IMTEK
UNI FREIBURG
FRITZ HUETTINGER CHAIR OF MICROELECTRONICS
PROF. DR-ING. Y. MANOLI

VHDL “Cheat Sheet”

Working with vectors

```
signal vector_8bit : std_logic_vector(7 downto 0);
signal vector_4bit : std_logic_vector(3 downto 0);
signal msb, lsb : std_logic;

-- individual signals can be accessed using index operations:
msb <= vector_8bit(7);
lsb <= vector_8bit(0);

-- This works for other vectors, too.
-- Example: assign the first four bits of vector_8bit to vector_4bit:
vector_4bit <= vector_8bit(7 downto 4);

-- Vectors can be concatenated with the '&' operator.
-- Example: concatenate vector_4bit to itself and assign the result
-- to vector_8bit:
vector_8bit <= (vector_4bit & vector_4bit);

-- This also works with constants.
-- Example: assign vector_4bit to the first 4 bits of vector_8bit
-- and fill the remaining bits with '0'
vector_8bit <= (vector_4_bit & "0000");
```

Component instantiation

A more compact way to directly instantiate an entity without having to declare a `component`.

```
architecture <super_architecture> of <super_entity> is
    -- No component declaration required.
begin
    <label>: entity <sub_entity>(<sub_architecture>)
        port map (...);
end architecture;
```

Registers

To create a synchronous register, use a process triggered on the rising clock edge and an enable signal. Multiple independent registers can be combined into a single register block.

```
signal data, reg1 : std_logic_vector(7 downto 0);
signal en_reg1 : std_logic;

process(clk)
begin
  if rising_edge(clk) then
    if en_reg1 = '1' then
      reg1 <= data;
    end if;
    -- More registers can be added here ...
    -- if en_reg2 = '1' then
    --     reg2 <= data;
    -- end if;
  end if;
end process;
```

Multiplexers

There is a simple way to write multiplexers and other combinatorial logic without a `process` statement.

Multiplexer with 2 inputs:

```
signal in1, in2, out1 : std_logic_vector(7 downto 0);
signal sel : std_logic;

out1 <= in1 when sel = '0' else in2;
```

When building a multiplexer with multiple inputs, make sure all cases are covered!

```
signal in1, in2, in3, out1 : std_logic_vector(7 downto 0);
signal sel : std_logic_vector(1 downto 0);

out1 <= in1 when sel = "00"
  else in2 when sel = "01"
  else in3 when sel = "10"
  else "00000000"; -- sel = "11" doesn't correspond to any input
```

Tri-State buffer

```
signal result, bus : std_logic_vector(7 downto 0):
signal en_result : std_logic;

-- (others => 'Z') is a simpler way to write "ZZZZZZZZ",
-- and independent of the actual vector width.
bus <= result when en_result = '1'
  else (others => 'Z');
```

Finite state machine in one process

All outputs are registers and therefore delayed by one clock cycle. This must be considered when assigning outputs that depend on a certain state.

```
architecture one_proc of fsm is
  type state_type is (state_0, state_1, state_2);
  signal state : state_type;
begin
  process(clk, reset)
  begin
    if reset='1' then
      state <= state_0;
      output_1 <= '0';
    elsif rising_edge (clk) then
      case state is
        when state_0 =>
          if input_1 = '1' then
            state <= state_1;
            -- output for state_1 is assigned here!
            output_1 <= input_2;
          end if;
        when state_1 =>
          state <= state_2;
          output_1 <= '0';
        --
        -- etc. ...
        --
        when others =>
          null;
      end case;
    end if;
  end process;
end one_proc;
```

Finite state machine in two processes

Since `next_state` and `output_1` are signals and not registers, values must be assigned in each `when` statement. This can be simplified by assigning default values before the `case` statement

```
architecture two_proc of fsm is
  type state_type is (state_0, state_1, state_2);
  signal state, next_state : state_type;
begin
  state_register: process(clk, reset)
  begin
    if rising_edge(clk) then
      if reset = '1' then
        state <= state_0;
      else
        state <= next_state;
      end if;
    end if;
  end process;

  next_state_and_output: process(state, input_1, input_2, ...)
  begin
    -- Set default values for all outputs.
    next_state <= state;
    output_1 <= '0';

    -- Overwrite defaults based on current state and inputs.
    case state is
      when state_0 =>
        if input_1 = '1' then
          next_state <= state_1;
        end if;
      when state_1 =>
        next_state <= state_2;
        output_1 <= input_2;
      --
      -- etc. ...
      --
      when others =>
      null;
    end case;
  end process;
end two_proc;
```