`DigDes`: Digital Design

Naoki Pross – `naoki.pross@ost.ch`

Spring Semester 2021

Contents

License

1 Development model

The workflow for the development is show in figure 1. In the Gajski-Kuhn Y-model has 3 axis for the perspectives of the product. It is typical to start from the behavioral axis, by treating the systems as a black-box, and then to jump back and forth between the other axis while gravitating towards the origin (project goal).

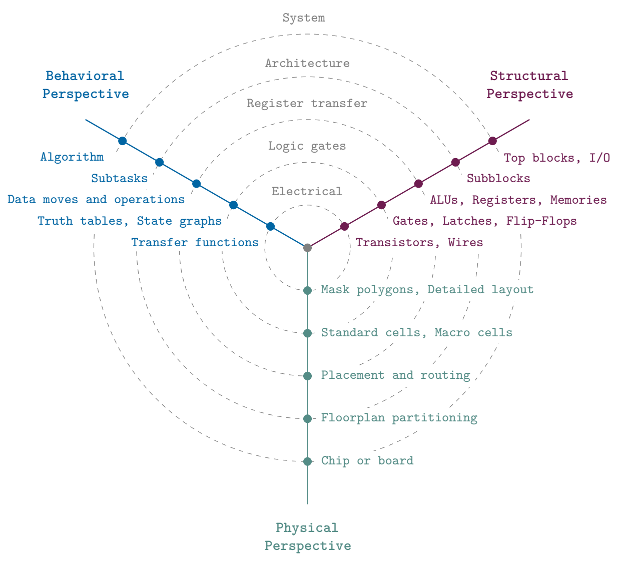


Figure 1: Gajski-Kuhn Y-chart.

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 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.

```
library ⟨library name⟩;
```

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

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

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

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

2.3 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.

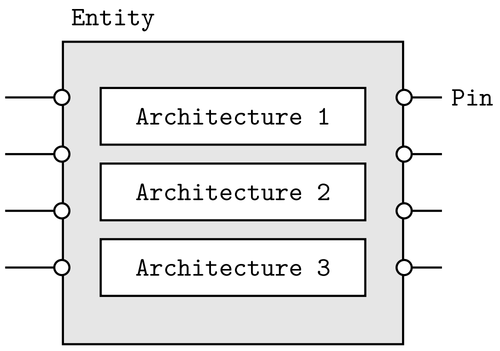


Figure 2: An entity is a black box, that can have multiple architectures.

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.

```
entity ⟨name⟩ is
  port(
    ⟨pin⟩ : ⟨mode⟩ ⟨type⟩;
    [more pins];
    ⟨pin⟩ : ⟨mode⟩ ⟨type⟩
  );
end entity [name];
```

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

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

2.4 Type system

2.4.1 Electric types

VHDL provides some types such as

- `boolean` true or false,
- `bit` 0 or 1,

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

- `bit_vector` one dimensional array of bits,
- `integer` 32-bit binary representation of a value.

From external (standard) 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 table 1. For the

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

Table 1: Possible values for `std_logic` signals.

resolved types, i.e. `std_logic` types, when a signal is multiply driven the conflict is resolved according to table 2. Unresolved types will give a synthesization error. A good example is a tri-state bus:

```
architecture tristate of buscontrol is
begin
  bus_read: inp <= bus_io;

  bus_write: process(enable, oup)
  begin
    bus_io <= (others => 'Z');
    if enable = '1' then
      bus_io <= oup;
    end if;
  end process;
end architecture tristateout;
```

	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

Table 2: Resolution table when a `std_logic` signal is multiply driven.

2.4.2 Arithmetic types

For arithmetic operations two more types `signed` and `unsigned` (as well as their unresolved equivalents `u_signed` and `u_unsigned`) can be imported (together with many others for ex. `natural`) from the library `ieee.numeric_std`. Arithmetic types support the operations in table 3.

Syntax	Operator	Note
`+`	Addition	
`-`	Subtraction	
`abs()`	Absolute value	
`*`	Multiplication	
`/`	Division	Typically not available
`**`	Power	Only powers of 2
`mod`	Modulo	Only modulo of 2^k
`rem`	Remainder	Only of division by 2^k
`=`	Equality	
`/=`	Inequality	
`<`, `>`	Lower, greater	
`<=`, `>=`	Lower, greater or equal	Same the assignment operator, however it is always clear from context.

Table 3: Arithmetic operations from the `numeric_std` library.

2.4.3 Array type

Arrays types (fields) of other types can be define with the following.

```
type ⟨name⟩ is array (⟨upper
    limit⟩ downto ⟨lower limit⟩) of ⟨base type⟩;
```

2.4.4 Custom enumeration types

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

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

2.4.5 Physical types

For variables that represent physical dimensions it is possible to create values with units with the following:

```
type ⟨name⟩ is range ⟨min⟩ to ⟨max⟩
units
  ⟨base unit⟩;
  [multiples of base unit];
end units;
```

for example:

```
type CAPACITANCE is range 0 to 1E30
units
  pf;
  nf = 1000 pf;
  uf = 1000 nf;
  mf = 1000 uf;
end units;
```

2.4.6 Reisizing vectors

VHDL has a function

```
function resize(arg: signed; new_size:
    ↪ natural) return signed;
```

that allow to reisze vector types. When resizing a vector of signed type to a higher number of bits the `resize` function cleverly fills the extra bits 1s or 0s to not mess up the two's complement. Toghether with the `resize` function an often used feature is the `'length` attriubte, that returns the size (in bits) of the identifier.

```
y <= resize(a, y'length);
```

2.4.7 Type casting and conversion

When two signals have the same underlying type it is always possible to perform a *type cast* using the following syntax.

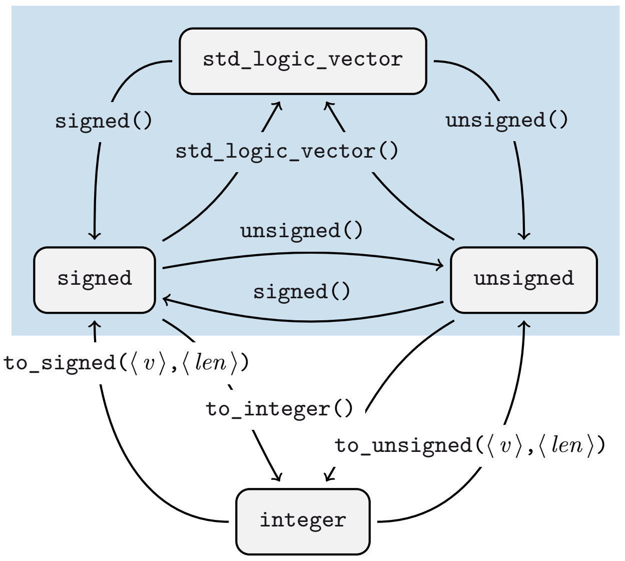
```
⟨destination⟩ = ⟨type name⟩(⟨source⟩);
```

For example:

```
architecture behavoral of cast_example
  signal a_int, b_int :
    std_logic_vector(3 downto 0);
  signal s_int : unsigned(3 downto 0);
begin
  s_int <= unsigned(a_int)
           + unsigned(b_int);
end architecture;
```

When the conversion is between signals with a different underlying type it is a (potentially lossy) *type conversion*. The syntax for a conversion is:

```
⟨destination⟩ = to_⟨type name⟩(⟨source⟩);
```



2.5 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⟩);
```

2.6 Concurrent Area

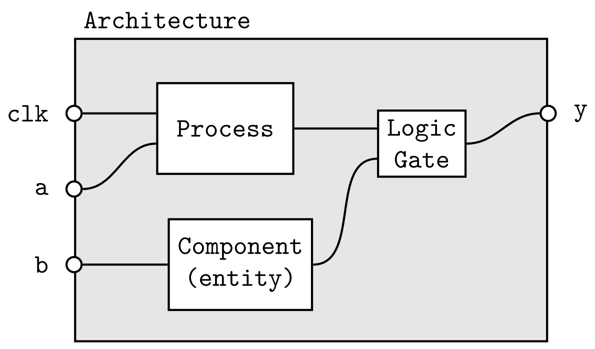


Figure 3: In the concurrent area statements are not interpreted sequentially.

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.

2.6.1 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));
```

2.6.2 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')
```

2.6.3 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;
```

2.6.4 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, rst : 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: component flipflop
  port map(
    clk => clk_int,
    set => a,
    rst => b,
    Q   => y,
    Qn  => z
  );
```

2.6.5 Processes

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

```
[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 §2.5 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 next* `wait for` *statement or 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 *attributes* of signals. Typically it is used for clocks. There are also other attributes such as `s'stable(t)`.

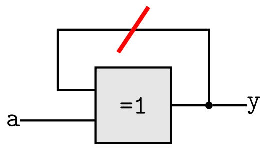
```
process (clk)
begin
  -- rising edge
  if clk'event and clk = '1' then
    ... end if;
  if rising_edge(clk) then
    ... end if;

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

2.7 Pitfalls and RTL model

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

```vhdl
-- wrong!!!
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.

```vhdl
-- combinatorial
y_next <= y xor a;
-- sequential
process (clk)
begin
  if rising_edge(clk) then
    y <= y_next;
  end if;
end process;
```

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

2.8 Generic Parameters

Sometimes a group of components have a very similar structure, so instead of rewriting multiple similar interfaces it is desirable to have *parameters* and a *generic* entity, for example in the case of a binary counter's range. To solve the problem using signals with conditional statements would generate unnecessary hardware, while constants cannot change the entity's port. Thus there is a syntax:

```vhdl
generic(
  ⟨param name⟩ : ⟨type⟩ := ⟨initial value⟩;
  [more parameters];
  ⟨param name⟩ : ⟨type⟩ := ⟨initial value⟩
);
```

that has affects at *synthesization time*.

2.8.1 Generic entity and declaration

Entities are parametrized as follows.

```vhdl
entity ⟨name⟩ is
  generic(⟨parameters⟩);
  port(⟨pins⟩);
end entity ⟨name⟩;
```

For example:

```vhdl
entity counter is
  generic(CNT_MAX : natural := 127);
  port(
    clk, rst, ena : in std_logic;
    -- adjust to a power of 2
    count : out std_logic_vector(
      (natural(ceil(
        log2(real(CNT_MAX +1)))) -1)
        downto 0);
end entity;
```

And in the architecture it is possible to access generic values in a similary way. Another example is a clock divider.

```vhdl
entity clockdivider is
  generic(DIV_FACTOR : natural := 128);
  port(...);
end entity;

architecture RTL of clockdivider is
  signal cnt, cnt_next : natural range 0
      ↪ to (DIV_FACTOR -1);
  ...
```

2.8.2 Generic mapping (Concurrent Area)

To map a generic entity into a structural design the syntax is extended accordingly with `generic map()`.

```vhdl
-- definition
component ⟨generic entity⟩ is
  generic(⟨parameters⟩);
  port(⟨pins⟩);
end component;
```

```vhdl
[label]: component ⟨generic component⟩
  generic map(
    ⟨parameter⟩ => ⟨constant or parameter⟩,
    ...
  );
  port map(
    ⟨pin⟩ => ⟨signal or pin⟩,
    ...
  );
```

3 State Machines

3.1 Encoding the state

For Mealey and Moore machines it is typical to write:

```vhdl
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 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.

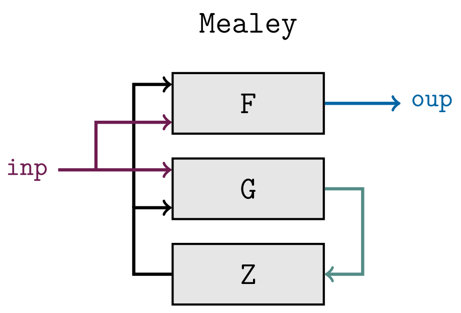
```vhdl
attribute enum_encoding : string;
attribute enum_encoding of state_type:
  type is "0001 0010 0100 ...";
```

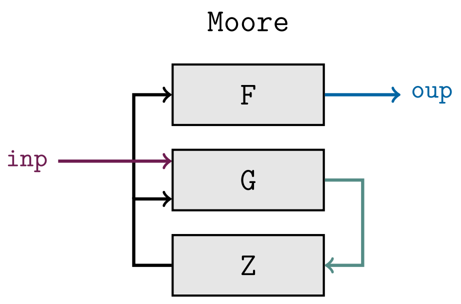
Or an equivalent approach is to use a vector subtype and constants.

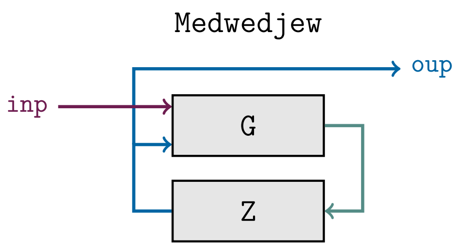
```vhdl
subtype state_type is bit_vector(3 downto
    ↪ 0);

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 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*.

4.1 Simulator

VHDL simulates digital systems using *delta cycles*.

4.2 Transport delay

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

```
y <= transport ⟨expression⟩ after ⟨time⟩;
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`.

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

4.3 Generate stimuli

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

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

```
-- concurrent
clock0: process
begin
  clk <= '1'; wait for (T/2);
  clk <= '0'; wait for (T/2);
end process;

clock1: process
begin
  clk1 <= '1';
  loop
    wait for (T/2);
    clk1 <= not clk1;
  end loop;
end process;

-- lazy way
clock2: clk2 <= not clk2 after (T/2);
```

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

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

Repeating sequences can be created using processes.

```
sequence: process
begin
  tb_sig <= '0';
  wait for 20 ns;
  tb_sig <= '1';
  wait for 10 ns;
  ...
end process;
```

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

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

A concrete example:

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

```
-- sequential
for i in 0 to 2**n -1 loop
  a <= std_ulogic_vector(
          to_unsigned(i, n));
  for k in 0 to 2**n -1 loop
    b <= std_ulogic_vector(
            to_unsigned(k, n));
  end loop;
end loop;
```

4.4 Assertions

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

```
[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:

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