ECE 545 Project Recommendations

Coding

1. Simple combinational components, such as 2-to-1 MUX, 4-to-1 MUX, etc.

Do not define separate entities for simple combinational components, such as 2-to-1 MUX, 4-to-1 MUX, decoder, shift, rotation, etc.

Use concurrent statements instead.

In particular, use

A. **when-else** for a 2-to-1 MUX
B. **with-select-when** for a 4-to-1 MUX.

2. Simple sequential components, such as registers, counters, etc.

When you define an entity for a simple sequential component, such as a register or a counter, make the width of an input and output bus generic, and set its default value to the width most often used in your circuit.

This way, you will need to use GENERIC MAP during instantiation only when the width of the component is different than the default value.

All sequential components should have an enable input, called `en`, active high.

Use
```
      rising_edge(clk)
```
rather than
```
      clk'event and clk=1
```

Do not define sequential components active on a falling edge of the clock.

3. Concurrent statements vs. sequential statements

Concurrent statements, such as

a. when-else
b. with-select-when

can be used only outside of processes (*at least in VHDL-93 which is a primary variant of VHDL, which we use in this project*).

Sequential statements, such as

a. if-else
b. case-is-when

can be used only inside of processes.

4. For-generate

Use `for-generate` statements to describe circuits with a regular, repetitive structure.

Note that you can have one `for-generate` loop included inside of another `for-generate` loop.

In order to use `for-generate` loops efficiently you may need to:

- declare signals as arrays of std_logic_vectors.
- rename signals on the edges of your circuit (which names violate the regular structure). Please note that it is perfectly OK to refer to the same node/bus in the circuit, using two different names, e.g., `signal_1` and `signal_2`, as long as these signals are connected using the concurrent statement:
  ```
        signal_2 <= signal_1;
  ```

RTL for synthesis

1. If statements in processes

Avoid testing for multiple conditions in your IF statements, especially in the case of the rising edge of the clock and control signals with a clear priority ranking, such as reset, enable, and load.

For example, the following IF statement leads to problems with synthesis:

```
if  rising_edge(clk) and en='1' then
…
end if
```

Use instead

```
if  rising_edge(clk) then
     if en='1' then
     …
     end if
end if
```

Test values in the following order:

```
1. rising_edge(clk)
2. en
3. ld (if present).
```

2. Range of indices

Ranges of indices in signals/ports/constants of the type `standard_logic_vector`, `unsigned`, and `signed` can be determined only using numbers, constants and generics.

You cannot use signals or ports (even after conversion to the integer type).

For example, the following code is not synthesizable:

```
signal lsb: std_logic_vector(6 downto 0);
…..
MSB <= input(127 downto to_integer(unsigned(lsb)));
```

This is because the value of `lsb` is unknown at the time of compilation, and this operation does not have any physical representation.

Similarly you cannot use either signals or ports to determine the range of an index in the `for-generate` loop.

3. Two assignments within a process

The code, such that

```
if  rising_edge(clk) then
     q1 <= a and b;
     q2 <= q1;
end if;
```

infers two registers:

- register with the output `q1` (and the input connected to the output of the gate `a and b`), and
- register with the output `q2` (and the input connected to the output of the register `q1`).

Thus, it takes two rising edges of the clock for the value of the expression `a and b` to propagate to `q2`.

Conventions

1. Active values of control signals

Assume that all internal control signals are active high.

2. Clock

Assume that your circuit has only one clock, with the same name, `clk`, at all levels of hierarchy.

Assume that all sequential components react only to the rising edges of this clock.

3. Local controllers

Any significant unit of your datapath (e.g., Hash_Datapath) can be equipped with its own local controller (e.g., Hash_Control).

When such a controller is added, the top level unit (e.g., Hash), containing this controller and the corresponding datapath should be created, and its interface and symbol clearly defined.

This approach will simplify the design of the entire control system, and reduce the number of control signals handled by the top-level controller.

4. No reset in the datapath

Please assume that no reset is used outside of control units.

Thus, no reset (either asynchronous or synchronous) should be used in any of your registers, counters, and higher-level datapath components other than the components including already a local controller.

Assume that an initial state of all sequential components is unknown until this state is changed during the operation of the circuit.

On the other hand, please use the external reset, called `rst`, active high, as an input to your controller (and any lower level controllers).

5. Meaningful names

Try to assign meaningful names to all entities, ports, signals, constants, and generics in your code.

Try to make them as close as possible to the specification of an implemented algorithm, but be aware of limitations on the use of special characters (only "_" if you follow the recommended VHDL-87 naming conventions).

Synthesis and Implementation

1. Target FPGA

Specify the following FPGA family and device as your first targets for both synthesis and implementation:

Family: Artix-7

Device: xc7a12t

Package: csg325

Speed: -3

Full name: xc7a12tcsg325-3

2. Case sensitivity

Please note that your synthesis tool, unlike your simulator, may be case-sensitive.

Therefore, please make sure that the spelling of all your names in VHDL source codes is consistent (up to the use of lower-case and upper-case characters) before you start synthesis.