VHDL notes

Fabien Le Mentec, fabien.lementec@gmail.com

version: 3524f8e

Abstract

Set of VHDL related notes. It addresses topics ranging from coding conventions, verification, synthesis, optimisation, reusability and documentation. Some notes are vague while others are quite specific to tools or target platforms. Also, it addresses a wide audience, so some materials may seem obvious to the reader, depending on her background.

Contents

1 Unconstrained types

topics: reusability, documentation, verification

When not explicitly specified by the developer, a type length is deduced during component instantiation. This favors component reusability by letting the user decide of the type length according to its particular needs.

For instance, *count* range is unconstrained in the following component declaration:

```
component my_component
port
(
 ...
 count: in unsigned;
 ...
);
```

The actual range is deduced during instantiation:

```
signal count: unsigned(7 downto 0);

work.my_component
port map
(
 ...
 count => count
 ...
);
```

One important issue with unconstraint types is that a component user may inadvertently use types that are larger than required, possibly leading to unecessary large resource instantiation. Documentation is a good tool to solve this kind of issue. Also, assertions can be used to check for degenerate cases:

```
component my_component
port
(
  ...
 -- WARNING
 -- hardware comparator infered in subsequent logic
 -- use appropriate length
 count: in unsigned;
  ...
);
end component;
```

If you want to make sure the length fits within a given range, use assertion in the component entity definition:

```
entity my_component
port
(
  ...
 count: in unsigned;
  ...
);
end my_component;

architecture my_component_rtl of my_component is
begin
...
assert (count'length <= 16)
report "invalid counter length"
severity failure;
...
end my_component_rtl;
```

2 Type attributes

Use type attribute as much as possible, esp. type'length and type'range

3 Generics instead of package constants

topics: reusability

Often, a component parameter can be set either using generic or package constants. Using package constants forces the user to modify your package. On the other hand, generics let the user specializes the component without modifying any existing source.

4 Per component test benches

topics: simulation

Per component test benches generally requires less code than project wide ones. It makes them easier to maintain, and encourages the developer to write self contained components. Also, it makes simulation run faster.

5 Hardware resource inference

topics: synthesis

Usually, a VHDL developer does not explicitely indicate what hardware resource to use to implement logic. The synthetiser deduces that from its source code understanding (ie. signal netlist and operations). This process is known as inference.

Inference is very sensitive to the way code is written. For instance, the use of an additional signal to reset a shift register may prevent the synthetiser to infer a hardware shift register.

Thus, VHDL developers try as much as possible to write code in a standard way, that is known to be well understood by the synthetiser.

6 Explicit resource instantiation

topics: synthesis

TODO: wip

Non portable but sure to instanciate the right resource.

7 Reset signals

related notes: 11

Avoid reset signals. If not possible, make reset synchronous.

TODO: explain why

8 Shift registers inference

topics: synthesis

TODO: wip

XILINX FPGAs have hardware resources to implement shift registers.

9 Assertions

topics: verification

TODO: wip

Use assertion to check data type lengths when unconstraints arrays

10 Test benches as documentation

topics: documentation

A component developer should consider test benches an important part of the documentation since they are used as reference materials by the component user. Thus, test benches should be up to date, clearly written and well documented. If possible, they should cover different use cases, without flooding the user with unrequired contents.

11 Writing synchronous processes

topics: synthesis

There is one standard way of writing synchronous process:

```
process(clk, rst)
begin
 if rising_edge(clk) then
  if rst = '1' then
  else
  end if;
 end if;
end process;
```

Another way which is synthetizable:

```
process
begin
 wait until rising_edge(clk);

 if rst = '1' then
 end if;

end process;
```

Since the **wait** statement must come first, all the signal are synchronous, esp. the reset. Also, this convention results in a somewhat clearer code.

12 Clocking

TODO: wip

Clear convention about how data passed to/from a component are clocked. by default, clocked using the component domain. idem for latching.

13 Appropriate typing

topics: verification, documentation

Use the most specialized types (unsigned, boolean ...) and sizes early in the design hierarchy. It avoids further casting and simplifies the code. It acts as documentation since the reader deduces information from the type itself. For instance, an unsigned counter tells it can not be negative. Typing also improve static time checks.

14 Component directory structure

topics: reusability, documentation

Having a clear, self contained directory structure is helpful for both the user and the developer. I opted for the following one, that simple but fits most of the cases:

```
my_component/
 src/
  my_component_pkg.vhd
  my_component_rtl.vhd
  ...
 sim/
  common/
   main_tb.vhd
  isim/
   isim.tcl
   isim.prj
   isim.sh
  modelsim/
   ...
 syn/
  ise/
   xc7k325t.ucf
   xc7vx485t.ucf
  vivado/
   ...
 doc/
  my_component.pdf
```

Providing synthesis files allows the user to synthetise the component for a given platform. It should not synthetise a full working design, only the bare minimum so the user can check its toolchain (esp. version), and investigate what hardware resources are infered.