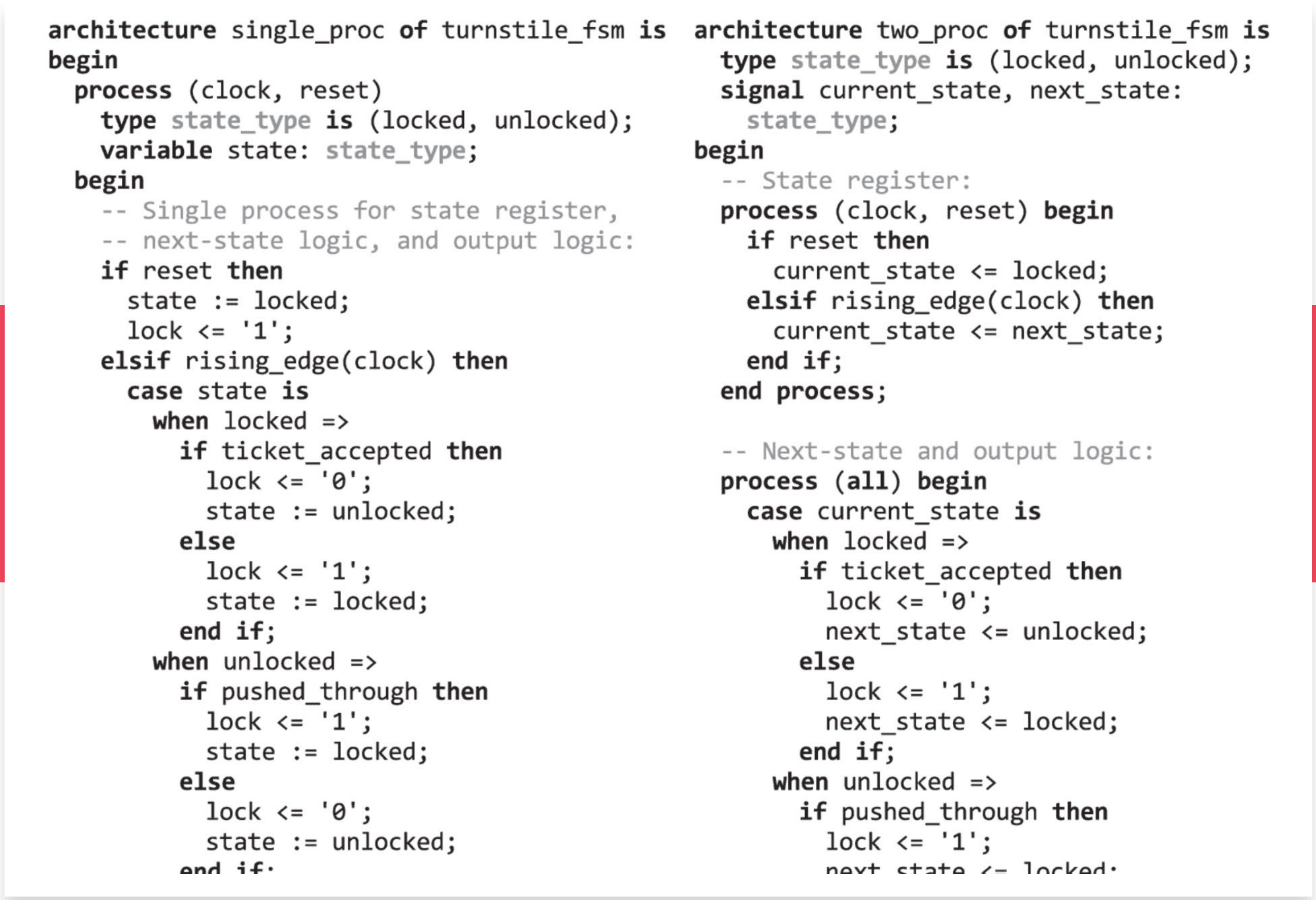
```vhdl
architecture two_proc of turnstile_fsm is
  type state_type is (locked, unlocked);
  signal current_state, next_state:
    state_type;
begin
  -- State register:
  process (clock, reset) begin
    if reset then
      current_state <= locked;
    elsif rising_edge(clock) then
      current_state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

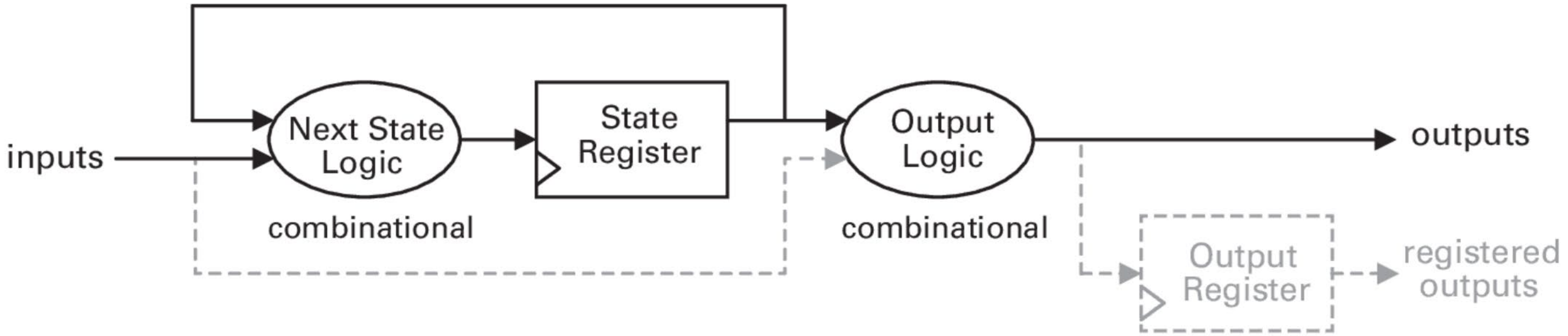
Two

+ 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

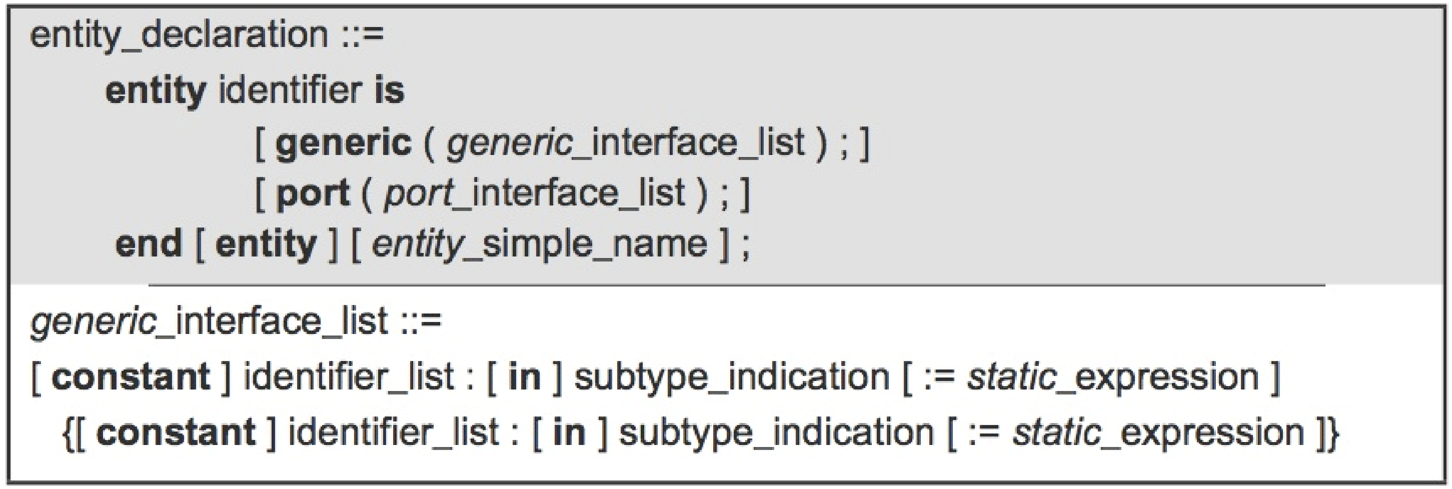
```
type statemachine is (S0, S1, … , SX);
signal state: statemachine;

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’;
```

Generics

UNIVERSITY OF TURKU



```
entity_declaration ::=
      entity identifier is
            [ generic ( generic_interface_list ) ; ]
            [ port ( port_interface_list ) ; ]
       end [ entity ] [ entity_simple_name ] ;
```

```
generic_interface_list ::=
[ constant ] identifier_list : [ in ] subtype_indication [ := static_expression ]
  {[ constant ] identifier_list : [ in ] subtype_indication [ := static_expression ]}
```

Generics

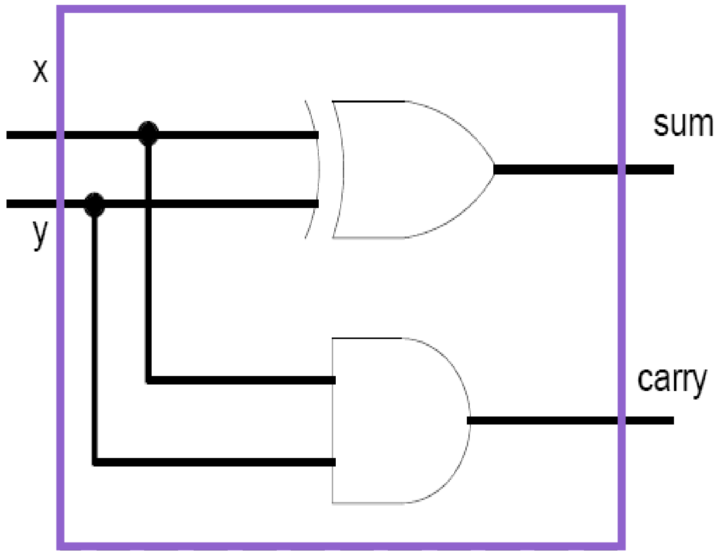
- Generics allow a design to be described so that its structure can be altered easily during the instantiation of the module

Generic Example

- Generics are visible in the entity in which it is declared as well as in the corresponding architecture body

```
entity half_adder is
generic (Tpd: time := 10 ns)
port( x,y: in std_logic;
      sum, carry: out std_logic);
end half_adder;

architecture myadder of half_adder is
begin
  sum <= x xor y after Tpd;
  carry <= x and y after Tpd;
end myadder;
```



Up Counter

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

entity Counte3 is
port( clk  : in  std_logic;
      reset: in  std_logic;
      count: out std_logic_vector(3 downto 0));
end Counte3;

architecture behavioural of Counte3 is
  signal counting: std_logic_vector(3 downto 0);
begin
  process (clk,reset)
    begin
      if reset = ‘0' then
         counting <= “0000”;
      elsif (rising_edge(clk)) then
         counting <= counting + 1;
      end if;
  end process;
  count <= counting;
end behavioural;
```

Up Counter

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

entity Counte3 is
generic( w: natural := 4)
port( clk:   in std_logic;
      reset: in std_logic;
      count: out std_logic_vector(w-1 downto 0));
end Counte3;

architecture behavioural of Counte3 is
  signal counting: std_logic_vector(w-1 downto 0);
begin
  process (clk,reset)
    begin
      if reset = '0' then
         counting <= (others => 0);
      elsif (rising_edge(clk)) then
         counting <= counting + 1;
      end if;
    end process;
  count <= counting;
end behavioural;
```

Port Mapping with Generics

```
entity Counte3 is
        generic( w: natural := 4)
        port( clk: in std_logic;
                   reset: in std_logic;
                   count: out std_logic_vector(w-1 downto 0));
        end Counte3;
```

```
c1:     entity work.counter (behavioural)
                generic map (16)
                port map (m_clk, m_reset, m_count1);
```

```
c2:     entity work.counter (behavioural)
                generic map (12)
                port map (m_clk, m_reset, m_count2);
```

```
c3:     entity work.counter (behavioural)
                generic map (w => 16)
                port map (m_clk, m_reset, m_count3);
```

UNIVERSITY
OF TURKU

Port Mapping with Generics (VHDL-2008)

```
generic ( type T; constant init_val : T );
signal v : T := init_val;


entity Counte3 is
  generic( type data_type)
  port( clk: in std_logic;
        reset: in std_logic;
        count: out data_type);
end Counte3;
```

```
c1: entity work.counter (behavioural)
       generic map (data_type => std_logic_vector(3 downto 0))
              port map (m_clk, m_reset, m_count);
```

Port Mapping with Generics (VHDL-2008)

NOTE

- To use types in generics requires one to be careful!
- For example, arithmetic operators are not defined for all the available types

```
entity counte3 is
generic( type data_type )
port(  . . .
        Dout : out
data_type);
architecture
. . .
Dout <= Dout + 1;
end architecture;
```

UNIVERSITY OF TURKU

UNIVERSITY
OF TURKU