Integrated System Architecture

Lab 2 Report

Academic Year 2020-2021

Professors:	Masera Guido	
	Martina Maurizio	
	Valpreda Emanuele	
	Walid Walid	
Students:	Lagostina Lorenzo	288019
	Lanza Mauro	290766
	Tudisco Antonio	285433
Group Number:	5	

Contents

1 Simulation

To verify the *FPUVHDL* project given to us we created a Testbench that uses 3 components:

- CLK_GEN: a component that creates the clock signal with a period of *10ns*. The clock is stopped when and *END_SIM* signal is received
- FPmul: the floating-point multiplier under test
- Data_Maker: it extracts from a *fp_samples.hex* file the data that is fed to the TB, when all the data is analyzed, it sends an *END_SIM* signal that stops the simulation

After executing the multiplication (we executed the square of each signal) we had the following results (results from *sim_res.txt* file)

Sample	**Result**
00000000	00000000
3e9e377a	3DC3910D
25a00002	0BC80005
3f4f1bbd	3F278DDF
25400003	0B100005
3f800000	3F800000
a5e00002	0C440004
3f4f1bbd	3F278DDF
a6900000	0DA20000
3e9e377a	3DC3910D

We then proceeded to add registers in the FPU and we verified that the results were the same (these can be found in the *sim_res_reg.txt* file)
We were able to see that in both cases the results are the same as in the *fp_prod.hex* file given to us to check the correct behavior of our project.

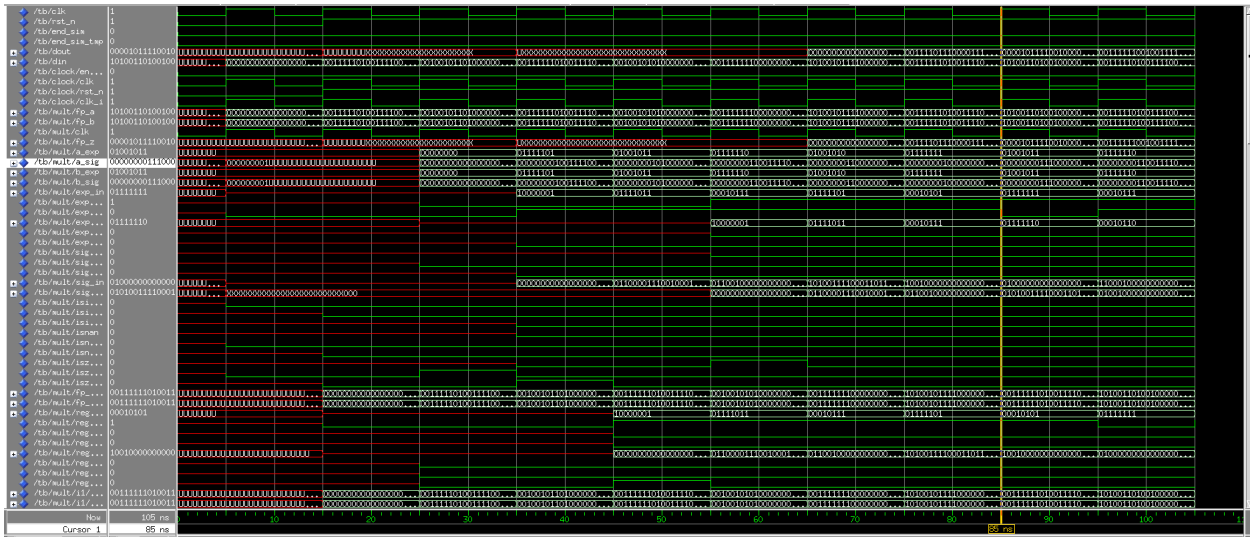

Figure 1.1: Waveform of the simulation

2 MBE

2.1 Python Scripts

To design the MBE multiplier, we've decided that the most efficient way was using a series of Python scripts to calculate:

- Number of stages required
- Maximum number of signals per column for every stage

In order to performe those actions we used the following function:

$$l_j = \frac{3}{2} l_{j-1} \tag{2.1}$$

This recursive function must be repeated until the maximum number of elements per column accepted is higher or equal to the maximum number of signals generated, in our case, until

$$\frac{n_b}{2} + 1 \leq l_j \tag{2.2}$$

In which n_b is the number of bits of the input.
Then, we designed the matrix that contains all the signals following the instruction of the Booth modified and also of the signal extension, guaranteeing also the reduction of the height.
After designing the initial matrix, which is seen by column, the Dadda algorithm is applied, to add the minimum number of full-adder and half-adder.
The function for every column and every stage saves the number of half adders and

full adders needed and also the number of resulting elements on the column.
Dadda's algorithm written, analyzes the number of signals present in the current column. If the number of elements is greater than the maximum number accepted, the algorithm: if the difference is exactly equal to 1 will add a half adder, and so reduces 2 elements to 1 in the same column and 1 in the following column, else will add a full adder, and so reduces from 3 elements to 1 in the same column and 1 in the next column. This mechanism is repeated until the number of elements in the column analyzed does not correspond to the maximum value accepted.
In the end, the resulting values are reported in 3 matrices in a package file written in VHDL.

```
library ieee;
use iee.std_logic_1164.all;
use ieee.numeric_std.all;
package bf is
        CONSTANT NBIT : INTEGER := 8;
        CONSTANT ROW_NUMBER : INTEGER := 5;
        CONSTANT STAGE_NUMBER : INTEGER := 4;
        CONSTANT COLUMN_NUMBER : INTEGER := 16;
        type stage is array (0 TO 15) of INTEGER;
        type mat_add is array (0 to 2) of stage;
        type mat_el is array (0 to 3) of stage;
        CONSTANT fa_mat : mat_add := (
(0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0),
(0,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0),
(0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1));
        CONSTANT  ha_mat : mat_add := (
(0,0,0,0,0,0,1,1,0,0,0,0,1,0,0,0),
(0,0,0,0,1,1,0,0,0,0,0,0,0,0,1,0),
(0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0));
        CONSTANT  element_mat  : mat_el := (
(2,1,3,2,4,3,5,4,5,5,5,5,4,3,3,2),
(2,1,3,2,4,3,4,4,4,4,4,4,4,3,2),
(2,1,3,2,3,3,3,3,3,3,3,3,3,3,3),
(2,1,2,2,2,2,2,2,2,2,2,2,2,2,2));
end bf;
```

Listing 2.1: Pack

2.2 VHDL description

To describe the MBE, we exploited the for generate function of VHDL, bound to iterate for a number of times equal to the total stage number plus 2, together with a redundant cubical signal structure.
In the first iteration, we instantiate the Booth Encoders, and assign the values coming from them to the first layer of our structure, effectively replicating the disposition in Figure 2.1.
The second interaction "flips" the most significant half of the matrix upside-down, distributing the signals in the pyramid-like shown in Figure 2.2. This allows the use of the same VHDL code to place the FAs and HAs at every step, which is instantiated in all the following stages.
The last step is used to create the RCA, which performs the final addition, delivering the result.

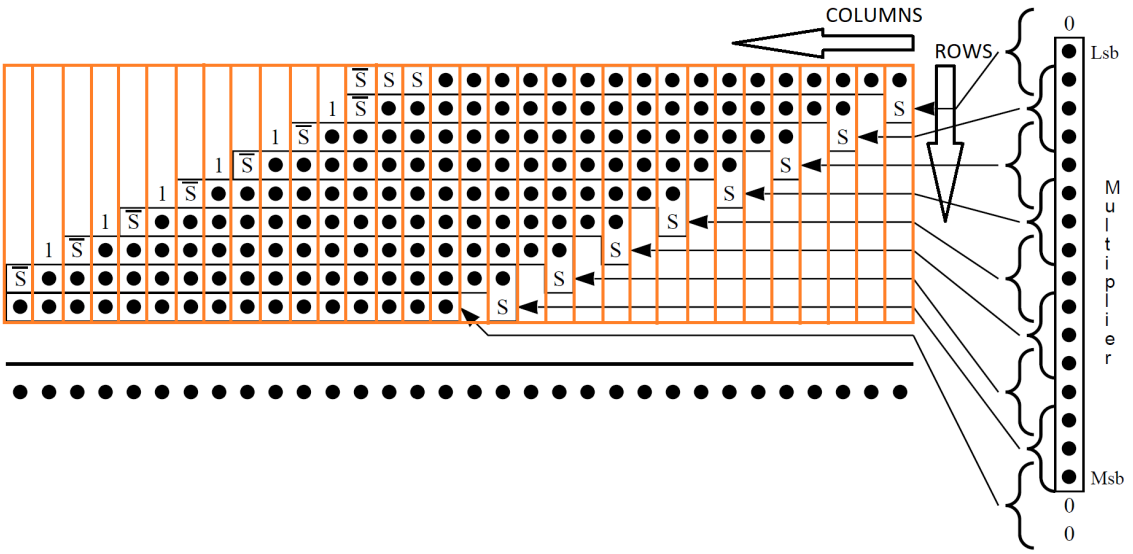

Figure 2.1: Disposition of the signals in the first layer

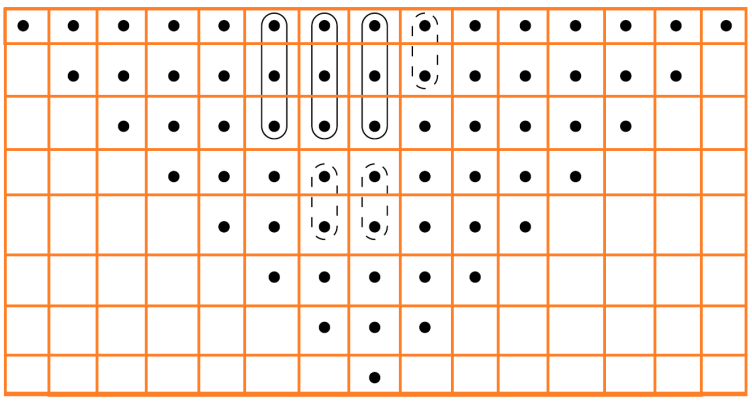

Figure 2.2: Disposition of the signals in the second layer

At each stage, the signals from the previous layer are used as inputs, and the results are transmitted to the following one In the generic stage, the HAs and FAs are instantiated. We achieved this by exploiting the integer matrices stored in the package by the Python script. Each row of a matrix describes the number of either full adders or half adders to be instantiated on every column of the stage. By iterating on both these rows and the number of HAs and FAs, we managed to create the adders and properly connected them inside of the signal cube. Any spare signal is connected to the first free signal of the following layer. A visualization of the signal cube and FA instantiation is presented in Figure 2.3.

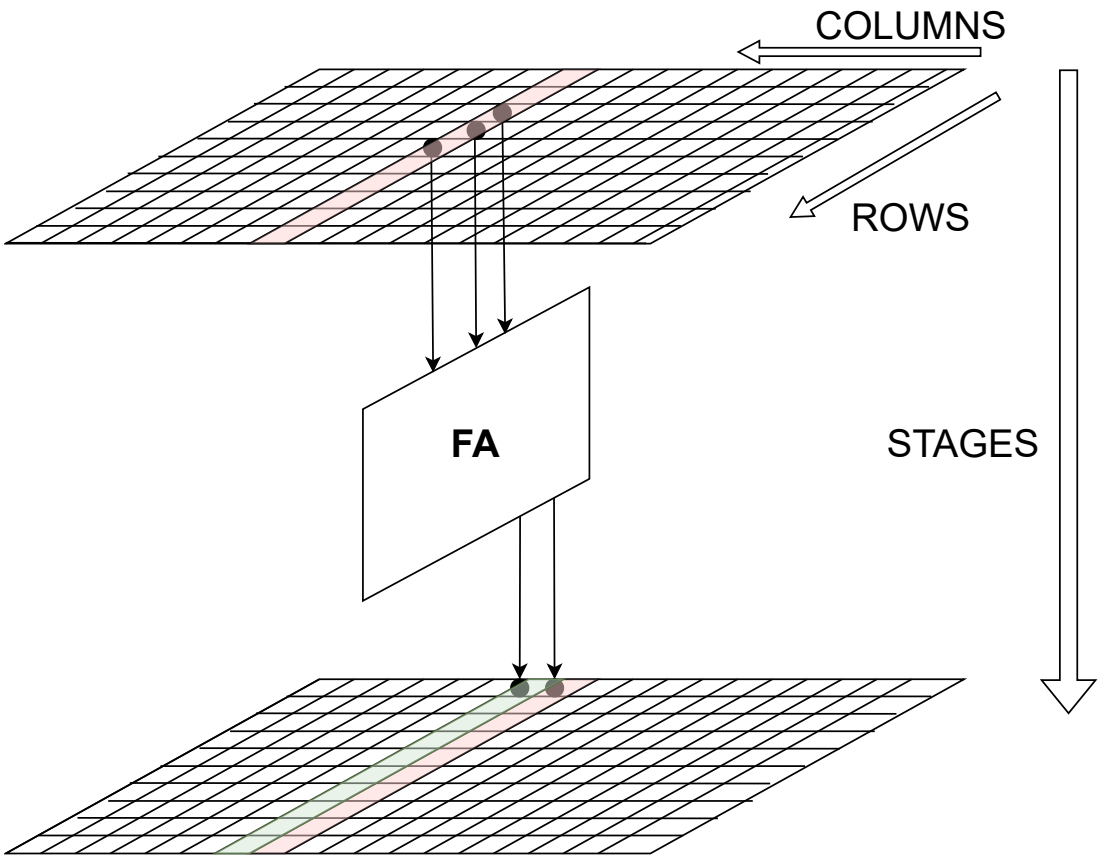

Figure 2.3: Visualization of the signal structure and FA instantiation

The resulting signal distributing is shown in a testbench image in Figure 2.4.

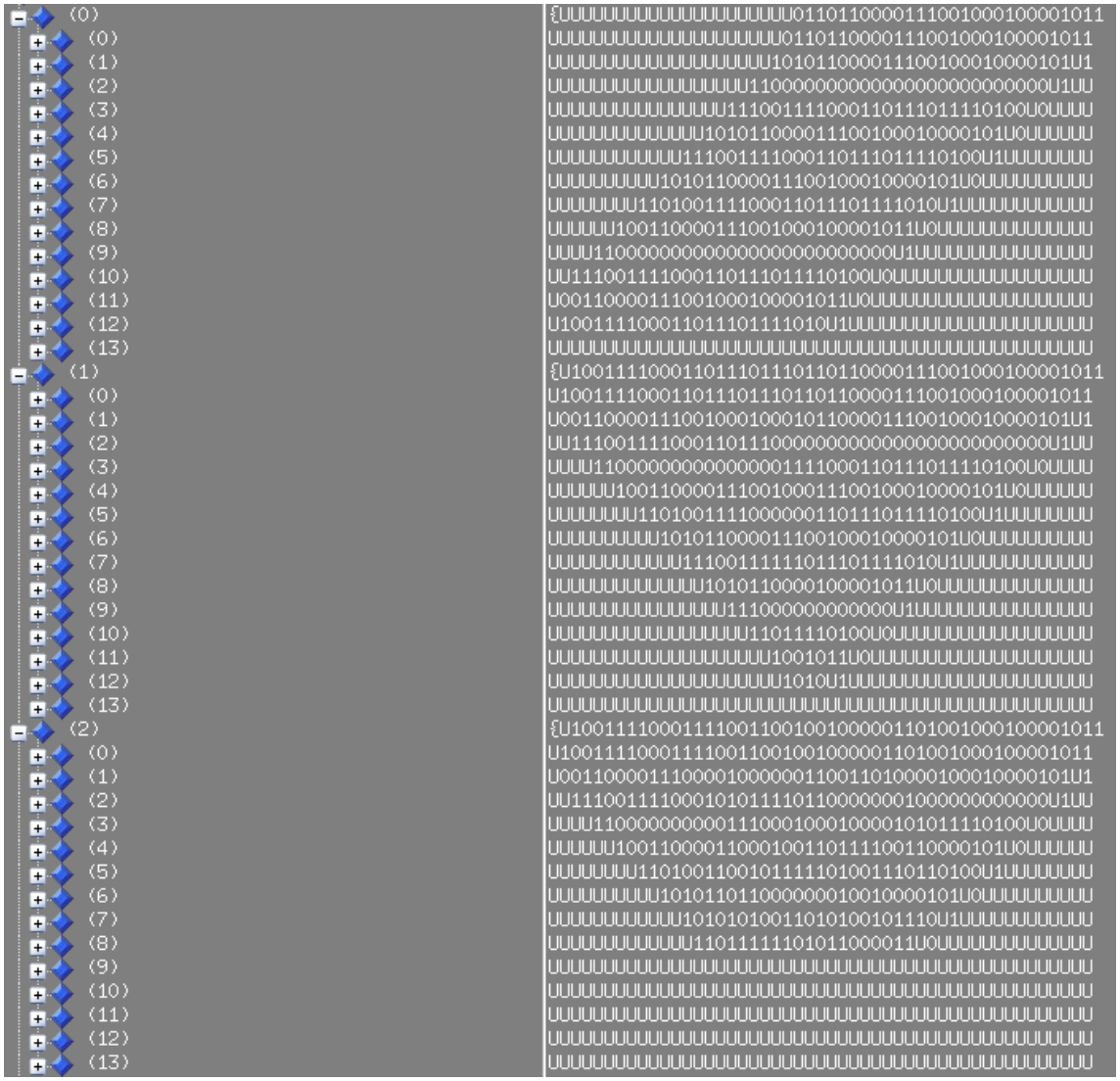

Figure 2.4: Testbench image of the signal distribution

3 Synthesis Analysis

Legend

- CP: Critical Path
- No imp: synthesized without specifying any particular implementation
- CSA: synthesized with Carry Save Adder architecture
- PPARCH :synthesized with Parallel Prefix Architecture
- MBE_32: Modified Booth Encoder Multiplier developed to accept 32 bit significands (33 bit wide with the hidden '1')
- MBE_23: Modified Booth Encoder Multiplier developed to accept 23 bit significands (24 bit wide with the hidden '1')
- No opt: synthesis executed applying only the `ungroup -all -flatten` command
- Opt Reg: synthesis executed adding the `optimize_register` command
- Ultra: synthesis executed adding the `compile_ultra` command

3.1 Critical Path

Table 3.1: Critical Path Results

CP (ns)	No imp	CSA	PPARCH	MBE_32	MBE_23
No opt	1,57	4,24	1,59	0,8	0,78
Opt Reg	0,8	1,25	0,8	0,81	0,82
Ultra	0,8	1,25	0,8	1,57	1,52

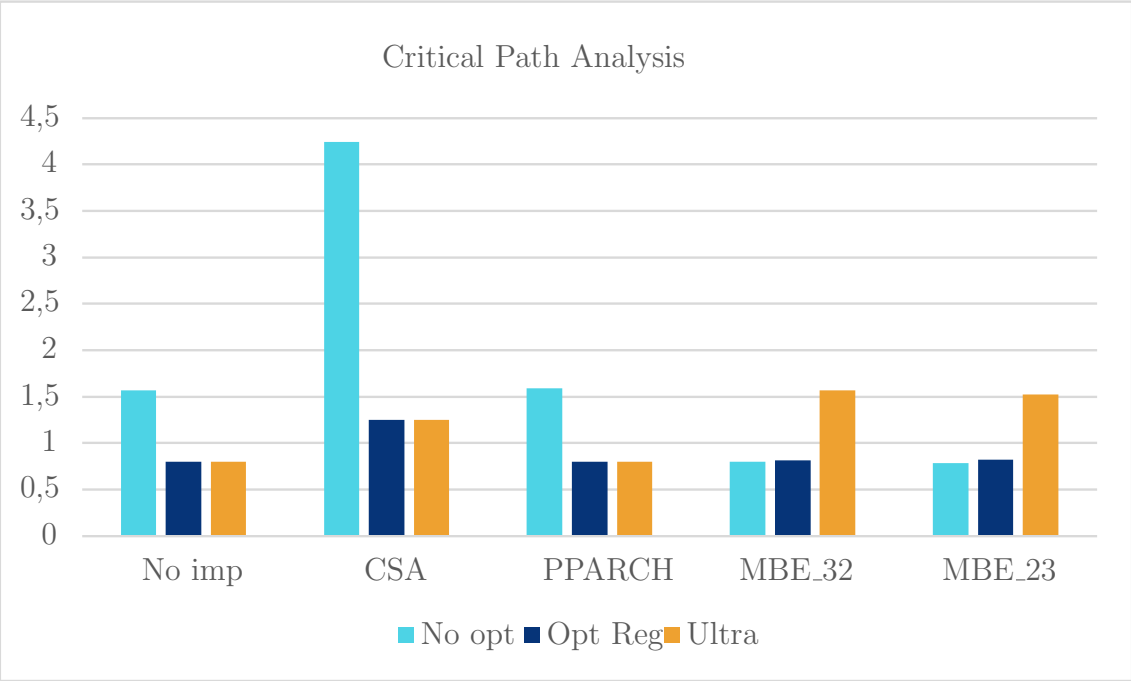

Figure 3.1: Bar diagram of the critical paths in ns

What emerges from the data is that the **CSA** implementation is the worst perfoming one in terms of critical path. The **No Imp** and **PPARCH** display similar behaviours, and manage to achieve the best performance (0.8 ns) in both the **Opt Reg** and **Ultra** analysis. Instead, the **MBE_32** and **MBE_23** perform best in the first two analysis, while achieving similar in the **ultra** one results similar to those of **No Imp** and **PPARCH** in the first case.

3.2 Area

Table 3.2: Area Results

Area (μm^2)	No Imp	CSA	PPARCH	MBE_32	MBE_23
No opt	4037,6	4853,7	4011	1248,6	1249,7
Opt Reg	4915,1	4176,2	4915,1	7864,3	7086
Ultra	4915,1	4176,2	4915,1	5533,9	4999,2

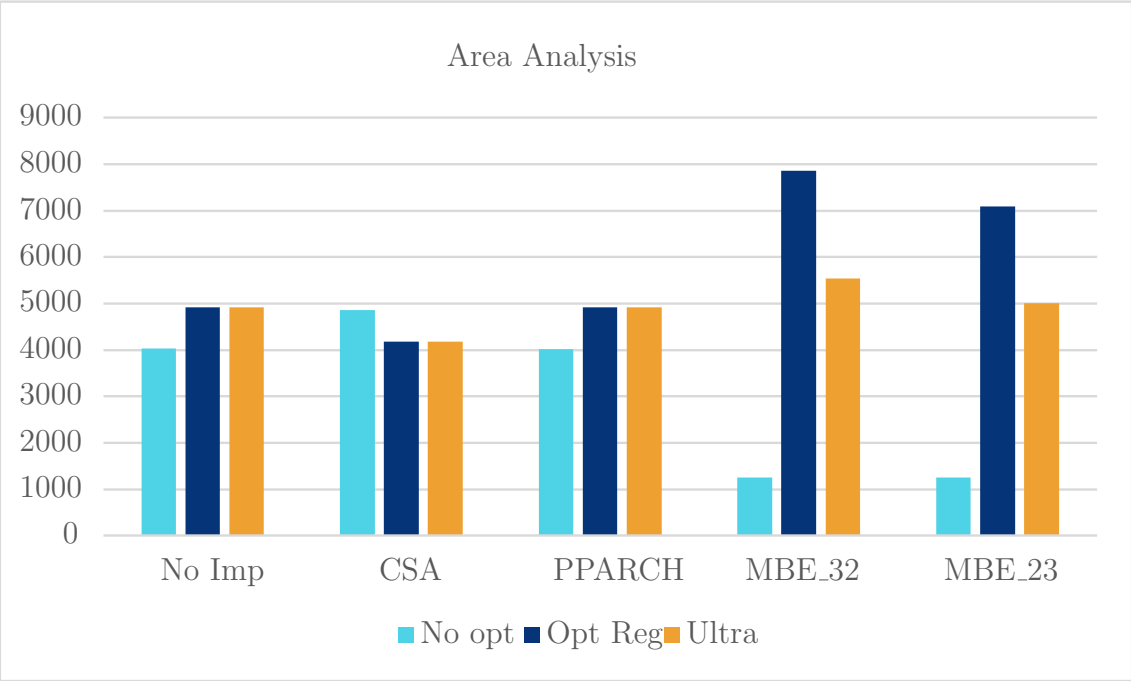

Figure 3.2: Bar diagram of the area in μm^2

In terms of area, the **CSA** performs worse only in the first analysis and delivers the smaller area for the **Opt Reg** and **Ultra** cases. The **No Imp** and **PPARCH** have average values of the area, while the **MBE_32** and **MBE_23** achieve the best result in the **No Opt** synthesis while delivering the greatest areas in the other two cases. Differently from what seen the Critical Path analysis, here the **MBE_23** actually performs better than the **MBE_32**.

3.3 Conclusions

After comparing both timing and area performances, it is clear that the best area-delay product is delivered by the **MBE** architectures synthesized without exploiting further optimizations. This clearly shows how the choice of an optimal architecture is critical for the performance of a digital circuit since the absolute best result cannot be achieved by the synthesizer alone.
What the results also suggest is that Design Compiler used the **PPARCH** implementation when no particular implementation is specified. Another curious phenomenon is related to the two **MBE**'s behavior: the analysis of the critical path might suggest that the synthesizer avoided the instantiation of cells on those paths that would not have been driven in the 32-bit implementation. However, the analysis of the area shows a difference in the **Opt Reg** and **Ultra** cases, suggesting that either more hardware is allocated in the 32-bit case, or that a VHDL code that specifies the correct parallelism from the beginning allows the synthesizer the execute the best choices.
Ultimately, it can be observed how the `optimize_register` command allows significant improvement in the timing in the first three implementations, leading however to an area overhead for the **No Imp** and **PPARCH**. The `compile_ultra` command instead shows different results from the previous one only for the **MBE**s, where it appears to trade off the delay to reduce the area.

4 Appendix

The following table describes the content of each folder

Folder Name	**Description**
fpuvhdl	Normal multiplier given to us to download
fpuvhdl_2	Multiplier with the addition of the registers
fpuvhdl_2_MBE	Modified Booth Encoder with a 32 bit input with an extra register in the pipeline
fpuvhdl_2_MBE_23	Modified Booth Encoder with a 23 bit input with an extra register in the pipeline
fpuvhdl_MBE	Modified Booth Encoder with a 32 bit input
fpuvhdl_MBE_23	Modified Booth Encoder with a 23 bit input
MBE	Source files for the MBE
PythonScripts	Contains the scripts used to generate the matrix
syn	Synthesis of the multiplier without the extra register
syn_2	Synthesis of the multiplier with the extra register
syn_2_MBE	Synthesis of the 32 bit MBE with the extra register
syn_2_MBE_23	Synthesis of the 23 bit MBE with the extra register
syn_2_ultra	Synthesis of the MBE compiled ultra
syn_MBE	Synthesis of the 32 bit MBE without the extra register
tb	Testbench files for the normal multiplier
tb_2	Testbench files for the multiplier with the extra register
tb_2_MBE	Testbench files for the 32 bit MBE
tb_2_MBE_23	Testbench files for the 23 bit MBE