



Politecnico di Torino
III Facoltà di Ingegneria

Lab 2 Report

Integrated Systems Architecture

Master degree in Computer Engineering

Authors: ISA36

Nicole Dai Prà s274501, Leonardo Izzi s278564

December 6, 2020

Many thanks to Prof. Mariagrazia Graziano for providing us with this template.

Contents

1	Introduction	1
2	FP Multiplier	2
2.1	Model Verification	2
2.2	Synthesis	2
2.3	Fine-grain Pipelining and Optimization	2
3	MBE Multiplier	4
3.1	MBE Implementation	4
3.1.1	Dadda Tree	4
3.2	Synthesis	5

CHAPTER 1

Introduction

The lab 2 assignment consisted in various synthesis experiments on a floating point multiplier and in the development of a unsigned integer multiplier, based on the Booth's algorithm and Dadda's tree, to be used within the floating point multiplier.

As required, there is a GitHub repository available at the following link: https://github.com/leoizzi/isa_labs/tree/main/lab2.

The folder is organized as follows:

- `fpvhd1`, the folder containing all the VHDL files of both the floating point and integer unsigned multipliers.
- `lab2_report.pdf`, this file.
- `report`, the folder containing the Latex files of the report.
- `sim`, the folder where all the simulation scripts are stored.
- `syn`, the folder where all the synthesis script, as well as the reports, are saved.
- `tb`, the folder containing the testbench files.
- `dadda.py`, a python script that generates the VHDL instantiation of the Dadda tree.

CHAPTER 2

FP Multiplier

2.1 Model Verification

Before starting any work on the multiplier, we verified that it was working as intended. Hence, we have created a file called `tb_fpumult.vhd`, where the DUT computes the square of the numbers stored in `fp_samples.hex` and the result is compared against the values stored in `fp_prod.hex`. Then, we added the required input registers and we have verified again that the results were correct.

2.2 Synthesis

We have performed various synthesis to analyze the differences between various implementations and constraints. To do this, we have written three different synthesis scripts, that are `syn_script.tcl`, `syn_script_csa.tcl` and `syn_script_pparch.tcl`. The first one synthesizes the multiplier by leaving all the implementation choices to the Synopsys' tool, while the second forces the usage of CSA-based multipliers and the last one the usage of parallel-prefix based multipliers. The asked results are shown in table 2.2.

Multiplier architecture	T_{ck}	Area
Chosen by the synthesizer	1.6 ns	3999.04 μm^2
CSA	4.6 ns	4807.68 μm^2
PPArch	4.5 ns	3734.91 μm^2

Performance is remarkable when the synthesizer is allowed to choose the implementations by itself. By looking at the resources' report the synthesizer chooses the parallel prefix multiplier, although it is optimized differently in respect to the one used in the synthesis done with `syn_script_pparch.tcl`. In fact, the former is optimized for both area and speed, while the latter only for area. However, for an increase of about only 7% in area we obtain a performance boost of 65%, hence in a real application there is no doubt in the implementation we would choose.

2.3 Fine-grain Pipelining and Optimization

We added, as asked, the register after the significands' multiplier in the second stage, as well as all the required registers to maintain the correct timing. We verified the correctness of the updated design with the `tb_fpumult_reg.vhd` testbench. Then we have run two synthesis, one with the `compile_ultra` command (`syn_script_comp_ultra.tcl`) and one with a simple `compile` and the `optimize_registers` (`syn_script_opt_reg.tcl`). The results are summarized in table 2.3.

Synthesis commands	T_{ck}	Area
<code>compile + optimize_register</code>	0.8 ns	4969.41 μm^2
<code>compile_ultra</code>	1.5 ns	4216.1 μm^2

The synthesis with retiming reaches double the frequency of both the one done with the `compile` in the previous section and the one done with the `compile_ultra`. This shows the value of the optimization techniques we have studied. However, it suffers of an area increase with respect to the `compile_ultra` synthesis, since it probably adds more registers due to the non-negative register count on the graph's arcs. Nevertheless, its speed is outstanding.

CHAPTER 3

MBE Multiplier

3.1 MBE Implementation

We have implemented an unsigned integer multiplier using the radix-4 Modified Booth's Encoding. We generate the partial products as shown in `modified-booth.pdf` without using any adder/subtractor. To reduce the height of the partial products' tree (and hence the number of HAs and FAs used) we have organized the bits as explained in `sign_extension_booth_multiplier_Stanford.pdf`. Unfortunately, to us this structure seems to be not very regular from the perspective of the Dadda tree implementation, in fact we have found difficulties in writing the algorithm by exploiting the `for generate` statement. Writing the whole tree by hand was not an option neither due to the tree size, therefore we have developed a Python script that generates the tree in VHDL by using only wire connections and FAs and HAs instantiations.

3.1.1 Dadda Tree

The Python script is fairly easy: first, with the function `gen_start_matrix` the initial partial products matrix is generated. With a 1 we indicate that in position (i, j) a dot is present, with a 0 the opposite. Then, with the function `count_levels` we calculate how many levels the Dadda tree has, given the number of partial products, and how many dots are allowed in a column for each level.

After, we generate an array for each level where we keep the count of how many dots have been placed for each column. This is used to find out when a HA or a FA must be instantiated. This is accompanied by a matrix per level that has been used for debugging purposes and, in the last tree level, to find out where a ground connection must be placed.

The main algorithm works as follow:

- For each tree level l it passes over all the column of `cnt_matrix[l]`
- It computes the difference between how many dots are present in the j -column (that is, the dots present in the current column of the current level), plus the the eventual carries coming from the FAs and HAs of the column $j - 1$) and minus the maximum number of dots allowed in the next level.
- If $diff \leq 0$ all the dots can be simply propagated to the next level, otherwise HAs and/or FAs must be allocated.
- The allocation process first tries to add as many FAs as possible, then it tries to allocate as many HAs as needed. All the remaining dots (if any) are then propagated to the next level.

The script outputs on a file the VHDL instantiation, which has been directly copied in `dadda.vhd`, since we used an entity to encapsulate the auto-generated code.

3.2 Synthesis

We performed synthesis with `compile`, `compile_ultra` and `compile + optimize_registers`. The results are summarized in table 3.2.

Synthesis commands	T_{ck}	Area
<code>compile</code>	2.6 <i>ns</i>	5497.42 μm^2
<code>compile_ultra</code>	0.7 <i>ns</i>	μm^2
<code>compile + optimize_register</code>	0.8 <i>ns</i>	4969.41 μm^2