DigDes: Digital Design

Naoki Pross – naoki.pross@ost.ch

Spring Semester 2021

Contents

License

1 Development model and Hardware

2 VHSIC Hardware Description Language (VHDL)

2.1 Basic syntax and identifiers

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

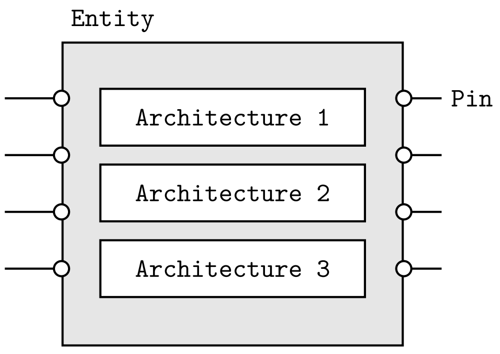
- is not a keyword,
- does not start with a number or `_`,
- does not have two or more `_` in a row.

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

```
expression; -- comment
```

2.2 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), `out` output (only RHS), `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.

```
entity ⟨name⟩ is
  port(
    ⟨pin⟩ : ⟨mode⟩ ⟨type⟩;
  );
end ⟨name⟩;
```

Architectures are normally named after the design model, example are `behavioral`, `structural`, `selective`, etc.

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

2.3 Electric types and Libraries

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`

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

2.4 Declarations

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

```
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.

```
signal ⟨name⟩, [name, ...] : ⟨type⟩;

variable ⟨name⟩, [name], [...] : ⟨type⟩;
variable ⟨name⟩ : ⟨type⟩ := ⟨expression⟩;
```

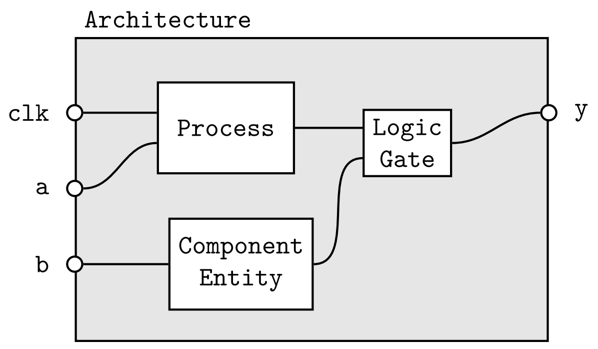
For the hierarchical designs, when external entities are used, they must be declared as components. The `port()` expression must match the entity declaration.

```
component ⟨entity name⟩ is
  port(
    [list of pins]
  );
end component;
```

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

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

2.5 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.

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

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

2.5.1 Signal assignment and simple gates

Signals are assigned using `<=`.

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

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

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

2.5.2 Aggregates

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

```
⟨vector⟩ <= (
  ⟨index⟩  => ⟨source or value⟩,
  ⟨index⟩  => ⟨source or value⟩,
  [others] => ⟨source or value⟩
);
```

```
-- declaration
signal data : bit_vector(6 downto 0);
signal a, b : bit;
-- concurrent
data = (1 => a, 0 => b, others => '0')
```

2.5.3 Selective and conditional assignment

Higher level conditions can be written in two ways.

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

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

2.5.4 Components

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

```
-- declaration
component flipflop is
  port(
    clk, set, reset : in  std_ulogic,
    Q, Qn           : out std_ulogic
  );
end component flipflop;

signal clk_int, a, b : in  std_ulogic;
signal y, z          : out std_ulogic;
```

```
-- concurrent
u1: flipflop
  port map(
    clk   => clk_int,
    set   => a,
    reset => b,
    Q     => y,
    Qn    => z
  );
```

2.5.5 Processes

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

```
[label]: process ([sensitivity list])
-- declarations
begin
  -- sequential statements
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 §2.4 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.

```
if ⟨condition⟩ then
  -- sequential statements
elsif ⟨condition⟩ then
  -- sequential statements
else
  -- sequential statements
end if;
```

```
case ⟨expression⟩ is
  when ⟨choice⟩ =>
    -- sequential statements
  when ⟨choice⟩ =>
    -- sequential statements
  when others =>
    -- sequential statements
end case;
```

Processes can detect *events* of signals. Typically it is used for clocks.

```
process (clk)
begin
  -- rising edge
  if clk'event and clk = '1' then
    ... end if;
  if rising_edge(clk) then
    ... end if;

  -- falling edge
  if clk'event and clk = '0' then
    ... end if;
  if falling_edge(clk) then
    ... end if;
end process;
```

2.6 Custom and arithmetic types

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

```
type ⟨name⟩ is (⟨identifier⟩, ⟨identifier⟩, ...);
```

3 State Machines

There are 3 types of state machines.

3.1 Encoding the state

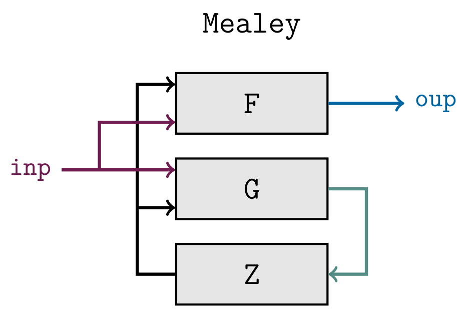
This is typical for Mealey and Moore machines.

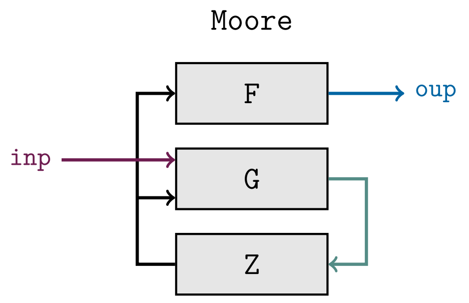
```
type state_type is (st_rst, st_a, st_b,
  ↪ st_c, ...);
signal present_state, next_state :
  ↪ state_type;
```

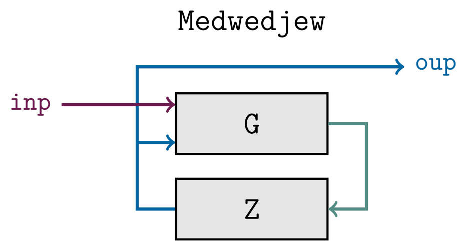
The encoding of the state is left automatically to the synthesizer or configured in the graphic interface of the tool. If a custom encoding is required (Medwedjew), adding the following generates a custom encoding.

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

Or alternatively a completely different approach is using a vector type.







```
subtype state_type is bit_vector(3 downto
  ↪ 0);

constant st_rst : state_type := "0001";
constant st_a   : state_type := "0010";
constant st_b   : state_type := "0100";
...

signal present_state, next_state :
  ↪ state_type;
```

3.2 Updating the state register (Z)

```
register_logic: process (clk, rst)
begin
  -- asynchronous reset
  if rst = '1' then
    present_state <= st_rst;

  -- clock
  elsif rising_edge(clk) then
    present_state <= next_state;
  end if;
end process;
```

3.3 Updating the state (G)

```
next_state_logic:
process (present_state, [inputs])
begin
  -- default value
  next_state <= state_rst;

  case present_state is
    when st_rst =>
      -- reset state logic
      next_state <= ⟨state⟩;

    when st_a =>
      -- logic using inputs
      next_state <= ⟨state⟩;

    ...
    when others => null;
  end case;
end process;
```

3.4 Updating the output (F)

Mealey

```
output_logic:
process (present_state, ⟨inputs⟩)
begin
  -- logic with state and inputs
  ⟨output⟩ <= ⟨expression⟩;
end process;
```

Moore

```
output_logic: process (present_state)
begin
  case present_state is
    when st_rst =>
      ⟨output⟩ <= ⟨value⟩;

    ...
  end case;
end process;
```

Medwedjew

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

4 Testbench