**Integrated Systems Architectures**

# Lab 2: digital arithmetic
# Assignement

Read the Lab 2 description and address the following points.

## 1 Digital arithmetic and logic synthesys

### 1.1 Synthesis strategies

- Download from the *Portale della didattica* the "Floating Point Unit lite" (cvfpu_lite) project. This project is a simplified version of the *cvfpu* project developed by the OpenHW group (`https://github.com/openhwgroup/cvfpu`). It is Floating Point Unit developed in SystemVerilog and is compliant with the IEEE 754-2008 standard:
  `https://en.wikipedia.org/wiki/IEEE_754`.
  Moreover, it supports different floating point formats, including the half-precision (16 bit) one:
  `https://en.wikipedia.org/wiki/Half-precision_floating-point_format`.
- Unpack the project. You have sources (`src`) and testbench (`tb`) files. As wou can see you have two topfiles for the testbench: *tb_fpnew_top_rtl.sv* and *tb_fpnew_top_net.sv* as user defined types at the top module interface in the RTL description are translated to standard arrays in the netlist. The text file prj.txt gives you the hierachy order of the files (source and testbench), so it can be useful to derive scripts both for simulation and synthesis.
- Verify the model by testing it with a simple testbench to perform at least 10 different half-precision floating point multiplications$^{(1),(2)}$.
- Build a table summarizing the results asked in the following points:
  1. Synthesize with `compile`. Find the maximum frequency and the area. Verify the netlist behaviour via simulation.

2. Repeat the previous step issueing the `optimize_registers` command after `compile`. Find the maximum frequency and the area. Verify the netlist behaviour via simulation.
3. Repeat the previous step issueing only the `compile_ultra` command. Find the maximum frequency and the area. Verify the netlist behaviour via simulation.
4. Force Design Compiler to flatten the hierarchy and to implement the Significands multiplier (Mantissa multiplier in fpnew_fma.sv [(3)]) as a CSA multiplier. Find the maximum frequency and the area with the commands `compile` and `optimize_registers`. Verify the netlist behaviour via simulation.
5. Repeat the previous step by forcing Design Compiler to implement the Significands multiplier as a PPARCH mulitplier. Find the maximum frequency and the area with the commands `compile` and `optimize_registers`. Verify the netlist behaviour via simulation.

- Add as an appendix in your report the full text of the `report_timing` and `report_area` commands. For the CSA and PPARCH architecture add the full text of the `report_resources` command as well.

**Note:** the maximum frequency corresponds to the minimum period with slack met and equal to zero.

(1) you can extend the testbench already provided in cvfpu_lite project. You also have in the project a simple C++ program (fp16_num.cpp), which performs the product between two half-precision floating point numers and prints the multiplicand, the multiplier and the result showing also the binary representation in compliance with the IEEE 754-2008 standard. Note that the program relies on the *flexfloat* library (`https://github.com/oprecomp/flexfloat`) which requires *CMake 3.1 or higher* and *GCC 7.1 or higher*. Thus, it does not build on the server. Nevertheless, you have the sources and a static-compiled executable (fp16_num) which you can run on the server (e.g. `./fp16_num 15.0 204.0`).

(2) you will see some warnings during the compilation concerning potential conflicts with always_comb and always_latch variables. Extra checking is done later when invoking vsim, so you can neglect this kind of warning.

(3) in the hierarchy of the design you find the mantissa multiplier in: *gen_operation_group[0]*

→ *i_opgroup_block* → *gen_parallel_slices[2]* → *active_format* → *i_fmt_slice* → *gen_num_lanes[0]* → *active_lane* → *lane_instance* → *i_fma*.

### 1.2 R4-MBE multiplier implementation

Design in SystemVerilog a Radix-4 Modified Booth Encoder (R4-MBE) based multiplier for unsigned data and use it as the Mantissa multiplier in fpnew_fma.sv.

Partial products must be generated with no adders/substracters [1]. The adder plane must rely on a Dadda-tree [2]. Sign extension bits in the Dadda-tree must be simplified in order to reduce the number of half adders and full adders [3].

Once the design is finished you have to:

1. Simulate the multiplier first as a stand-alone component and then included in the FPU as the Mantissa multiplier;

2. Synthesize the FPU including your muliplier with `compile_ultra`. Find the maximum frequency and the area. Verify the netlist behaviour via simulation.

3. Add as an appendix in your report the full text of the `report_timing` and `report_area` commands.

## References

[1] M. Martina. The modified booth encoder multiplier.

[2] M. Martina. About wallace and dadda trees.

[3] G.W. Bewick. *Fast multiplication: algorithms and implementation - Appendix A - Sign Extension in Booth Multiplier*. PhD thesis, Stanford, 1994.