DigDes: Digital Design

Naoki Pross – `naoki.pross@ost.ch`

Spring Semester 2021

Contents

License

1 Development model

2 Hardware

3 VHSIC Hardware Description Language (VHDL)

3.1 Basic syntax and identifiers

In VHDL an identifier is a case insensitive string composed of `A-Z a-z 0-9 _` that

- is not a keyword,
- does not start with a number or `_`,
- does not have two or more `_` in a row.

Expressions are terminated by a semicolon `;`. Two dashes in a row cause the rest of the line to be interpreted as a comment.

```
1 expression; -- comment
```

3.2 Structure and Libraries

The VHDL code is organized into *libraries* declared with the `library` keyword. The library of your code is called `work`, standard features (`bit`, `integer`, …) are found in `std`, and IEEE standard parts are in `ieee`. `work` and `std` are always implicit and must not be declared.

```
1 library ⟨library name⟩;
```

Once declared a library is composed of *packages*, which can contain elements (constants, entities, …). To access the elements the syntax is

```
1 ⟨library⟩.⟨package⟩.⟨element⟩;
```

To avoid having to write a long name every time it is possible to import names using

```
1 use ⟨library⟩.⟨element or all⟩;
2 use ⟨library⟩.⟨package⟩.⟨element or all⟩;
```

3.3 Entities and Architectures

In VHDL the concept of *entity* describes a black box of which only inputs and outputs are known. The internals of an entity are described through an *architecture*. There can be multiple architectures for a single entity.
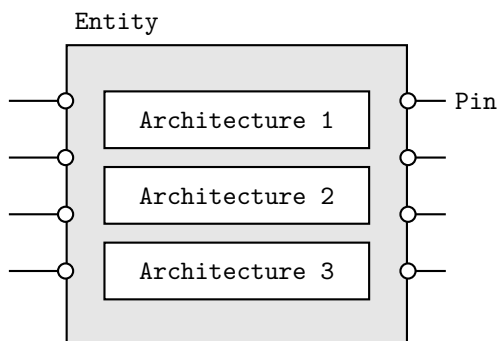
Entities are declared with `port()` that may contain a list of pins. Pins have a mode that can be `in` input (only LHS[1]), `out` output (only RHS[2]), `inout` bidirectional or `buffer` that can stay both on LHS and RHS. The usage of the latter is discourareged in favour of an internal signal.

```
1 entity ⟨name⟩ is
2   port(
3     ⟨pin⟩ : ⟨mode⟩ ⟨type⟩;
4   );
5 end ⟨name⟩;
```

Architectures are normally named after the design model, examples are `behavioral`, `structural`.

```
1 architecture ⟨name⟩ of ⟨entity⟩ is
2   -- declare used variables, signals and
        ↪ component types
3 begin
4   -- concurrent area
5 end [name];
```

3.4 Electric types

VHDL provides some types such as

- `boolean` true or false,
- `bit` 0 or 1,
- `bit_vector` one dimensional array of bits,
- `integer` 32-bit binary representation of a value.

From external libraries other types are available:

- `std_logic` advanced logic with 9 states,
- `std_ulogic` same as the previous but *unresolved*.

The above are from the `ieee.std_logic_1164` library, and can take the values described in the following table.

Value	Meaning	Usage
U	Uninitialized	In the simulator
X	Undefined	Simulator sees a bus conflict
0	Force to 0	Low state of outputs
1	Force to 1	High state of outputs
Z	High Impedance	Three state ports
W	Weak Unknown	Simulator sees weak a bus conflict
L	Weak Low	Open source outputs with pull-down resistor
H	Weak High	Open drain output with pull-up resistor
-	Don't care	Allow minimization

For the *resolved* types, i.e. `std_logic` types, when a signal is multiply driven the conflict is resolved according to the table below. Unresolved types will give a synthesization error.

---

[1]Left hand side
[2]Right hand side

	U	X	0	1	Z	W	L	H	-
U	U	U	U	U	U	U	U	U	U
X	U	X	X	X	X	X	X	X	X
0	U	X	0	X	0	0	0	0	X
1	U	X	X	1	1	1	1	1	X
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	W	W	X
L	U	X	0	1	L	W	L	W	X
H	U	X	0	1	H	W	W	H	X
-	U	X	X	X	X	X	X	X	X

A good example is a tri-state bus:

```
1  architecture tristate of buscontrol is
2  begin
3    bus_read: inp <= bus_io;
4
5    bus_write: process(enable, oup)
6    begin
7      bus_io <= (others => 'Z');
8      if enable = '1' then
9        bus_io <= oup;
10     end if;
11   end process;
12 end architecture tristateout;
```

3.5 Declarations

Before a `begin` – `end` block, there is usually a list of declarations. A self evident examples are *constants*.

```
1  constant ⟨name⟩ : ⟨type⟩ := ⟨value⟩;
```

Next, *signals* and *variables*. Signals is are wires, they can only be connected and do not have an initial state. Variables can be assigned like in software, but can cause the synthesization of an unwanted D-Latch.

```
1  signal ⟨name⟩, [name, …] : ⟨type⟩;
2
3  variable ⟨name⟩, [name], […] : ⟨type⟩;
4  variable ⟨name⟩ : ⟨type⟩ := ⟨expression⟩;
```

For the hierarchical designs, when external entities are used, they must be declared as components. The `port()` expression must match the entity declaration.
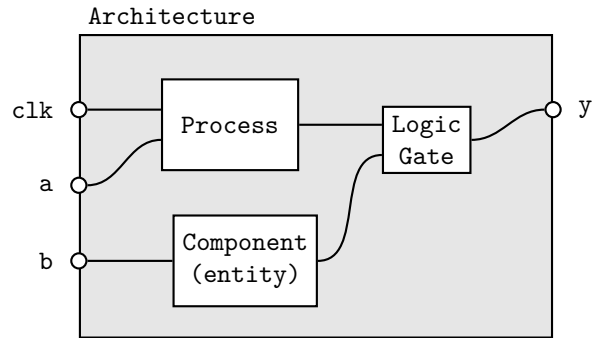
```
1  component ⟨entity name⟩ is
2    port(
3      [list of pins]
4    );
5  end component;
```

For entities with multiple architectures, it is possible to choose which architecture is used with the following expression.

```
1  for ⟨label or all⟩: use entity ⟨library⟩.
     ↪ ⟨entity⟩(⟨architecture⟩);
```

3.6 Concurrent Area

In the architecture between `begin` and `end`, the expressions are *not* read sequentially, everything happens at the same time. Statements inside the concurrent area optionally have a label.

```
1  [label]: ⟨concurrent statement⟩;
```

In the concurrent area signals, components and processes can be used to create a logic.

3.6.1 Signal assignment and simple gates

Signals are assigned using `<=`.

```
1  [label]: ⟨signal⟩ <= ⟨expression⟩;
```

Simple logic functions such as `not`, `and`, `or`, `xor`, etc. can be used.

```
1  y <= (a and s) or (b and not(s));
```

3.6.2 Aggregates

For vector types it is possible to create a value out of multiple signals.

```
1  ⟨vector⟩ <= (
2    ⟨index⟩   => ⟨source or value⟩,
3    ⟨index⟩   => ⟨source or value⟩,
4    [others] => ⟨source or value⟩
5  );
```

```
1  -- declaration
2  signal data : bit_vector(6 downto 0);
3  signal a, b : bit;
```

```
1  -- concurrent
2  data = (1 => a, 0 => b, others => '0')
```

3.6.3 Selective and conditional assignment

Higher level conditions can be written in two ways.

```
1  -- using when
2  [label]: y <= ⟨source⟩ when ⟨condition⟩ else
3         ⟨source⟩ when ⟨condition⟩ else
4         ⟨source⟩ when ⟨condition⟩;
```

```
1  -- using with
2  [label]: with ⟨signal⟩ select ⟨dest⟩ <=
3    ⟨source⟩ when ⟨value⟩,
4    ⟨source⟩ when ⟨value⟩,
5    ⟨source⟩ when others;
```

3.6.4 Components

External components that have been previously declared can be used with the `port map(`⟨*assignments*⟩`)` syntax. For example:

```
1 -- declaration
2 component flipflop is
3   port(
4     clk, set, rst : in  std_ulogic,
5     Q, Qn         : out std_ulogic
6   );
7 end component flipflop;
8
9  signal clk_int, a, b : in  std_ulogic;
10 signal y, z          : out std_ulogic;
```

```
1 -- concurrent
2 u1: flipflop
3   port map(
4     clk => clk_int,
5     set => a,
6     rst => b,
7     Q   => y,
8     Qn  => z
9   );
```

3.6.5 Processes

For more sophisticated logic VHDL offers a way of writing sequential statements called *process*.

```
1 [label]: process ([sensitivity list])
2 -- declarations
3 begin
4   -- sequential statements
5 end process;
```

Processes have a *sensitivity list* that can be empty. When a signal in the sensitivity list changes state, the process is executed. With an empty sensitivity list, the process runs continuously. In the declaration, everything from §3.5 applies. For the sequential statements, the following applies:

- Neither selective (`with`) nor conditional (`when`) should be used. They are replaced with new sequential constructs (`if` and `case`).
- Signal assignments (with `<=`) change their value *only at the end of the process*.
- Variables on the other hand change as soon as they are assigned (with `:=`).

And for good practice:

- Before any `if` or `case` default values should be assigned.
- Any signal on the RHS should be in the sensitivity list.
- Processes with empty sensitivity lists should only be used for simulations.

The sequential replacements for `with` and `when` are in the listings below.

```
1 if ⟨condition⟩ then
2   -- sequential statements
3 elsif ⟨condition⟩ then
4   -- sequential statements
5 else
6   -- sequential statements
7 end if;
```

```
1 case ⟨expression⟩ is
2   when ⟨choice⟩ =>
3     -- sequential statements
4   when ⟨choice⟩ =>
5     -- sequential statements
6   when others =>
7     -- sequential statements
8 end case;
```

Processes can detect *attributes* of signals. Typically it is used for clocks. There are also other attributes such as `s'stable(t)`.

```
1  process (clk)
2  begin
3    -- rising edge
4    if clk'event and clk = '1' then
5      ... end if;
6    if rising_edge(clk) then
7      ... end if;
8
9    -- falling edge
10   if clk'event and clk = '0' then
11     ... end if;
12   if falling_edge(clk) then
13     ... end if;
14 end process;
```
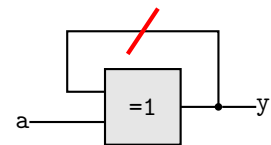
3.7 Custom and arithmetic types

It is possible to create custom types, usually to create state machines.

```
1 type ⟨name⟩ is (⟨identifier⟩, ⟨identifier⟩, …);
```

3.8 Pitfalls and RTL model

Coming from a programming language, a common pitfall is to write something like

```
1 -- wrong!!!
2 y <= y xor a;
```


but this will be synthesised into an oscillating circuit, that must be avoided at all costs. The correct way is to have a memory for the next state, with a logic separated into combinatorial and sequential parts.

```
1 -- combinatorial
2 y_next <= y xor a;
3 -- sequential
4 process (clk)
5 begin
6   if rising_edge(clk) then
7     y <= y_next;
8   end if;
9 end process;
```

This method is known as *register transfer level* design.

4 State Machines

Mealey

Moore

Medwedjew
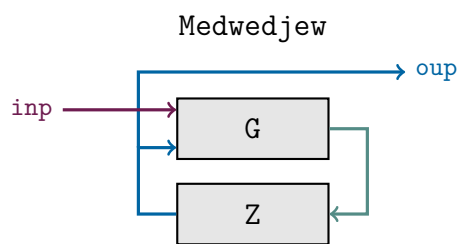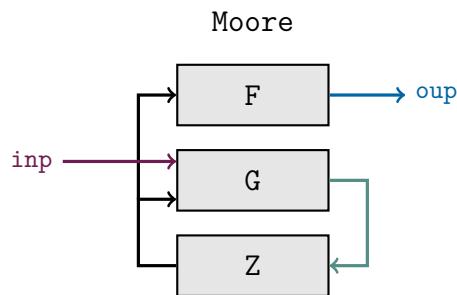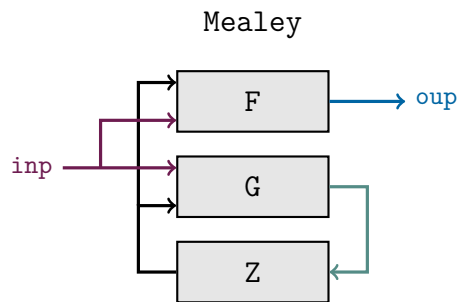
4.1 Encoding the state

For Mealey and Moore machines it is typical to write:

```
1 type state_type is (st_rst, st_a, st_b,
      ↪ st_c, ...);
2 signal present_state, next_state :
      ↪ state_type;
```

The encoding of the state is left to the synthesizer or can be configured in the graphical interface of the tool. If a custom encoding is required (Medwedjew), adding the following generates a custom encoding.

```
1 attribute enum_encoding : string;
2 attribute enum_encoding of state_type:
3   type is "0001 0010 0100 ...";
```

Or an equivalent approach is to use a vector subtype and constants.

```
1 subtype state_type is bit_vector(3 downto
      ↪ 0);
2
3 constant st_rst : state_type := "0001";
4 constant st_a   : state_type := "0010";
5 constant st_b   : state_type := "0100";
6 ...
7
```

```
8 signal present_state, next_state :
      ↪ state_type;
```

4.2 Updating the state register (Z)

```
1 register_logic: process (clk, rst)
2 begin
3   -- asynchronous reset
4   if rst = '1' then
5     present_state <= st_rst;
6
7   -- clock
8   elsif rising_edge(clk) then
9     present_state <= next_state;
10  end if;
11 end process;
```

4.3 Updating the state (G)

```
1 next_state_logic:
2 process (present_state, [inputs])
3 begin
4   -- default value
5   next_state <= state_rst;
6
7   case present_state is
8     when st_rst =>
9       -- reset state logic
10      next_state <= ⟨state⟩;
11
12    when st_a =>
13      -- logic using inputs
14      next_state <= ⟨state⟩;
15
16    ...
17    when others => null;
18  end case;
19 end process;
```

4.4 Updating the output (F)

Mealey

```
1 output_logic:
2 process (present_state, ⟨inputs⟩)
3 begin
4   -- logic with state and inputs
5   ⟨output⟩ <= ⟨expression⟩;
6 end process;
```

Moore

```
1 output_logic: process (present_state)
2 begin
3   case present_state is
4     when st_rst =>
5       ⟨output⟩ <= ⟨value⟩;
6
7     ...
8   end case;
9 end process;
```

Medwedjew

```
1 output_logic: ⟨output⟩ <= present_state;
```

5 Testing

To simulate a digial circuit it is possible to write test benches using VHDL. The code in this section may no longer be synthetisable, and is usually written by a *test designer*.

5.1 Simulator

VHDL simulates digital systems using *delta cycles*.

5.2 Transport delay

To model a time delay of a signal there are two ways:

```
1 y <= transport ⟨expression⟩ after ⟨time⟩;
2 y <= inertial ⟨expression⟩ after ⟨time⟩;
```

When `transport` is used, the output changes only exactly after the specified time, the simulator simply waits. With `inertial`, the output is also delayed, but only if the input lasts more than the specified time. This means that for example with a time of `10 ns` a pulse of `5 ns` will be ignored. This is much more typical and realistic, thus when unspecified, `after` is interpreted as `inertial ... after`.

```
1 y <= ⟨expression⟩ after ⟨time⟩;
```

5.3 Generate stimuli

Simple stimuli (signals) are generated using processes. For example a clock signal done in three ways:

```
1 -- declaration
2 constant f : integer := 1000;
3 constant T : time    := 1 sec/f;
4 signal clk0, clk1, clk2 : std_ulogic;
```

```
1 -- concurrent
2 clock0: process
3 begin
4   clk <= '1'; wait for (T/2);
5   clk <= '0'; wait for (T/2);
6 end process;
7
8 clock1: process
9 begin
10   clk1 <= '1';
11   loop
12     wait for (T/2);
13     clk1 <= not clk1;
14   end loop;
15 end process;
16
17 -- lazy way
18 clock2: clk2 <= not clk2 after (T/2);
```

One time stimuli can be modelled using the following expression. Note that the time is absolute.

```
1 tb_sig <= '0',
2   '1' after 20 ns,
3   '0' after 30 ns, -- 10 ns later
4   ⟨value⟩ after ⟨time⟩;
```

Repeating sequences can be created using processes.

```
1 sequence: process
2 begin
3   tb_sig <= '0';
4   wait for 20 ns;
5   tb_sig <= '1';
6   wait for 10 ns;
7   ...
8 end process;
```

For loops are also available, and can be synthesised if they run over a finite range.

```
1 [label]: for ⟨parameter⟩ in ⟨range⟩ loop
2   -- sequential statements
3 end loop [label];
```

A concrete example:

```
1 -- declaration
2 constant n : integer := 3;
3 signal a, b : std_ulogic_vector(n-1
      ↪ downto 0);
```

```
1 -- sequential
2 for i in 0 to 2**n -1 loop
3   a <= std_ulogic_vector(
4         to_unsigned(i, n));
5   for k in 0 to 2**n -1 loop
6     b <= std_ulogic_vector(
7           to_unsigned(k, n));
8   end loop;
9 end loop;
```

5.4 Assertions

Assertions are used write tests to check that a signal is in the correct state.

```
1 [label]: assert ⟨condition⟩ report ⟨string⟩
      ↪ severity ⟨severity⟩;
```

The `report` and `severity` are optional but strongly advised. The severity can take one of 4 values: `note`, `warning`, `error`, `failure`. Simulations can be configured to stop in when an error of the desired severity occurrs. An example:

```
1 assert (tb_y = '0') report "error at
      ↪ vector 11" severity error;
```