

DigMe: Digital Microelectronics

Naoki Pross – naoki.pross@ost.ch

Fall Semester 2021

Contents

1 License	i
2 Design Flow	1
3 Design constraints and static timing analysis (STA)	1
3.1 Physical constraints	1
3.2 Timing constraints	1
4 System level VHDL	1
4.1 Aliases	1
4.2 Generics	1
4.3 Generators	1
4.4 Functions and procedures	1
4.5 Arrays and records	1
4.6 Packages	2

1 License

This work is licensed under a [Creative Commons](https://creativecommons.org/licenses/by-nc-sa/4.0/) “Attribution-NonCommercial-ShareAlike 4.0 International” license.

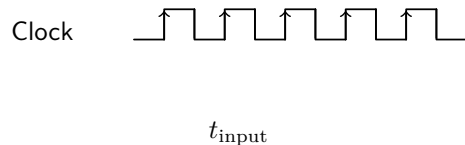


2 Design Flow

3 Design constraints and static timing analysis (STA)

3.1 Physical constraints

3.2 Timing constraints



4 System level VHDL

4.1 Aliases

The goal is now to build re-usable IP blocks with VHDL. For that we need to refresh some important features of the language. The first of which are aliases.

```
1 signal data_bus :
2   std_logic_vector(31 downto 0);
3 alias first_nibble :
4   std_logic_vector(0 downto 3)
5   is data_bus(31 downto 28);
```

4.2 Generics

4.3 Generators

Another useful feature are `generate` statement, with the syntax that allows the instantiation of multiple components.

```
1 [label]: for <identifier> in <range> generate
2   -- optional declaration part
3   -- begin only required if there is a
4     ↳ declaration
5 [begin]
6   -- concurrent statements
7 end generate [label];
```

For example:

```
1 for i in 0 to 7 generate
2   x(i) <= a(i) xor b(7 - i);
3 end generate;
```

Or in a more realistic case, with components imported from elsewhere.

Listing 1: Example of generate with a component.

```
1 -- in architecture
2 bcd_to_sseg_inst_loop:
3 for i in 0 to nr_digits - 1 generate
4   bcd_to_sseg_inst: component bcd_to_sseg
5     port map(
6       clk => clk,
7       rst => rst,
8       bcd => bcd_array(i),
9       sseg => sseg_array(i)
```

```
10     );
11 end generate;
```

where `bcd_array` and `sseg_array` are of course array types, and `nr_digits` is a constant.

4.4 Functions and procedures

Furthermore VHDL has functions that can be useful to avoid rewriting the same code. Function have multiple inputs and a signal output, are allowed to be called recursively, but cannot declare or assign signals, nor use `wait` statements.

```
1 function <name> ([list of arguments with type])
2   return <return type>
3 is
4   [declaration of variables]
5 begin
6   -- sequential statement (but not wait)
7 end function <name>;
```

An example is a parity generator:

Listing 2: An odd parity generator function.

```
1 function pargen(avect: std_ulogic_vector)
2   return std_ulogic
3 is
4   variable po_var : std_logic;
5 begin
6   po_var := '1';
7   for i in avect'range loop
8     if avect(i) = '1' then
9       po_var := not po_var;
10    end if;
11  end loop;
12  return po_var;
13 end function pargen;
```

In testbenches it is common to see procedures. They differ for function as they can have multiple inputs and *multiple output*. Because of this they in practice are usually not synthesizable. The syntax is similar to functions:

```
1 procedure <name> ([list of arguments with
2   direction]) is
3   [declaration of variables]
4 begin
5   -- sequential statement
6 end procedure <name>;
```

With *list of arguments with direction* it is meant an expression like `a, b : in real; w : out real`, similar to the arguments of `port`.

4.5 Arrays and records

To efficiently use `generate` statement, such as in listing 1, we need array types. Arrays types (fields) of other types are defined with the following syntax.

```
1 type <name> is array (<upper
2   limit> downto <lower limit>) of <base type>;
```

For example to complete listing 1, we create 1 by 1 matrices.

```

1 constant nr_digits : integer := 3;
2 type bcd_array_type is
3   array (0 to nr_digits -1)
4   of std_ulogic_vector(3 downto 0);
5 type bcd_array_type is
6   array (0 to nr_digits -1)
7   of std_ulogic_vector(6 downto 0);

```

While all arrays elements must have the same underlying type, *records* allow for different types to be combined together. For example:

```

1 type memory_access is record
2   address : integer range 0 to
3     ↪ address_max -1;
4   mem_block : integer range 0 to 3;
5   data : std_ulogic_vector(word_width -1
6     ↪ downto 0);
7 end record;

```

4.6 Packages

To declare your own packages, the syntax is rather easy:

```

1 <library and / or use statements>
2 package <name> is
3   [declarations]
4 end <package name>;

```

And possibly in another file the implementation is give with:

```

1 package body <name> is
2   <list of definitions>
3 end package body <name>;

```

In practice it is common to see for example a configuration package, that contains all constants for the project. For example if we were to put the function `pargen` from listing 2 we could do:

```

1 package parity_helpers is
2   constant nibble : integer;
3   constant word   : integer;
4   function pargen(avect :
5     ↪ std_ulogic_vector) return
6     ↪ std_ulogic;
7 end package parity_helpers;
8
9 package body parity_helpers is
10  -- functions
11  function pargen(avect :
12    ↪ std_ulogic_vector)
13    return std_ulogic
14  is From listing 2
15  end function pargen;
16  -- instantiation of variables
17  constant nibble : integer := 4;
18  constant word   : integer := 8;
19 end package body parity_helpers;

```

And later use it with `use work.parity_helpers.all`.