Politecnico di Torino

Politecnico di Torino

Integrated System Architecture

Digital Arithmetic

Second Lab Report

`https://github.com/aledolme19/ISA_LAB.git`

Francesco Babbaro Pierfrancesco Boccardi Alessandra Dolmeta

2021-2022

1 Introduction

Multiplier plays an important role for performing the arithmetic operations. In digital systems, multipliers are the one of the basic blocks and consequently are one of the most important contribution to the computational speed and power consumption of the digital system. So,the need for designing high speed multiplier with minimal power dissipation is very crucial for a digital system. Thus, it can boost the efficiency of the digital systems.

The aim of this laboratory experience is testing and verification through simulations and logic synthesis of a given floating point pipelined multiplier, which must be properly modified according to the required specifications.
The task of the laboratory can be divided into two major tasks with several sub-goals:

1. **Analysis of the given Pipelined Floating Point Multiplier**:
   - Verification of the given floating Point multiplier by ModelSim simulations through a developed testbench;
   - Assignment of registers at the input of the multiplier, flatten the hierarchy and re-verification through ModelSim simulation;
   - Force the Design Compiler to use CSA and PPARCH multiplier architectures in place of behavioral descriptors of multipliers and then verify the whole design;
   - Fine grain pipelining by placements of registers on the output of the significands multiplier stage to improve the basic architecture by exploiting the re-timing procedure of the Logic Synthesizer;
   - Analysis of maximum operating frequency and occupied area by Synopsys Design Compiler logic synthesis.
2. **Designing of Modified Booth Encoder Multiplier in VHDL**:
   - Modification of the basic architecture by substituting the multiplication of the two input significands by a 32 bits Modified Booth Encoder in radix-4 multiplier, whose adder plane relies on a full Dadda tree;
   - Verification through simulations and synthesis of developed architecture.

2 Description

The pipelined floating point multiplier analysed works with 32 bits operands, exploiting the single precision floating point format.
The size of each field of the input data is the following:

- 1 bit for the sign;
- 8 bits for the exponent value;
- 23 bits for the mantissa;

During the verification step, both operands are provided by the testbench in **hexadecimal representation**. These bits are sent to the `unpack` unit, whose purpose is to separate each field of the floating point number.
The final sign bit is obtained by `EXOR-ing` the two sign bits, using the `XOR` unit represented in Figure 1.
The **exponent** must be handled properly since during the mantissa multiplication a non-normalized value can be obtained, so at the end a **normalization** has to be performed to achieve the right result. As a consequence this operation affects the value of exponent. In order to perform a multiplication between two floating point numbers that follows the standard IEEE-754, exponents have to be **added** together. However, due to the fact that both exponents are biased, a `-127` subtraction is performed to removed the bias, otherwise it would be present twice in the final result.
As far as the **significands**, they are **multiplied** together using an **unsigned multiplier**. However to do that we have to keep in mind that in the significand field (23 bits) only the fractional part of the normalized number is present, so in order to correctly perform the multiplication a `'1'` must be considered as 24^{th} bit. Nevertheless, the resulting product could be a

non-normalized number. Typically a normalized number can be seen as a fractional number, therefore a fixed point number with only one integer bit and all the other fractional bits. After the multiplication, therefore it can happen that the integer part of result requires two bits rather than only one, so in this case the **normalization** must be performed. For this purpose the exponent value must be incremented of 1 unit.
Since the result's mantissa have to be always on 23 bits, whereas at the output of multiplier we have a number on 48 bits, a **rounding** operation must be performed, that is executed in two "phases". In a first step, the output of the significant multiplier is truncated and only 28 MSBs `(47 downto 20)` are sent to the first normalize stage. After the normalization, the rounding stage performs a *round to nearest number* on 25 bits: this operation could bring to a denormal number, requiring an additional normalization just before the packing.
After all these operations the 32 bits are put together in the `pack` unit.

The basic structure of the FP Multiplier is shown in Figure 1.

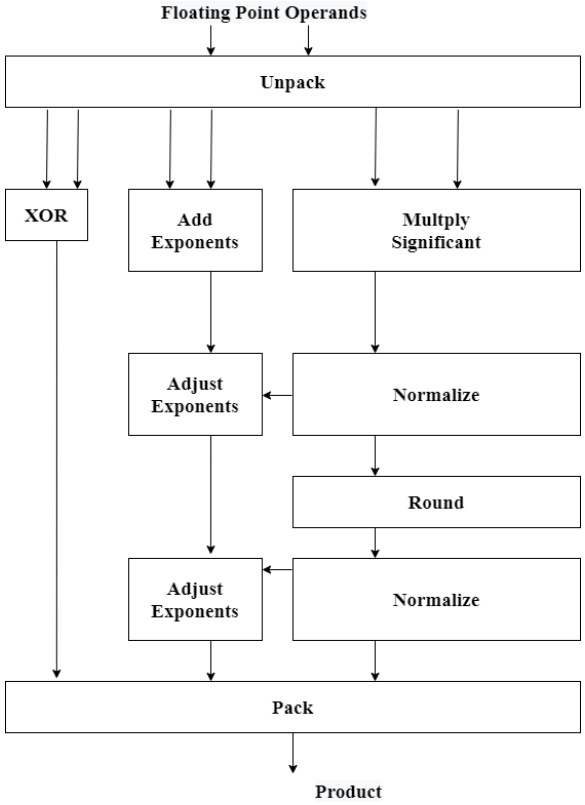


Figure 1: Basic scheme for the floating point multiplier

3 Simulations and Synthesis

3.1 Simulation

Simulation without input registers

In order to perform simulations, a testbench is developed to provide data and to monitor the results. The output results of VHDL simulation are compared with the output result of reference model to verify its correctness, as shown in Figure 2.

#FP products	#FP output Modelsim
00000000	00000000
3dc3910d	3DC3910D
0bc80005	0BC80005
3f278ddf	3F278DDF
0b100005	0B100005
3f800000	3F800000
0c440004	0C440004
3f278ddf	3F278DDF
0da20000	0DA20000
3dc3910d	3DC3910D

Figure 2: Comparison between provided and obtained results

The system works as expected. The testbench produces a file, named "*output_results.hex*", containing the results of simulation written in hexadecimal representation.
It is important to point out that in the mantissa multiplication stage, there are four pipeline stages which delay the data by four clock cycles. There are no additional rows of zeros present at the beginning of the output file, since the testbench is designed in order to write the outputs after four clock cycles, when the pipe queue is full.
All the VHDL files and the results are reported in the GitHub folder `/1_version_mult_pipe`.

Simulation with input registers

Before proceeding to the logic synthesis, it is necessary to insert an input stage at the top entity of the floating point multiplier, as shown in Figure 3.
To verify the architecture with the addition of the two registers, also in this case, a simulation is performed on ModelSim Altera.
Once again, the output results of VHDL simulation are compared with the ones of the reference model as a verification step. The system works again correctly.
All the VHDL files and the results are reported in the GitHub folder
`/2_version_mult_pipe_input_registers`.

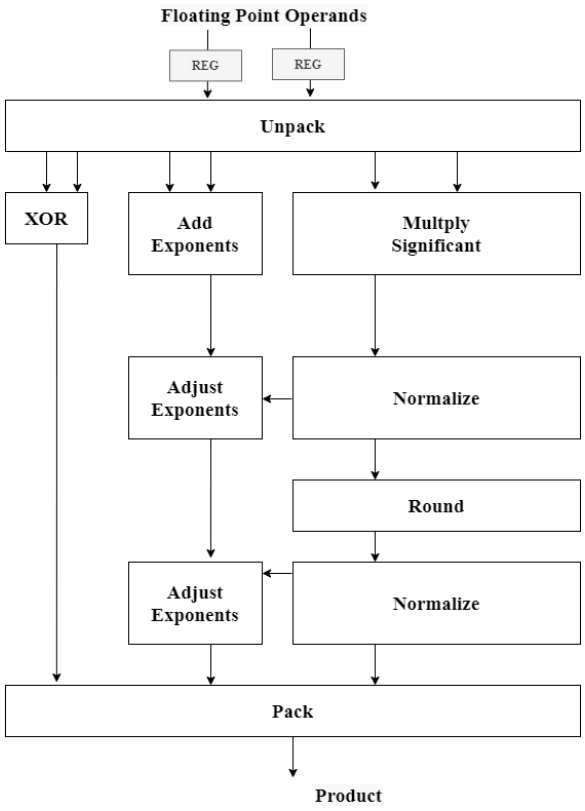


Figure 3: Basic scheme for the floating point multiplier with input registers

3.2 Logic Synthesis

It is required to perform three different logic synthesis, forcing *Design Compile* to use different implementations for the **multiplier**:

- Logic Synthesis without forced implementation;
- Logic Synthesis forcing the **CSA** implementation;
- Logic Synthesis forcing the **PPARCH** implementation.

The three different synthesis are reported in the following subsections and then compared in the subsection 3.2.
All the following results in terms of area and power have been obtained with a synthesis forced with 0 ns clock period and then the set-up of the actual clock period required by the specific architecture (without re-synthesis).

Logic Synthesis without forced implementation

First of all, we need to find the maximum clock frequency of our design, forcing a 0 ns clock period. Applying a **null-clock** to a purely combinational architecture corresponds to creating a virtual clock signal, which is associated to a not real clock pin. Therefore, the optimization potentialities of the synthesizer are quite constrained. This is evident from the obtained results in terms of delay and power.
In order to get the **maximum clock frequency** f_M the Design Compiler must be forced to use a clock with a period equal to zero, in this way the total time required by the architecture to perform all operations in the critical path will be shown. In particular:

$$t_{CLK \to Q} + t_{comb} + t_{setup} = -t_{slack}$$

Since the slack is defined as positive, its value will be certainly negative, in fact the Design Compiler reports a slack equal to -1.65 ns. This is the **critical path**.
The maximum allowed frequency is:

$$f_M = \frac{1}{t_{cp}} = 606.6 \text{ MHz}$$

In terms of **area**, with a synthesis with 0 ns clock, the circuit we obtain has a cell area equal to 4135.50 μm^2, which is divided into:

- Combinational area: 3032.93 μm^2
- Buf/Inv area: 233.28 μm^2
- Non-combinational area: 1102.56 μm^2

Logic Synthesis forcing the CSA implementation

Now, we are required to force the Design Compiler to use the implementation of the CSA for the second stage of the multiplier.
This can be done by inserting the command `set_implementation`, which forced the Design Compiler to use a CSA multiplier, taken by a collection of pre-made blocks.
With the same iterative method used before, a slack equal to -3.911 ns is obtained.
The maximum allowed frequency is:

$$f_M = \frac{1}{t_{cp}} = 255.69 \text{ MHz}$$

In terms of **area**, with a synthesis with 0 ns clock, the circuit we obtain has a cell area equal to 4925.25 μm^2, which is divided into:

- Combinational area: 3825.87 μm^2
- Buf/Inv area: 154.01 μm^2
- Non-combinational area: 1099.37 μm^2

Logic Synthesis forcing the PPARCH implementation

Now, we are required to force the Design Compiler to use the implementation of the PPARCH for the second stage of the multiplier.
With the same iterative method used before, a slack equal to -1.43 ns is obtained.
The maximum allowed frequency is:

$$f_M = \frac{1}{t_{cp}} = 699.3 \text{ MHz}$$

In terms of **area**, with a synthesis with 0 ns clock, the circuit we obtain has a cell area equal to 4083.63 μm^2, which is divided into:

- Combinational area: 2978.40 μm^2
- Buf/Inv area: 223.97 μm^2
- Non-combinational area: 1105.22 μm^2

Comparison between the Logic Synthesis

A comparison between the two logic synthesis is shown in Table 1.

Design	f_{ck}[MHz]
Flattened	606.06
Flattened + CSA multiplier	255.69
Flattened + PP multiplier	699.3

Table 1: Comparison between the Logic Synthesis

As it can be seen from Table 1, **the faster architecture is the one obtained by forcing the PPA implementation**. In particular, the first and the third architecture achieve frequencies which are quite similar, while the second one is slower. This was predictable, since by definition, the PPA architectures are designed in order to achieve a wide range of design trade-offs primarily in terms of speed and secondly in terms of area, power consumption and regularity.
Analyzing the *resource reports*, without forced implementation the **multiplier has been implemented automatically with a parallel prefix architecture**. The implementation adopted by the Design Compiler is used to improve the performance of that stage, since the parallel prefix architecture allows to operate on all bits to make multiplication faster.
For the logic synthesis performed by forcing the PPARCH, the PPARCH implementation is the same used by default by *Design Compiler* when no commands are given. However, forcing Synopsys to use the parallel prefix architecture does not means that it will use the same optimized architecture as in the previous section, but different radix form will be used.
In particular, as it can be seen from the resource reports shown in Figure 4, in the first case, radix-8 is used, while in the second case radix-4 is used.

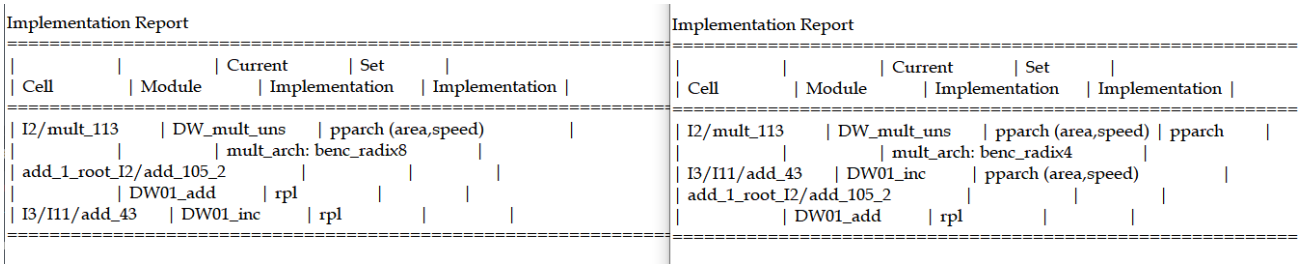


Figure 4: Resource reports

The implementation adopted by the Design Compiler is used to improve the performance of that stage, in fact a radix-8 algorithm reduces the number of sum-shift operations from 32 to 11 in this case. In the third case, the radix-4 algorithm reduces the number of sum-shift operations from 32 to 16.
Generally, the name radix-r suggests that, for each iteration, the block processes $\log(r)$ bits for each operand, speeding up the execution, since the evolution of the a general algorithm can proceed processing more than one bit at time. However, improving the throughput of the circuit does not imply a reduction of the critical path because more complex structures are needed to correctly manage an higher number of bits at a time. In fact, looking at the result obtain comparing the first synths and the third, we obtain an faster circuit considering the radix-4 implementation of the third case.
For what concerns the CSA implementation, from the performance point of view, the maximum operating frequency drops because of the long path of adders which connects an input bit with the output. Since no fine-grain pipeline is considered in this section, obtaining a lower clock frequency with respect to PPA architectures is more than reasonable.

In terms of **area occupation**, as expected, the floating point multiplier is very expensive, independently from the architecture of the multiplier which is actually used.
This is due to the large number of operations required to perform significant multiplication, to proper handle normalization, exponent adjustment, rounding and so on.
In Figure 5 a comparison between the three different total areas obtained is reported.

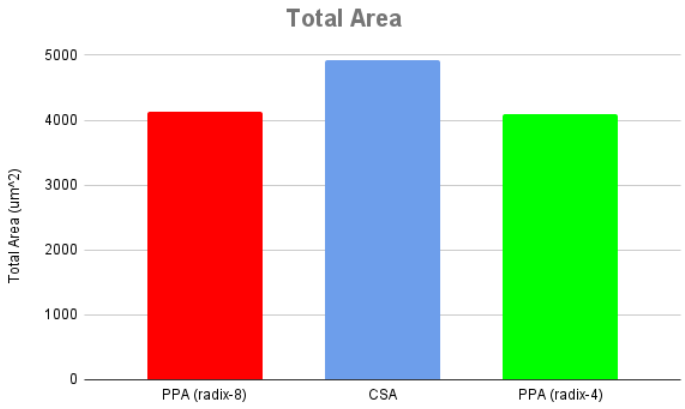


Figure 5: Total area comparison

The worst area report is the one of the Carry-Save-Adder implementation.
The amount of required area for this configuration is raised up to 4925.25 μm^2, which is slightly larger than the other ones. This is due to the structure of the architecture itself.
In particular, the CSA architecture allocates a full adder or half adder in a tree-like structure. Instead of connecting two adjacent adders together, they simply provide the sum in the usual notation and the carry bit is delayed to an adder of a lower level of the tree, in this way carry and sum bits are treated as vectors and summed row by row.
The main advantage of this architecture is that one half adder or full adder of the same row does not wait for the carry propagation and can start immediately the computation as the operands are ready. The main drawback is the high cost of components. As demonstrated with this short explanation, the higher the number of bits, the larger the area occupied.
In terms of area, it is also true that even though PPA architectures are mainly designed in order to achieve better results in terms of speed rather than in terms of area, their total area is slightly lower with respect to the CSA case.

4 Fine-grain Pipelining and Optimization

There are different methods to improve the performance of a given system at the RTL level. One of those is the pipelining, which reduces the critical path by breaking combinational paths. We can apply this technique to the *significant multiplier*, that is the bottleneck component in our case. For this reason a register was inserted at the output of the multiplier in order to allow Design Compiler to employ **re-timing**, moving automatically that register in the most suitable place within the architecture.
The basic structure of the FP Multiplier adding one register at the output of the significant multiplier and all the required registers such that the timing of the whole *stage2* is correct, is shown in Figure 6.

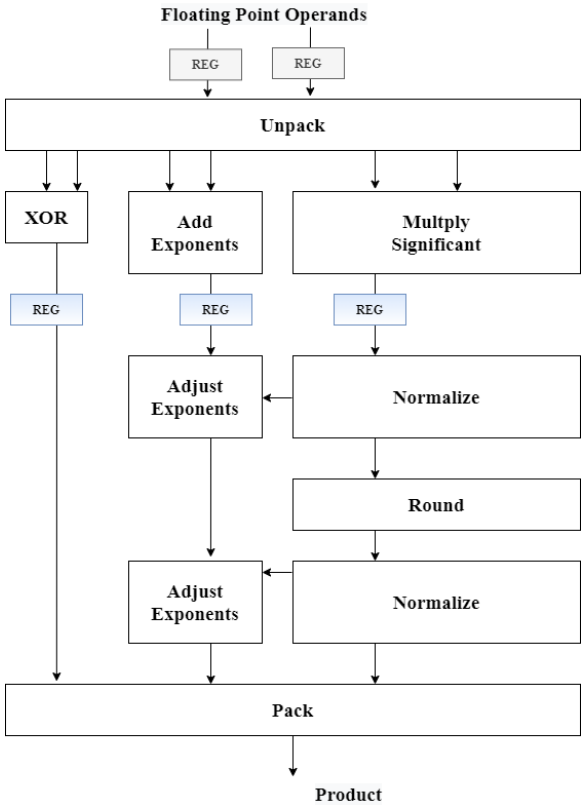


Figure 6: Basic scheme for the floating point multiplier with the inputs register and 1 level of pipeling

The timing diagram is shown in Figure 7. Here, we can notice that 5 clock cycles are needed to execute the operation. This 5 cycles are related to the input register introduced by us (input grey registers), the layer of registers which has just been added (blue registers) and by the ones already present in the original architecture.

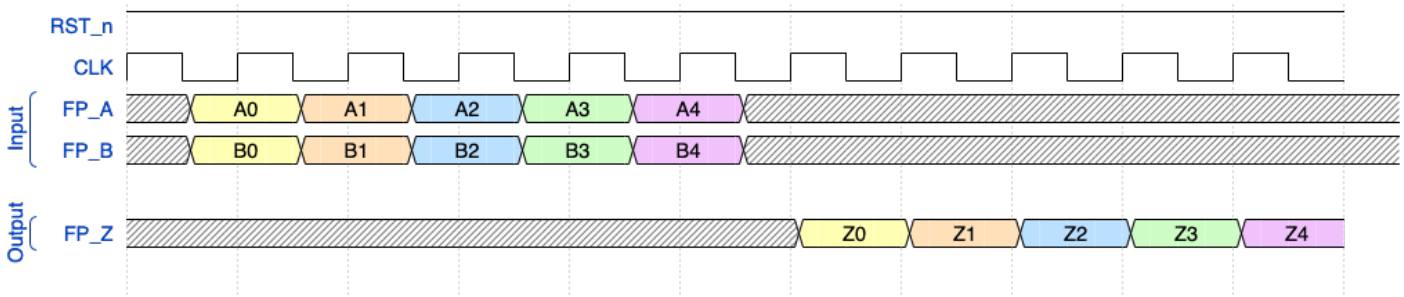


Figure 7: Timing diagram of the fine-grain pipelined version.

4.1 Logic Synthesis

It is required to perform two different logic synthesis:

- Logic Synthesis with `compile -exact_map` and `optimize_registers` commands.
  The last command is used to perform re-timing on a mapped gate-level netlist. It determines the placement of registers in a design to achieve a target clock period and minimizes the number of registers while maintaining that clock period.
- Logic Synthesis with `compile_ultra` command. This performs a two-step high-effort compile flow on the current design to achieve better results.

The two different synthesis are reported in the following subsections and then compared in subsection 4.1.

Logic Synthesis with `compile -exact_map` and `optimize_registers` commands

The first synthesis has been performed using the commands `compile -exact_map` and `optimize_registers`.
With the same iterative method used before, a slack equal to -0.874 ns is obtained.
The maximum allowed frequency is:

$$f_M = \frac{1}{t_{cp}} = 1.14\text{GHz}$$

In terms of **area**, with a synthesis with 0 ns clock, the circuit we obtain has a cell area equal to 5888.71 μm^2, which is divided into:

- Combinational area: 3708.84 μm^2
- Buf/Inv area: 237.80 μm^2
- Non-combinational area: 2179.87 μm^2

Logic Synthesis with `compile_ultra` command

The second synthesis has been performed using the command `compile_ultra`.
With the same iterative method used before, a slack equal to -1.502 ns is obtained.
The maximum allowed frequency is:

$$f_M = \frac{1}{t_{cp}} = 665.78\text{MHz}$$

In terms of **area**, with a synthesis with 0 ns clock, the circuit we obtain has a cell area equal to 4371.71 μm^2, which is divided into:

- Combinational area: 3086.13 μm^2
- Buf/Inv area: 187.53 μm^2
- Non-combinational area: 1285.57 μm^2

Comparison between the Logic Synthesis

A comparison between the two logic synthesis is shown in Table 2.

Design	f_{ck}[MHz]
`compile -exact_map` + `optimize_registers`	1140
`compile_ultra`	665.78

Table 2: Comparison between the Logic Synthesis

The first synthesis allows to achieve better results in terms of performance, leading the synthesizer to adopt retiming. On the other hand, area is larger, as shown in Figure 8.
Instead, using the `compile_ultra` command the increase of area is smaller than the previous case, but also the achieved frequency is lower.
It is related to the fact that this command allows a concurrent optimization of timing, area and power, while exploiting retiming the synthesizer moves registers along the critical path, in order to minimize it. Therefore, `optimize_registers` command brings to a better results in terms of **delay**.
In Figure 8 a comparison between the two different total areas obtained is reported.

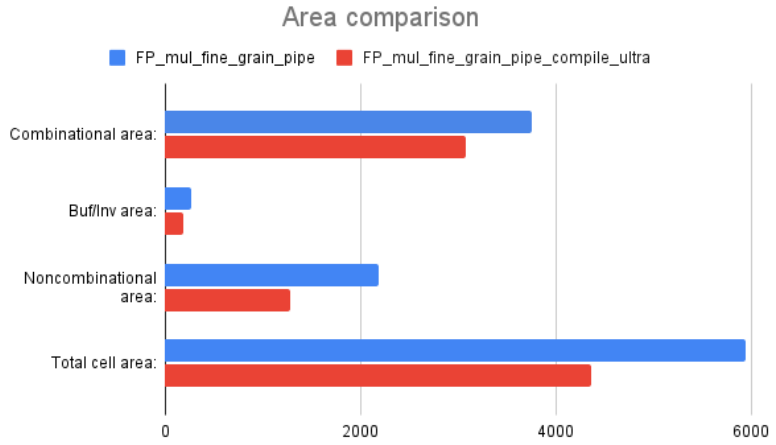


Figure 8: Total area comparison

As can be seen from Figure 8, the area of the multiplier with the `optimize_registers` commands is 26% higher respect to the one obtained in the case of the `compile_ultra` command.

5 Optimization of the multiplier

For the sake of curiosity, we try to optimize the overall structure of the floating point multiplier by adding different levels of pipeline, exploiting then the `optimize_registers` command. A comparison between the different logic synthesis is shown in Table 3.

Level of pipe	**Design**	T_{clock}ns	f_{ck}[MHz]
2	`compile -exact_map + optimize_registers`	0.83	1204
2	`compile_ultra`	1.61	621
3	`compile -exact_map + optimize_registers`	0.68	1470
3	`compile_ultra`	1.40	714

Table 3: Comparison between the Logic Synthesis

For the example we have analysed, the higher is the number of registers added the higher will be the possible operating frequency, as shown in Table 3.

6 Design of a MBE multiplier with Dadda tree

We are requested to optimize the Stage 2 of the floating point multiplier by designing an unsigned multiplier which exploits a modified version of the Booth-Encoding with a Dadda tree based adder plane.
First, the unsigned multiplier has been design on 24 bits, not on 32 bit. This is related to the fact that the unsigned multiplier is required as a multiplier for the significands, which are extendend on 24 bits only (integer bit included). This will simplify the design of our structure, reducing area and power consumption.

6.1 Modified Booth Encoding

One of the main advantage of the Booth encoding is to lighten the computation by reducing the number of ones in a bit stream, so that it is possible to reduce the number of partial products to sum. This result is achieved by defining two elements of a bit sequence of any length:

- the head is the first one of the sequence (starting from the MSB); it is encoded as `0` and the bit just on its left (which is a zero) becomes a `1`;
- the tail is the last one of the sequence (starting from the MSB), it is encoded as `-1`;
- all the bits between the tail and the head becomes `0`.

Design constraints require a **radix-4 Booth encoding** and multiplication must occupy only one clock cycle, therefore **3 bits is the length of a sequence and instead of taking a single stream from multiplier register per clock cycle**, all 24 bits are taken and 13 sequences of 3 bits each are obtained, as shown in Figure 9.

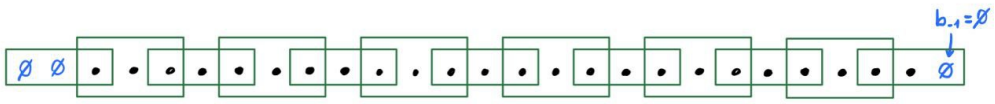


Figure 9: 13 sequences obtained from 24 bits

To derive the 13 sequences, one has to start from the bit in position `i=0` (LSB) and take the left and right bit. Then, to get a new sequence, the bit in position `(i+2)` is taken and grouped up with the one in position `[(i+2)+1]` and `[(i+2)-1]`. It is important to point out that the first sequence is made of bits in position `[1 0 -1]`, and since the bit in position -1 does not exist, this is fixed to 0.
Moreover, to take the last partial products, a zero padding extension must be performed. By doing so, the last partial product is never negated, because the 0-padding ensures that it is always positive. This will be exploited in the following sections in order to simplify a little bit the overall structure.
Once all 13 groups are obtained, it is possible to derive the partial products according to the Table 4.

B_{2j+1}	B_{2j}	B_{2j-1}	p_j
0	0	0	0
0	0	1	a
0	1	0	a
0	1	1	$2a$
1	0	0	$-2a$
1	0	1	$-a$
1	1	0	$-a$
1	1	1	0

Table 4: Modified Booth Encoding

6.2 Dadda tree

Column compression multipliers have gained popularity because of their high-computational speed.
Wallace and Dadda multipliers are the well renowned column compression multipliers. Both these Wallace and Dadda multipliers are reduction based. The reduction is achieved by compressing the columns with a **[3, 2] counter** (full adder) and a **[2, 2] counter** (half adder). The three stages that are involved in both Wallace and Dadda Multipliers are similar.

Dadda tree is a famous structure for adding multiple operands. It uses a selection of full and half adders to sum the partial products in stages (the Dadda tree or Dadda reduction) until only two operands are left. As in the Wallace tree structure, the idea is to exploit the so-called **partial addition by columns**.
Their first step is to align data so that partial terms with the same weight are in the same column.
However, the Dadda tree takes the opposite approach (**ALAP**) with respect to Wallace tree and it allocates FAs and HAs only when needed. In fact, given that for each level the maximum achievable reduction is 3/2, it only uses FAs and HAs to reach the required height for the next stage. This brings to a lower complexity, but an higher parallelism for the two final vectors, bringing an higher complexity in the next sum stage.
In Wallace Multiplier the reductions are done **ASAP** in the layer, therefore the number of FAs/HAs required by Wallace multiplier is quite high in number, while Dadda Multiplier requires less of them.
When a comparison is carried between them, the **Dadda Multiplier requires less hardware than the Wallace** and it is also **slightly faster in computation than Wallace**.
In Figure 10 it can be noticed that partial products are summed together according to the Dadda tree structure. This tree is divided in layers and each layer has an height equal to the maximum number of dots in a column. The key point of any kind of partial or full tree structure is to start with a certain height, 13 in our case, and to reduce this number to 2, which is the number of bits used to do a simple addition using conventional 2-operand adder structures.
Dadda manages to do it by placing **[3,2]** and **[2,2] compressors** to reduce height where needed. In other words, 2/3 bits enters in those components and 1/2 bits exit (sum and carry out). The compressed output is then moved in the layer below.

The maximum height for each level can be computed according to the following equation in an iterative way:

$$l_j = \left(\frac{3}{2} \cdot l_{j-1} \right)$$

where j must be greater than zero and $l_0 = 2$, since the first layer l_0 (starting from the bottom of the figure) must have an height equal to two in order to be the input of a conventional 2-operand adder. Maximum height for each layer can be computed following the previous formula, as shown in Table 5.

j	l_j	required HAs	required FAs
0	2	0	0
1	3	3	43
2	4	3	40
3	6	6	68
4	9	9	72
5	13	12	40

Table 5: Dadda Tree Layers Table for $N_{bit} = 24$.

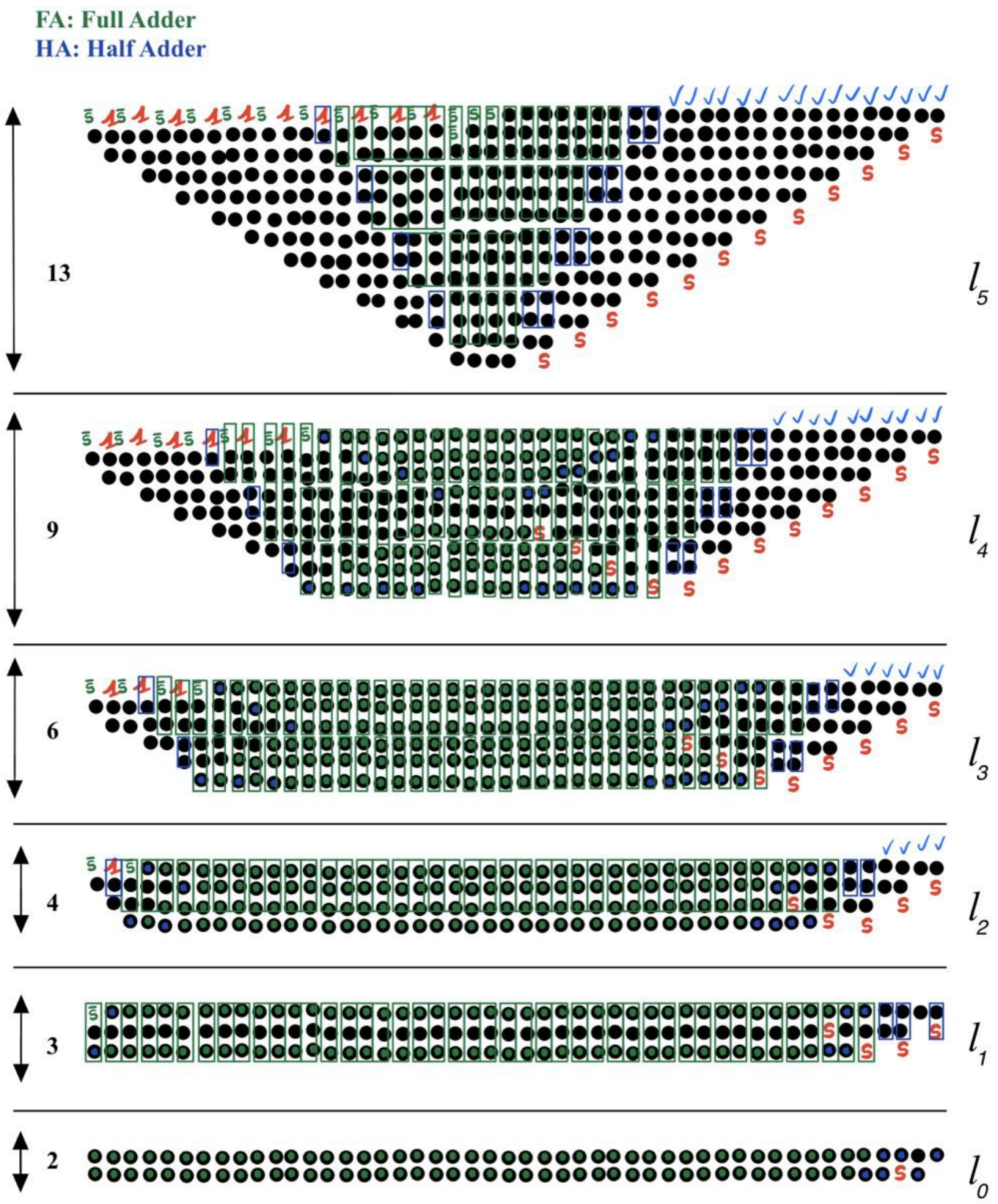


Figure 10: Dadda Tree

Since **Dadda Tree** follows an **ALAP** approach half adders and full adders are placed along each column so that in the next layer there is the maximum number of bits allowed, as reported in Table 5. However, we must consider that any adder will produce a sum bit in the same column and a carry out bit in the next one, both of them are moved directly in the next layer.
The goal of this algorithm is to reduce the number of bits to be summed in a column down to two. Finally, a standard carry propagate adder can be exploited to compute the final sum (e.g. RCA).
For instance, if in a column in the layer l_5 there are 10 bits, one half adder is enough to reduce them, but it must be taken into account that other bits may come from carries deriving from FAs or HAs placed in the previous columns.

In Figure 10 the dot tree is shown. Here:

- **black dots**: bits coming from the partial products;
- S and $\overline{S}$: they state if an addition (`S=0`) or a subtraction (`S=1`) has to be made. Notice that if `S=1`, the 1's complement of the operand to be subtracted is performed by a `XOR-plane`. `S` corresponds to the MSB of the considered 3-bit sequence coming from Booth Encoding (Table 4).
- **1**: needed to correctly perform sum/subtraction.

The latter ones are reported because they represent the +1 operation required for the 2's complement conversion; in this way, even if input operands are unsigned, the Booth encoding generates signed numbers which are then properly handled.
The reasons of the placement of S, $\overline{S}$ and 1 are explained with more details in the document provided for the lab.

6.3 Final Structure

The final structure is shown in Figure 11.

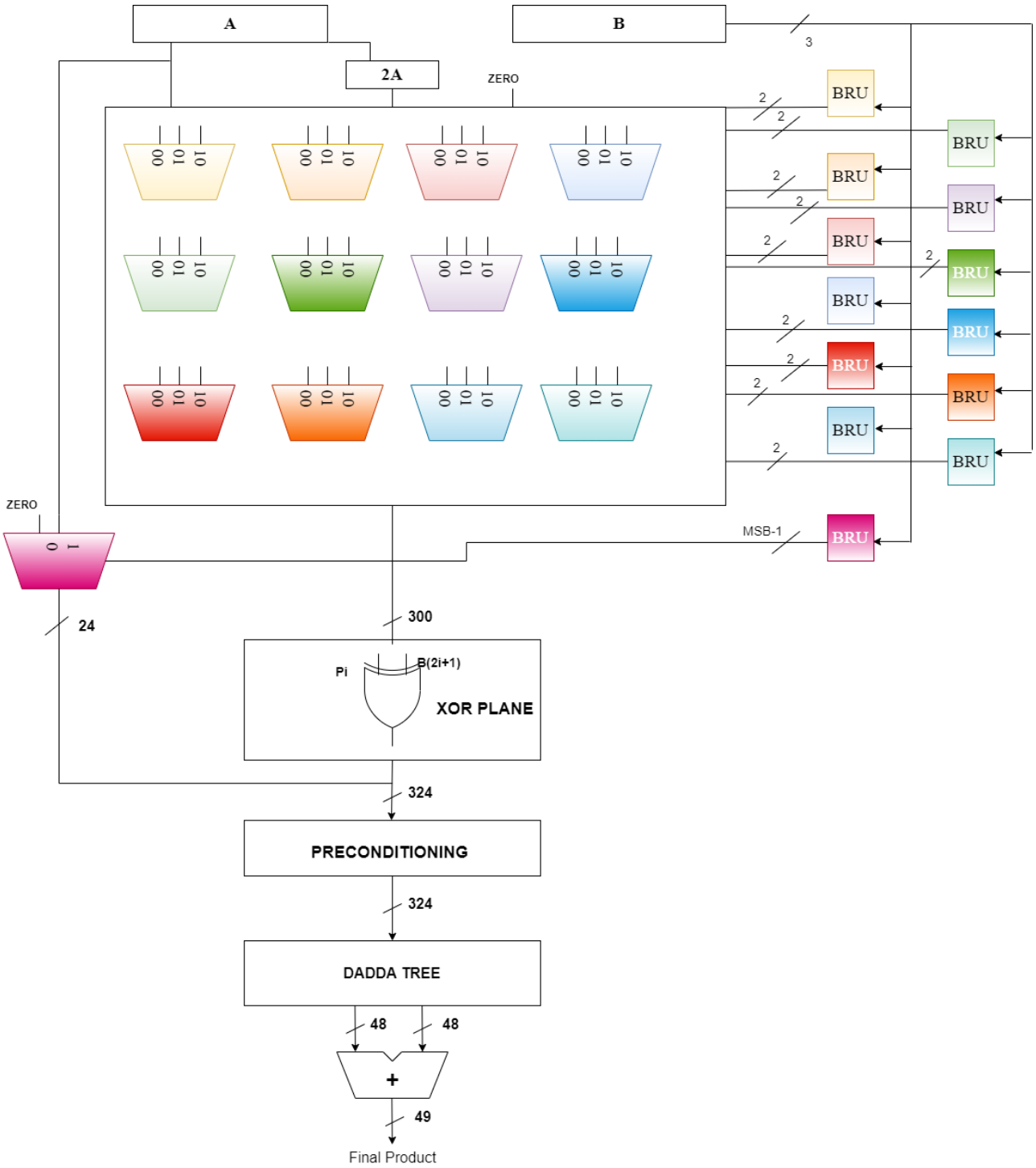
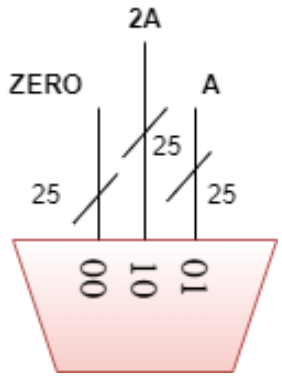


Figure 11: MBE final structure

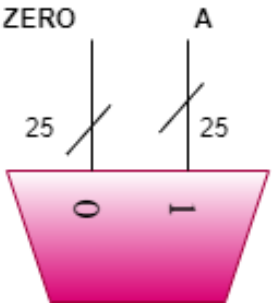
The multiplicand A and the multiplier B are taken by two input registers. $2A$ is obtained by simply shifting of one bit toward the left A.
From the 24 bits of B, 13 sequences of 3 bit each are obtained, as shown in Figure 9, and sent to 13 `BRU` units. The `BRU` units have been design in a behavioral way in order to implement Table 4.
Then, 13 multiplexers are allocated because a total of 13 partial products must be computed. However, as shown in Figure 11, two different multiplexers have been used:

- twelve **3to1-multiplexers**;
- one **2to1-multiplexer**.

The first twelve multiplexers are connected to the three possible inputs as shown in Figure 12a.



(a) 3to1-multiplexer structure



(b) 2to1-multiplexer structure

Figure 12: Multiplexers' structures

As explained in Section 6.1, to take the last partial products, a zero padding extension to the multiplicand must be performed. By doing so, the last partial product is never negated, because the 0-padding assures that is always positive, and can not be higher than A (it comes from a triplet of kind $00x$), thus the last multiplexer related to that partial product can be simplified.
Therefore, a 2to1 multiplexer has been used, as shown in Figure 12b, in order to slightly optimize area and delay of this component.
Each multiplexer receives a 2-bits selector which is coming from the `BRU` units, except the smallest one that needs only one bit of selection.
All the bits coming from the multiplexers are collected all together as the input of the following unit, which is the `XOR plane`, except bits from the smallest multiplexer that don't need to go through the `XOR plane` because the last partial product can never be negative.
In the `XOR plane`, twelve set of twenty-five XOR-ports have been instantiated. The XOR-operation has been performed between each bit of the *i-th* partial product and the most significant bit of corresponding triplet of B considered (B_{2i+1}).
By doing so, we perform the 1's complement of the partial product that must be subtracted instead of added, as explained in Section 6.2.
After that, the right partial products are reorganised in a **Dadda Tree**, as shown in layer l_5 in Figure 10, so that bits can be properly connected to half adders and full adders.
In order to obtain two addends of 48 bits each, a zero is added in the LSB of the second operand in the `dadda_tree` component.
At the end, a **carry propagate adder** is required to put together sum bit and carry bits. For this aim, a behavioral description of the adder was adopted in order to allow to Design Compiler to choose the best architecture.

One of the main drawbacks of the Booth encoding for this multiplier is that unsigned operands are turned into signed after the encoding operations. This means that we must handle the sign bit as explained before, putting additional dots in the tree representation (`S` and `1`)
Therefore **more HAs and FAs component must be placed**, increasing the area cost and the complexity of the design. However, no additional adder is needed in order to perform the 2's complement of input A in the case negative partial products, because this mechanism is handled in the structure of the tree, as explained in the document given for the lab.

6.4 Logic Synthesis

We decide to perform again the two different logic synthesis analysed in subsection 4.1.
The two different synthesis are reported in the following subsections, and then compared in the subsection 6.4.

Logic Synthesis with `compile -exact_map` and `optimize_registers` commands

The first synthesis has been performed using the commands `compile -exact_map` and `optimize_registers`.
The last command is used to performs (register) retiming on a mapped gate-level netlist. It determines the placement of registers in a design to achieve a target clock period and minimizes the number of registers while maintaining that clock period.
With the same iterative method used before, a slack equal to -0.831 ns is obtained.
The maximum allowed frequency is:

$$f_M = \frac{1}{t_{cp}} = 1.2\text{GHz}$$

In terms of **area**, with a synthesis with 0 ns clock, the circuit we obtain has a cell area equal to 6927.97 μm^2, which is divided into:

- Combinational area: 4303.35 μm^2
- Buf/Inv area: 511.78 μm^2
- Non-combinational area: 2624.62 μm^2

Logic Synthesis with `compile_ultra` command

The second synthesis has been performed using the command `compile_ultra`. This performs a two-pass high-effort compile flow on the current design for better QOR.
With the same iterative method used before, a slack equal to -1.58 ns is obtained.
The maximum allowed frequency is:

$$f_M = \frac{1}{t_{cp}} = 632.9\text{MHz}$$

In terms of **area**, with a synthesis with 0 ns clock, the circuit we obtain has a cell area equal to 5783.90 μm^2, which is divided into:

- Combinational area: 4498.86 μm^2
- Buf/Inv area: 367.08 μm^2
- Non-combinational area: 1285.05 μm^2

Comparison between the Logic Synthesis

A comparison between the two logic synthesis is shown in Table 6.

Design	f_{ck}[MHz]
`compile -exact_map + optimize_registers`	1203
`compile_ultra`	632.9

Table 6: Comparison between the Logic Synthesis

So the first synthesis allows to achieve better results in terms of performance, leading the synthesizer to adopt retiming. On the other hand, area becomes bigger, as in the previous section. Instead, using the `compile_ultra` command the increase of area is smaller than the previous case, but also the achieved frequency is lower.
Moreover, exploiting the MBE architecture and the `compile -exact_map + optimize_registers` commands we obtain better results with respect to the ones reported in Section 4, obtaining an increase of the 22% in terms of frequency.
Instead, exploiting the MBE architecture and the `compile_ultra` command, we obtain worst results with respect to the ones reported in Section 4, obtaining a decrease of the 10% in terms of frequency.
In Figure 13 a comparison between the two different total area obtained is reported.

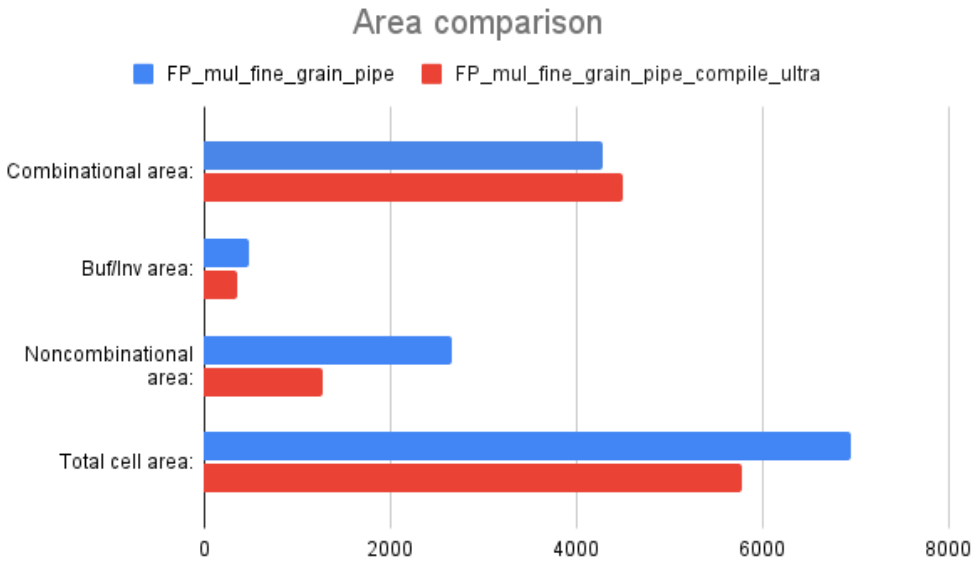


Figure 13: Total area comparison

As can be seen from Figure 13, the area of the multiplier with the `optimize_registers` commands is 17% higher respect to the one obtained in the case of the `compile_ultra` command.

7 Optimization of the MBE multiplier with Dadda tree

In the previous section, it was required to optimize the Stage 2 of the floating point multiplier by designing an unsigned multiplier which exploits a modified version of the Booth-Encoding with a Dadda tree based adder plane.
Now, since we have verified that the MBE multiplier is faster than the PPA multiplier which was implemented automatically by Synopsys in Section 4, we try to optimize the overall structure of the floating point multiplier by adding different level of pipeline, as shown in Figure 6, and then exploiting the `optimize_registers` commands.

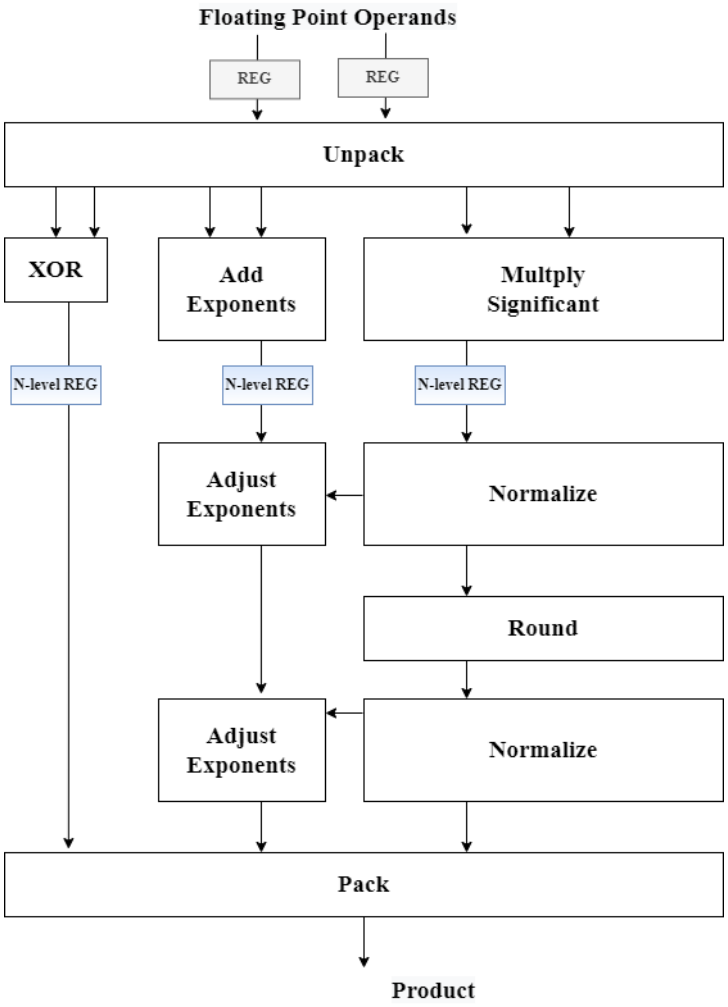


Figure 14: Basic scheme for the floating point multiplier with multiple registers at the output of the MBE

So we try to insert another pipeline register at the output of the MBE multiplier in order to achieve a lower delay. As a consequence one more clock cycle is needed to get the final result.
However, this time, the `optimize_registers` command was not able to improve the delay of the multiplier. This is probably due to the fact that Synopsys is not able to properly applied the `optimize_registers` command for component which are hierarchically organized, but we need additional commands (as done after few lines). In this case, power increases from 9.5093 mW for the 1-level pipe case up to 10.731 mW for the 2-level pipe case.
Obviosly, there is also an increase in terms of area, of almost 10%.

By using the `ungroup component_instance -flatten` command, we are able to unflatten the specific component (in this case, the MBE), making the `optimize_registers` command able to improve the overall critical path of the system. By doing so, we obtain the result shown in Table 7.

Architecture	**Design**	f_{ck}[MHz]
1 level of pipe	`ungroup MBE_instance -flatten` + `compile -exact_map` + `optimize_registers`	1205
1 level of pipe	`ungroup -all -flatten` + `compile -exact_map` + `optimize_registers`	1220
2 levels of pipe	`ungroup MBE_instance -flatten` + `compile -exact_map` + `optimize_registers`	1369
2 levels of pipe	`ungroup -all -flatten` + `compile -exact_map` + `optimize_registers`	1449

Table 7: Comparison between the Logic Synthesis

Adding two pipeline levels to the MBE multiplier an improvement in terms of delay is achieved, as expected, but on the other and also latency increases, so for this reason no further pipeline registers have been considered.

8 Final Results

In this section a comparison between analyzed architectures is reported. At the beginning a behavioural description of significant multiplier was adopted and we have seen that a better performance in terms of clock frequency is achieved imposing a Parallel Prefix Architecture in the multiplier. Then to achieve better performances, retiming and optimizations provided by `compile_ultra` were exploited. At the end, a Modified Booth Encoding Multiplier was implemented. All these architectures have been compared in terms of **delay**, **power** and **area**.
A comparison between the different frequency achieved with the structures implemented with the different logic synthesis is shown in Table 8.

Architecture Multiplier	**Design**	f_{ck}[MHz]
PPA	`compile -exact_map` + `optimize_registers`	1140
PPA	`compile_ultra`	665.78
PPA - 2 levels of pipe	`compile -exact_map` + `optimize_registers`	1204
PPA - 3 levels of pipe	`compile -exact_map` + `optimize_registers`	1470
MBE	`compile -exact_map` + `optimize_registers`	1203
MBE	`compile_ultra`	632.9
MBE - 1 level of pipe	`ungroup -all -flatten` + `compile -exact_map` + `optimize_registers`	1220
MBE - 2 levels of pipe	`ungroup -all -flatten` + `compile -exact_map` + `optimize_registers`	1449

Table 8: Comparison between the different frequencies obtained

A comparison between the different areas achieved with the structures implemented with the different logic synthesis is shown in Table 9.

Architecture Multiplier	**Design**	**Area** $[\mu m]^2$
PPA	`compile -exact_map` + `optimize_registers`	5888
PPA	`compile_ultra`	4371
PPA - 2 Level of pipe	`compile -exact_map` + `optimize_registers`	6909
PPA - 3 Level of pipe	`compile -exact_map` + `optimize_registers`	7252
MBE	`compile -exact_map` + `optimize_registers`	6927
MBE	`compile_ultra`	5384
MBE - 1 level of pipe	`ungroup -all -flatten` + `compile -exact_map` + `optimize_registers`	7297
MBE - 2 levels of pipe	`ungroup -all -flatten` + `compile -exact_map` + `optimize_registers`	8045

Table 9: Comparison between the different areas obtained

Overall, the synthesis with `compile -exact_map` + `optimize_registers` commands generally leads to the best results in term of possible frequency achievable. This was quite predictable, since the `optimize_registers` command is conceived to improve the circuit in terms of delays.
Instead, the synthesis with `compile_ultra` command leads to the best results for what concern the minimum area occupied.

From the point of view of the **area**, MBE is always worse than PPA, with an area $\sim 20\%$ larger on average.
About **delay**, MBE seems to be better, even though the only comparable results are the ones of the versions without additional pipe registers, because for the pipelined versions of the MBE we used the `ungroup -all -flatten` command, which let Synopsys do a better optimization. A proper comparison could be done redoing all the synthesis also for the PPA versions, but we think it is not so relevant.
Therefore, comparing the versions without additional pipe stages, MBE version is $\sim 5.5\%$ faster.

By evaluating the execution time, which is defines as:

$$\texttt{execution time} = \texttt{latency} \times \texttt{clock period}$$

we can state that the architecture exploiting the MBE is in conclusion the best trade off among the analyzed alternatives.