Politecnico di Torino

III Facoltà di Ingegneria

Exercises and Homeworks for the course Integrated Systems Architecture

LAB 2: digital arithmetic

Master degree in Electronic Engineering

Group number: 37

Costantino Taranto - 274492
Pasquale Santoro - 278329
Alberto Aimaro - 253196

December 16, 2020

Repository Link: `https://github.com/isa037/lab2`

Contents

CHAPTER 1

Digital arithmetics and logic synthesys

This analysis is focused on the pipelined Floating-Point multiplier shown in Fig.1.1 evaluated starting from the provided *vhd* files.

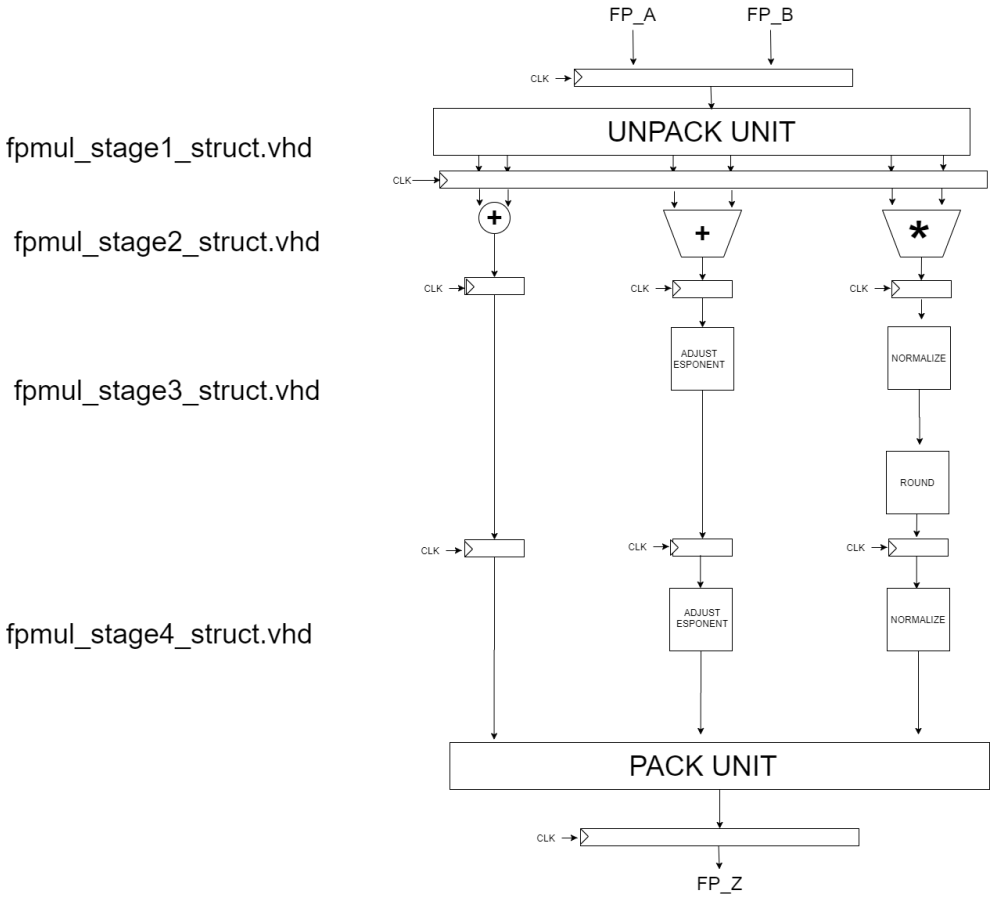

Figure 1.1: Pipelined FP multiplier

Notice that registers to sample the inputs have been added in an original file, called *fpmul_pipeline_inputRegistered.vhd*

1.1 Testbench for FP multiplication

The **contents** related to this section can be found in the **branch 'master'**

The verification of the FP multiplier is done through a testbench which generates the input data and checks the outputs automatically, writing the results of the simulation on a file (*results.csv*). The block diagram with the connections between all the entities is shown in fig 1.2.

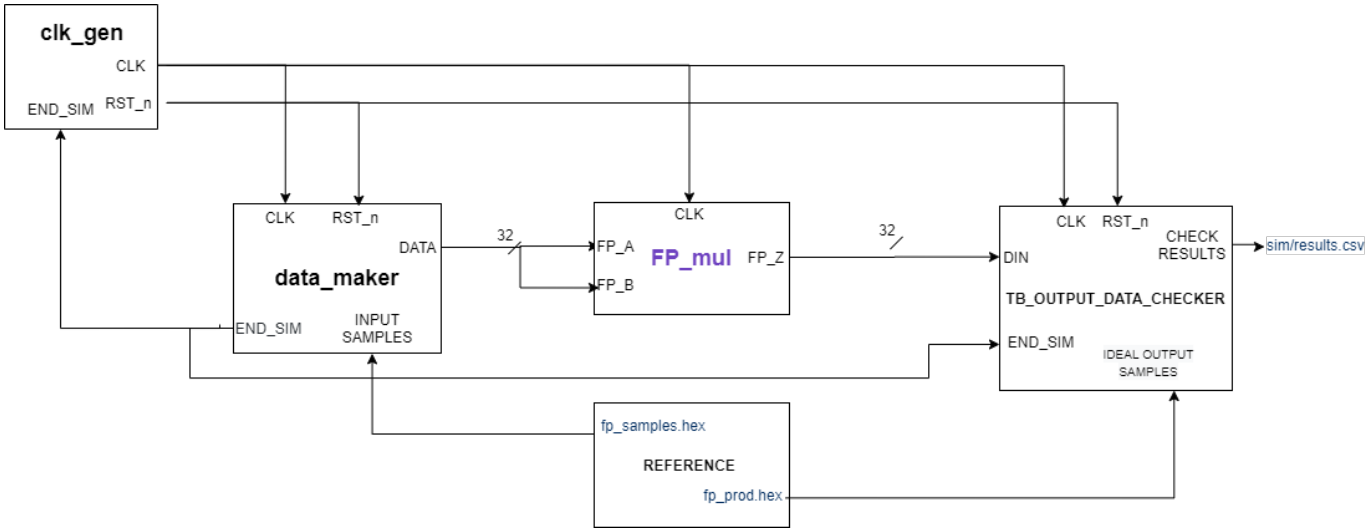

Figure 1.2: Testbench

The behaviour of the multiplier in both the implementations with and without registers on the inputs (*fpmul_pipeline_inputRegistered.vhd* and *fpmul_pipeline.vhd* respectively) is tested. What is checked is that, given the same value for both the inputs, the result is the square. The values of the input operands is taken by the file *fp_sample.hex*, the values of the outputs are written in the *results.csv*, while the value of the correct result is in the file *fp_prod.hex*.
The **correctness** of the results in both cases is reported by the testbench in the *results.csv* through a **successful** message.

1.2 Comparison of different multipliers

The **contents** related to this section can be found in the **branch 'point1/multiplier_with_input_registers'**

In the VHDL description, the Significands multiplication contained in Stage 2 is implemented as behavioral. Synopsys permits to handle the behavioural description of multipliers by directly inferring a component from a netlist of ready-to-use blocks. In particular, it contains a parametric multiplier *DW02_mult* that through the command *set_implementation*, can be forced to implement the Significand multiplication in two different ways:

- Multiplier based on CSA

  The command that must be used is *set_implementation DW02_mult/csa* [*find cell* $*mult*$]

- Multiplier based on Parallel Prefix adder

  The command that must be used is *set_implementation DW02_mult/pparch* [*find cell* $*mult*$]

In the following report are showed the Synopsis commands required to compile, for example, the design of the Parallel Prefix architecture.

```
#!/bin/bash
cd /home/isa37/git/lab2/syn/
rm -r work
mkdir work
mkdir analysis_results
source /software/scripts/init_synopsys_64.18

#run synopsis
dc_shell-xg-t

#*******Reading VHDL source files**************
analyze -f vhdl -lib WORK ../src/common/fpnormalize_fpnormalize.vhd
analyze -f vhdl -lib WORK ../src/common/fpround_fpround.vhd
analyze -f vhdl -lib WORK ../src/common/packfp_packfp.vhd
analyze -f vhdl -lib WORK ../src/common/unpackfp_unpackfp.vhd
analyze -f vhdl -lib WORK ../src/multiplier/fpmul_stage1_struct.vhd
analyze -f vhdl -lib WORK ../src/multiplier/fpmul_stage2_struct.vhd
analyze -f vhdl -lib WORK ../src/multiplier/fpmul_stage3_struct.vhd
analyze -f vhdl -lib WORK ../src/multiplier/fpmul_stage4_struct.vhd
analyze -f vhdl -lib WORK ../src/multiplier/fpmul_pipeline.vhd
analyze -f vhdl -lib WORK ../src/multiplier/fpmul_pipeline_inputRegistered.vhd

#to preserve rtl names in the netlist to ease the procedure for power consumption estimation.
set power_preserve_rtl_hier_names true
#elaborate
elaborate FPmul_REGISTERED -arch registered_pipeline -lib WORK > ./elaborate.txt
#uniquify #optional command to addres to only 1 specific architecture
link

#******* Applying constraints ***************
#create 100 Mhz clock
create_clock -name MY_CLK -period 0 clk
set_dont_touch_network MY_CLK

#jitter simulation
set_clock_uncertainty 0.07 [get_clocks MY_CLK]

#input/output delay
set_input_delay 0.5 -max -clock MY_CLK [remove_from_collection [all_inputs] clk]
set_output_delay 0.5 -max -clock MY_CLK [all_outputs]

#set output load (buffer x4 used)
set OLOAD [load_of NangateOpenCellLibrary/BUF_X4/A]
set_load $OLOAD [all_outputs]

#flatten the hiercarchy
ungroup -all -flatten
#implement the pparch multiplier
set_implementation DW02_mult/pparch

#********* Start the syntesis *************
compile > ./analysis_results/compilation_results.txt

#********* Save the results *************
report_timing > ./analysis_results/timing_results.txt
report_area > ./analysis_results/area_results.txt
report_resources > ./analysis_results/resource_report.txt

#Finally, we can save the data required to complete the design and to perform switchingactivity-based power estim
#ungroup -all -flatten
```

The two obtained FP multipliers and the one without forcing the implementation are compared in terms of max frequency and area in the table 1.1.

Multiplier implementation	Max Frequency $[Mhz]$	Area $[\mu m^2]$
Default	636.94	4093
CSA	233.64	4907
Parallel Prefix	641.03	4153

Table 1.1: Comparison of the different multiplier implementations

Comparing the results obtained by the different implementations, it is possible to observe that the Parallel Prefix multiplier brings to the higher **max frequency** ($641.03MHz$) without increase too much the **area occupancy** ($4153\mu m^2$). Instead, the CSA implementation brings to the lowest max frequency and also the worst area. This is related to the fact that the CSA implementation requires an additive RCA to provide the result on the only output. This further element adds an additive delay to the critical path and it is very costly in terms of area.

1.3 Different compilation techniques

The **contents** related to this section can be found in the **branch 'punto1.1/optimization'**

Synopsys gives the possibility to use more than one type of compilation for the design obtaining architectures with different speed but also different cost in terms of computation and time.

- *compile*

  It implements the simplest compilation of the design. Thus, it requires lower computation time but worse results in terms of speed with respect to the others.

- *compile* and *optimize_register*

  The added command permits to optimize registers position and improve the speed of our architecture.

- *compile_ultra*

  It is an advanced compilation that tries as much as possible to improve the speed of architecture but it requires a longer computation time.

In order to exploit the **fine grain pipelining** technique with the *optimize_register* command, a register at the output of the Significands multiplier must be added. To keep correct the timing of the Stage2, other registers on the whole feedforward cutset identified in figure 1.3 are added.

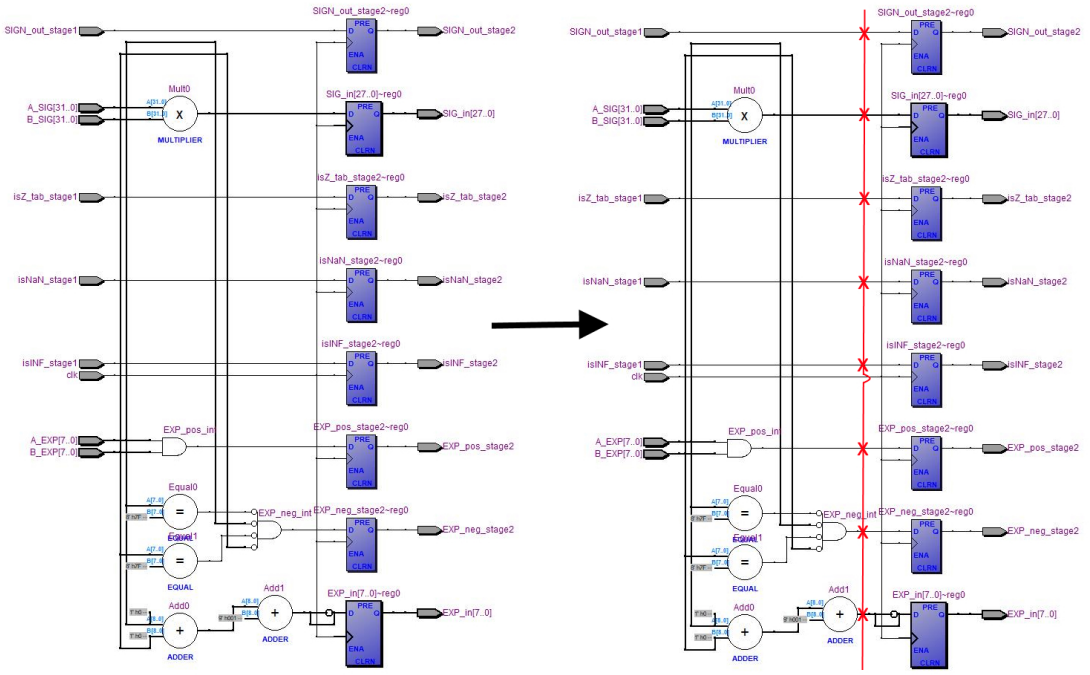

Figure 1.3: Stage2 Netlist View with the FF-Cutset highlited. Registers are added on nodes identified by the red X

The three different compile commands are used to synthesize the design of FP multiplier in case of **Default multiplier** and the results obtained are shown in table 1.2

Type of synthesis	Max Frequency $[Mhz]$	Area $[\mu m^2]$
compile	645.16	4351
compile + optimize_register	961.54	4401
compile_ultra	704.23	4341

Table 1.2: Default Multiplier: Result of different compilation techniques

The table shows that using the *compile* and *optimize_register* commands for this synthesis it is possible to obtain the fastest implementation reaching a max frequency of $961 MHz$ but at the same time a slightly higher occupacy of area $4401 \mu m^2$. In fact, since this architecture is based on the use of a **pipelining** technique, a synthesis direct to optimize register positions is the way to improve the speed. At the end, *compile* and *optimize_register* is the more adapted synthesis way for our purposes.

CHAPTER 2

MBE implementation

2.1 The MBE Multiplier

The **contents** related to this section can be found in the **branch 'mbe'**

Another possible implementation for the significands multiplier is the one based on the **MBE Multiplier**. MBE stands for *Modified Booth Encoding*, which is a technique used for generating the partial products in the multiplication. Here, the operation implied by each sequence of ones is translated into just a sum and a subtraction, like in the following example:

0	1	1	1
2^3	2^2	2^1	2^0

$$\longrightarrow \quad +2^2 + 2^1 + 2^0 = +2^3 - 2^0$$

Moreover, it is a Radix-2 multiplier. This means that two bits of the multiplier are processed at a time: they can produce as a partial product also the double of the multiplicand as indicated by table 2.1.

Table 2.1: Modified Booth Encoding

$x_{2j+1}x_{2j}x_{2j-1}$	p_j
000	0
001	A
010	A
011	$2A$
100	$-2A$
101	$-A$
110	$-A$
111	0

The inclusion of the negative terms (-A or -2A) is done by considering the *sign extension* technique, which brings some modifications in the partial product generation. Here, instead of caluclating the 2's complement of the multiplicand, we simply negate it and then add some special bits to take into account this simplification. An example with a 16 bits multipler is reported in figure 2.1.

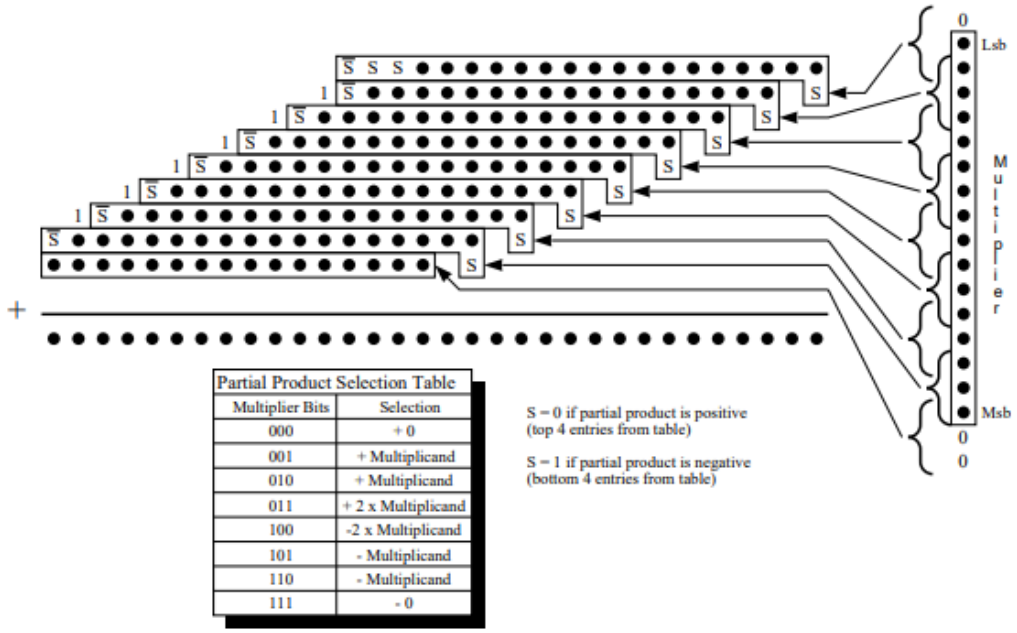

Figure 2.1: Partial products in 16-bit sign extension

The partial products are fed into a sum plane which relies on a Dadda-tree, an "ALAP" approach in which in every layer of the tree has the lowest number of adders possible. The maximum number of operands allowed in each layer l_j is computed by considering the compression capacity of a Full Adder ($3 \longrightarrow 2$):

$$l_j = \lfloor \frac{3}{2} l_{j-1} \rfloor$$

With $l_0 = 2$.
With this equation, the number of operators per stage is computed, reported in table 2.2.

Table 2.2: Maximum number of operands per stage

label	name	# op
l_6	stage 1	19
l_5	stage 2	13
l_4	stage 3	9
l_3	stage 4	6
l_2	stage 5	5
l_1	stage 6	3

The two operand adder in stage 7 ($l_0 = 2$) is implemented with a Ripple Carry Adder, described in section 2.2.4. An high-level block diagram for the whole MBE Multiplier is reported in Figure 2.2.

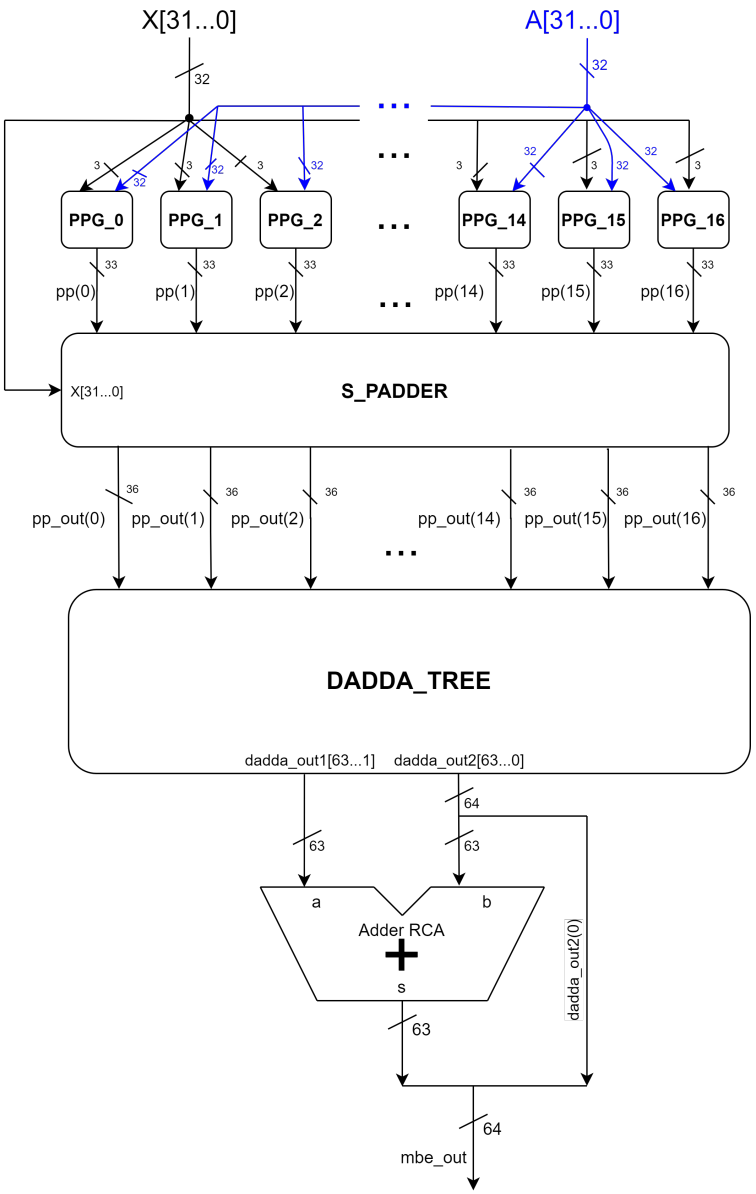

Figure 2.2: MBE Multiplier Block diagram

2.2 Implementation

In this section, each block reported in figure 2.2 is described showing its purpose and working principle.

2.2.1 Partial Product Generator

This block (PPG) is implemented in the file *partial_product_generator.vhd*, and its purpose is to generate the partial product (pp) for each multiplier triplet that has to be processed. The multiplier takes into account two bits, since it's Radix-2 (indicated as $x_{2j+1}; x_{2j}$) plus a third one (x_{2j-1}) because it exploits the Booth Encoding.
Notice that x_{-1} is considered euqal to zero, while $x_{32}; x_{33}$ are set equal to zero too, to guarantee a positive result. With such an algorithm 17 triplets are formed (fig. 2.3): the same number of partial

products must be produced with 17 PPG blocks.

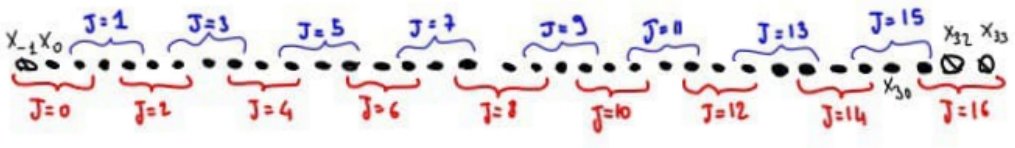

Figure 2.3: 17 triplets used for 32-bit multiplier

In figure 2.4 the PPG architecture is shown: it is basically a multiplexer implementing the table 2.1 with the aforementioned modifications. Since A is a 32-bit multiplicand and the output can be 2A in some cases, the PPG output is on 33 bits.

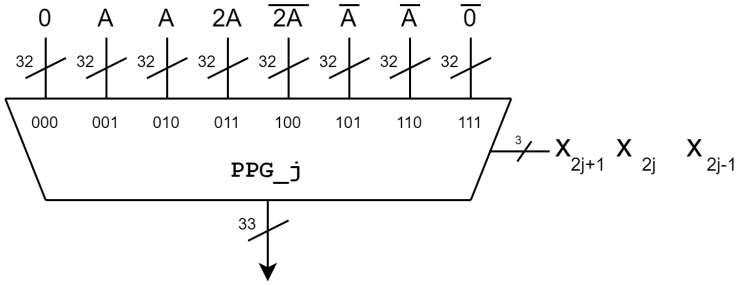

Figure 2.4: *j-th* instance of Partial Product Generator Block

2.2.2 S_Padder

The purpose of this block is just to add the correct bits to implement the sign extension (similarly as showed in figure 2.1). In figure 2.5 a more detailed view on the block is presented.

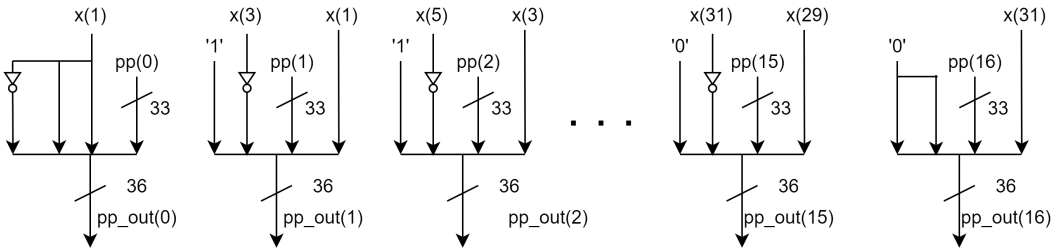

Figure 2.5: S_Padder block

2.2.3 Dadda Tree

The Dadda Tree is made up by 6 layers of adders, indicated as *stage1* to *stage6*. In each stage the HAs and FAs are organized in the appropriate way using the *Excel* software. This organization is showed in figures 4.1 to 4.6 in the *Appendix* section.
Each stage is implemented in a different VHDL code (*dadda_stage1.vhd* to *dadda_stage6.vhd*) and put together in the top-level entity *dadda_tree.vhd*

2.2.4 RCA63

The Dadda tree gives two outputs: one on 64 bits and the other on 63 bits. The LSB of the first one is already the LSB of the multiplication result, while the remaining 63 must be added with the second output of the tree. This is done with a 63-bit RCA (implemented in the file *rca63.vhd*) whose carry out bit is left floating (it would be the 65th bit of the final result).

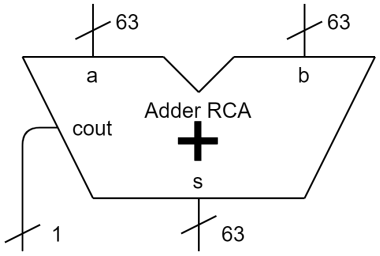

Figure 2.6: 63-bit RCA

2.3 Simulation

With the significands multiplier implemented with this MBE technique, a simulation of the system has been performed. The simulation results confirm that the model is still **working correctly**. In figure 2.7 a postscript of the *wave* window is showed.

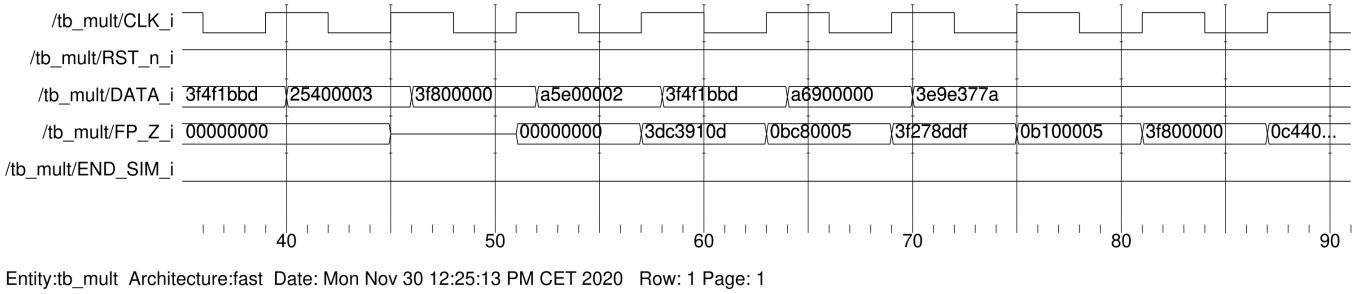

Figure 2.7: Simulation of the MBE-based fp architecture

In report 2.1 the simulation results printed by the tesbench are exposed.

Report 2.1: FPU MBE-based Simulation Results

```
INPUT, OUTPUT, EXPECTED OUTPUT,TEST RESULT

0,          0,      0,                    OK
1050556282,1036226829,1036226829,OK
631242754,197656581,197656581,OK
1062149053,1059556831,1059556831,OK
624951299,185597957,185597957,OK
1065353216,1065353216,1065353216,OK
-1512046590,205783044,205783044,OK
1062149053,1059556831,1059556831,OK
-1500512256,228720640,228720640,OK
1050556282,1036226829,1036226829,OK

SIMULATION ENDED SUCCESSFULLY
```

2.4 Synthesis

After the Simulation, the synthesis of the FPU with the MBE has been performed. This is done with different commands to have a complete view about the possible solutions. In table 2.3 the performance of the implementation in terms of clock frequency and area among different compilation commands are reported.

	Max Frequency $[Mhz]$	Area $[\mu m^2]$
compile	223,21	5149
compile + optimize_register	1149,43	8100
compile_ultra	617,28	5655

Table 2.3: FPU MBE-based compile with different commands

What can be observed is that the most costly compilation in terms of area is the one with the *compile + optimize_register* command, but it's also the fastest one (roughly 5 times faster than the slowest one).
On the contrary, the less costly version is the one generated with the "plain" *compile* command (about 1.5 times smaller than the biggest one), but it's also the slowest. Based on the required system constraints (speed or area) the choice of the most appropriate version can be done.

CHAPTER 3

Results Summary

In table 3.1 all the different implementations for the significands multiplier are compared.

	Max Frequency MHz	Area μm^2
No Fine Grain Pipelining		
Default Architecture	636,94	4093
CSA multiplier	233,64	4907
PP arch	641,03	4153
MBE	223,21	5149
Fine Grain Pipelining: compile + optimize registers		
MBE	1149,43	8100
Default Multiplier	961,54	4401
Fine Grain Pipelining: compile_ultra		
MBE	617,28	5655
Default Multiplier	704,23	4341

Table 3.1: Results Summary Table

Without applying fine grain pipelining, the fastest multiplier is the *PP arch* one, and it's also quite cheap in terms of area occupation (only 1% bigger than the smallest one).
Applying the fine grain pipelining, great improvements in terms of speed are achieved in the MBE architecture. With its $1.15GHz$ of speed, it's the fastest multiplier, but also the most costly in terms of area ($\simeq$ 100% larger than the smallest one).
A good compromise between cost and performance could be the default multiplier internally pipelined: much close to $1GHz$ and just 7% bigger than the smallest implementation.

CHAPTER 4

Appendix

4.1 Dadda Tree Stages

Figure 4.1: Dadda Tree - Stage 1

Figure 4.2: Dadda Tree - Stage 2

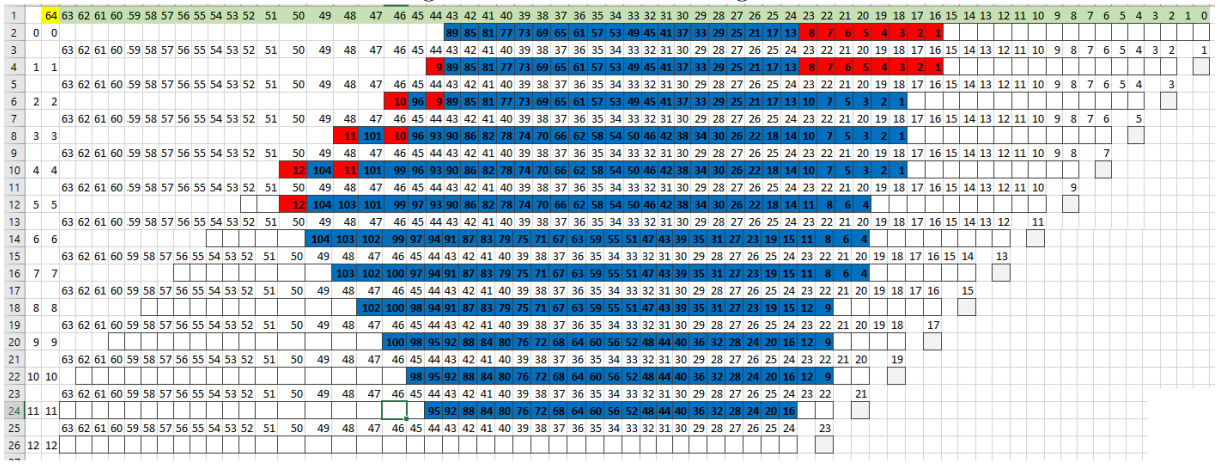

Figure 4.3: Dadda Tree - Stage 3

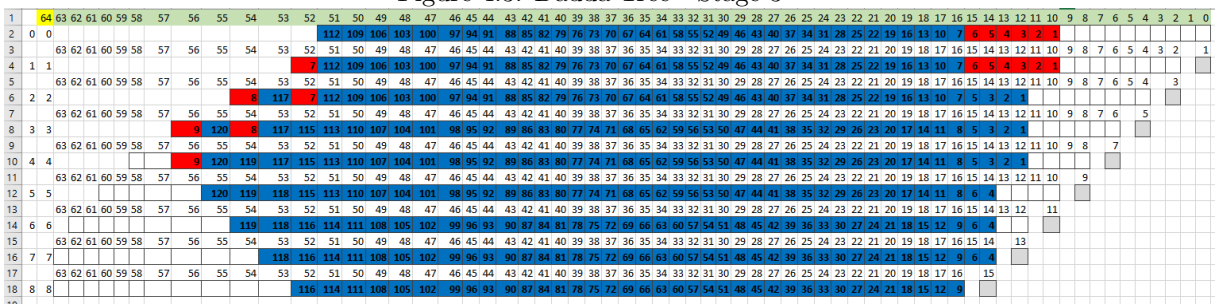

Figure 4.4: Dadda Tree - Stage 4

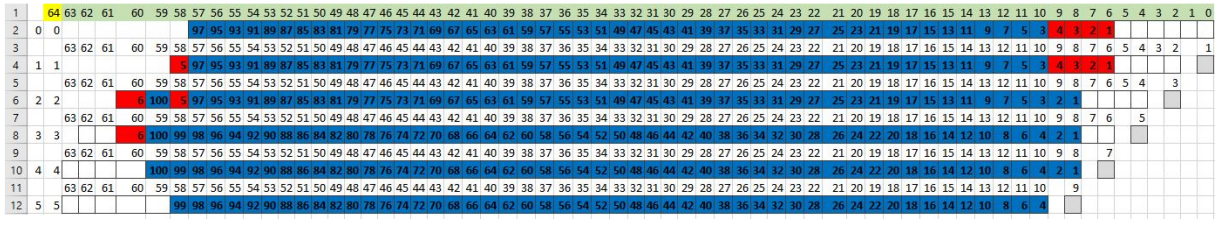

Figure 4.5: Dadda Tree - Stage 5

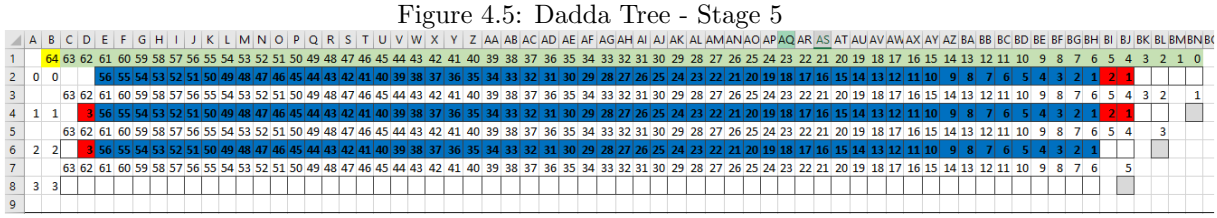

Figure 4.6: Dadda Tree - Stage 6 and its output (the 65th bit is discarded)