ECE 545 Project Recommendations

Part 1

November 16, 2014

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

```vhdl
rising_edge(clk)
```

rather than

```vhdl
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 (or 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:

\[
\text{signal}_2 \leftarrow \text{signal}_1;
\]

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:

\[
\text{if rising_edge(clk) and en='1' then}
\]

\[
\text{...}
\]

\[
\text{end if}
\]

Use instead

\[
\text{if rising_edge(clk) then}
\]

\[
\text{if en='1' then}
\]

\[
\text{...}
\]

\[
\text{end if}
\]

\[
\text{end if}
\]
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:

```vhdl
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

```vhdl
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, \texttt{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., \texttt{AES\_EncDec\_Datapath}) can be equipped with its own local controller (e.g., \texttt{AES\_EncDec\_Control}).

When such a controller is added, the top level unit (e.g., \texttt{AES\_EncDec}), 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 (the same as in the sample codes of AES and Keccak\_F provided by Ice).

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 \texttt{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 a given CAESAR candidate, 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. Tools
You can follow either

- Tutorial on FPGA Design Flow based on Xilinx ISE and ISim, or
- Tutorial on FPGA Design Flow based on Aldec Active-HDL

available at https://ece.gmu.edu/tutorials-and-lab-manuals

In either case, choose Xilinx XST as your synthesis tool.

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

**Family:** Virtex 6
**Device:** XC6VLX75T
**Package:** FF784
**Speed:** -3

3. Wrapper
For your Datapath (or a lower-level circuit) to fully pass the synthesis and implementation, you may need to reduce its pin requirements.

In order to do that, please write a `wrapper.vhd`, in which you instantiate your circuit, and then add a SIPO in front of any input bus, and a PISO at the end of each output bus. By setting the external bus width of SIPO and PISO to reasonably small values, such as 64, 32, 16, or even 8, you can considerably limit the pin requirements, without excessively effecting the maximum clock frequency and resource utilization of your circuit.

4. 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.
**Submissions**

1. **VHDL files only rather than the entire project folder**

Submit on Blackboard only VHDL files, rather than the entire contents of your project directory, containing a lot of intermediate, log, and report files.

Do not submit any old VHDL files, which you are not planning to use any longer (with the exception of testbenches that you have used to verify selected lower-level components).

Place all your testbenches in the subfolder `tb`.

2. **Source_list.txt**

Create a file `source_list.txt`, containing the list of all your active (still used) synthesizable source files arranged in the bottom-up order, i.e., starting from the packages, through low- and medium-level components, up to the top-level entity.

Each file name should be placed in a separate line, e.g.:

```
AES_pkg.vhd
AES_Sbox.vhd
AES_SubBytes.vhd
AES_ShiftRows.vhd
AES_mul.vhd
AES_MixColumn.vhd
AES_MixColumnns.vhd
AES_Round.vhd
AES_KeyUpdate.vhd
AES_map.vhd
AES_invmap.vhd
AES_Enc_Datapath.vhd
AES_Enc_Control.vhd
AES_Enc.vhd
```

3. **Reports from synthesis and implementation**

Locate readable reports from the synthesis and implementation and place them in a separate folder called `reports`.

4. **Number of clock cycles**

Prepare and keep updating a text, MS Word, or PDF file (timing.xxx) with formulas for the number of clock cycles required for

A. Key scheduling (generating round keys)
B. Authenticated encryption
C. Authenticated decryption
as a function of the number of blocks of plaintext/ciphertext, n, and the number of blocks of associated data, m.

5. Results

Prepare a text, MS Word, or PDF file (results.xxx) containing your first results concerning the datapath (or its largest chunk you have implemented), including:

A. Timing results

After synthesis:
- Minimum clock period
- Maximum clock frequency

After implementation:
- Minimum clock period
- Maximum clock frequency
- Throughput for processing of Associated Data only (assuming long data)
- Throughput for encryption of Message only (assuming long messages)
- Throughput for decryption of Ciphertext only (assuming long ciphertexts)

B. Resource utilization (after implementation only)

- Number of slices
- Number of LUTs
- Number of Flip-flops
- Number of BRAMs
- Number of DSP48 units
- Number of I/O pins.

6. Verification

Place all reference C source codes used to generate test vectors, with your modifications clearly marked using comments, in the folder called C. Feel free to create and include any scripts or makefiles that can help you to generate and format test vectors.

Prepare a text, MS Word, or PDF file (verification.xxx) describing
A. highest level entity verified for functional correctness
B. results of verification of lower-level entities, including
   - Name of the entity
   - Name of the testbench used for verification
   - Test vectors (actual inputs and the corresponding outputs in the hexadecimal notation)
   - Result of verification: correct or incorrect behavior
   - Possible sources of errors.
7. List of questions and problems

Please feel free to include a text, MS Word, or PDF file (problems.xxx) describing any questions you currently have and any new problems that you have encountered since our last meeting.

Extend this file with answers (or at least indications that a given problem was solved) during subsequent submissions.

8. Zip!

Zip all the aforementioned files together using the name with the following format:

<cipher_name>_<your_first_name>.zip

Submit all your codes and associated files related to the datapath on Blackboard under

DATAPATH CODES & TIMING ANALYSIS.

9. Updated block diagrams

Whenever you make any substantial change in your block diagram, please submit it on Blackboard under

CLOSE-TO-FINAL-BLOCK-DIAGRAM.

10. No deadlines or limits on the number of attempts

Please note the Blackboard assignments

DATAPATH CODES & TIMING ANALYSIS

and

CLOSE-TO-FINAL-BLOCK-DIAGRAM

have no deadlines or limits on the number of attempts.

Please submit as often as possible to make sure that Ice, Umar, and I have access to the most recent version of your work, and can locate any additional mistakes or inefficiencies in your designs.