Politecnico di Torino

III Facoltà di Ingegneria

Exercises and Homeworks for the course Integrated Systems Architecture

Master degree in Electrical Engineering

Authors: Group 20

Alessandro Lovesio, Lorenzo Poloni, Simone Pont

December 9, 2020

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

Contents

Link to our group repository:
`https://github.com/AleLovesio/Laboratory_ISA_Group20.20`
Direct link to the folder of Lab 2 in the same repository:
`https://github.com/AleLovesio/Laboratory_ISA_Group20.20/tree/main/Lab2`

0.1 Introduction

The purpose of this laboratory is to deal with digital arithmetic issues. In particular this laboratory focuses on three different points. The first is using the *Synopsys Design Vision* tool to optimize a Floating Point Multiplier by changing the architecture of the internal unsigned integer multiplier with different architectures by taking advantage of the *Design Ware* library. The second is about optimizing the FP multiplier by employing retiming and advanced *Synopsys* functions. Finally, the third is about designing an Unsigned integer multiplier, based on a Dadda Tree partial product addition with Booth Encoding, to be used for the Floating Point multiplier.
First we downloaded from the portale the "Floating Point Adder and Multiplier" (fpuvhdl) project, which is a package containing already developed Floating Point Addition and Multiplication circuits. We have then unpacked the project and we have taken only the necessary files for the multiplier, which consist of the main multiplier file, the multiplier stages, the rounding file, the normalization file and the pack/unpack files. This multiplier is implemented quite structurally, even though the circuit was implemented describing the general flow of the data and the arithmetic operations (additions, multiplication, incrementers...) have been implemented behaviorally. As a side note, the multiplier has been implemented as a 32x32 bit multiplication, while only a 24x24 bit multiplication would be sufficient. We think this was done so that *Synopsys* can pick the multipliers from the Design Ware library as 24x24 bit multiplier may not be available.
First, we developed a test bench based on the first laboratory files. It is composed of a main test bench that includes the Data maker, the Data sink, a clock generator and the device under test, the floating point multiplier. The clock generator now only generates the clock as the reset is no longer needed, while the Data sink has been made more generic so that by changing a few variables we can adapt it to different numbers of pipeline stages as the latency changes with their number. Then we tested the provided version of the multiplier with the test bench by taking advantage of the hex files to check the functionality of the DUT. To have a more complete check we used also a set of 100 random inputs, in addition to the provided ones, and the results' correctness was verified.
One last remark before getting into the real topic of the laboratory: we added a set of registers to the inputs of the multiplier so that the whole circuit is interfaced by registers both on output and on input. This can be useful when calculating the critical paths. This updated version has been tested with the test bench by increasing the variable containing the number of pipeline stages and it was demonstrated that it still works. This version will be the benchmark for all the improvements that will be described in the following sections.

0.2 Standard Implementation

In this laboratory activity attention is focused on comparison of different multiplier architectures in terms of timing and area. The DUT is a pipelined floating-point multiplier which basically splits the sign, the exponent and the significand of a number and treats them as integer values. As a consequence, a lot of integer adders and multipliers can be found in the design. In particular, we want to extract the performance parameters (i.e. area and timing) when using different types of multipliers at Stage 2 of the pipeline that is the point at which the significands are multiplied. The *Synopsys* tool comes to this problem's aid as it allows to implement different architectures available in the *DesignWare Library*. The basic architecture was firstly simulated by means of *Modelsim* before synthesis. It has been observed that the circuit works correctly. After this step, *Synopsys* was used to get the needed parameters. As in the previous laboratory, a script was written to automatize the synthesis process. Again, clock uncertainty, input and output delays and a load were added in order to allow for more realistic estimates. Before compiling, the *ungroup* command was run to flatten all the hierarchy for all cells recursively. As for the basic architecture the compiling was performed right after the previous command. For the other two instead the *set_implementation* command was added to specify the desired circuit. The architectures used for the sake of comparison are: the one based on the Carry Select Adder and the one based on a Parallel Prefix (PP) implementation. It is expected that the PP architecture would outperform the CSA one in terms of timing. To know why, some words on the two implementations must be spent. The principle behind the CSA is to compute the sum bits of the operands and then, only at the end, the carry, properly weighed, is added to the final result. Thus, the major delay resulting from carry propagation is handled only at the end using a two-operand adder, while the previous sums are computed in parallel across different levels. In practice, the CSA is a full adder employed as a 3-2 compressor. Hence, this kind of circuit is perfect for multi-operand additions. The CSA architecture available in the *DesignWare Library* is an array multiplier, while PPARCH is a delay-optimized Wallace Tree which also employs Booth Encoding. Moreover, in the last stage the two numbers are added using a Parallel Prefix Adder. The PPA is a two-operand adder. All parallel prefix structures are based on the CLA and work on the assumption that for associative operators, the operands can be computed in parallel. Given the inputs, a pre-calculation of all generate and propagate signals ensues. Thanks to these signals, the carry for each stage is computed using the well-known recursive equation. This step can be carried out in parallel by means of blocks that apply a special associative operator to G and P signals. After this, the final sum is computed. These adders are known to be very fast as the delay increases logarithmically with the number of bits. Benefiting from the tree structure, the booth encoder and a PPA, it is clear that the PPARCH implementation is significantly faster, as shown in the following table. The tree structure typical of the Wallace Tree also reduces the total area as fewer CSA are required. However, as the Wallace Tree structure is highly irregular, its area and delay might increase after the physical design because of an inefficient layout.

	Basic	CSA	PPA
Timing	$1.6ns$	$4.49ns$	$1.6ns$
Max. frequency	$625MHz$	$222.72MHz$	$625MHz$
Area	$3999\mu m^2$	$4806\mu m^2$	$3987\mu m^2$

Minimum clock period was calculating setting a clock constraint of 1 ns on the synthesizer and then increasing the period until +0.00 slack is reached.

0.3 Fine-Grain Pipelining And Optimization

The following sections describes the process of making two different kinds of optimizations. The first is about employing the *Synopsys Design Vision* tool to perform more "detailed" optimizations that are not just about implementing a different architecture of a block of the circuit, the unsigned multiplier in particular. The second part is about designing a new unsigned multiplier based on a Dadda Tree with booth encoding to further improve the design.

0.3.1 Fine-Grain Pipelining And Optimization with Synopsys

As a first step of this advanced optimization, a manually commanded fine-grain pipelining is performed by adding a new layer of registers and letting *Synopsys Design Vision* retime them into the unsigned multiplier, which is the component that drives the critical path of the Floating Point multiplier. As mentioned, a set of registers has been described in VHDL in the end of the stage 2. In the VHDL description, this set of registers is directly followed by the original set of registers that separates the second and the third stage. The multiplier has been left behaviorally implemented with the "*" operator.
This new design has been verified with *Modelsim* by performing a simulation with the usual values. The only difference in respect to the previous ones, is the increment of the pipeline stages variable in the Test Bench Data Sink file so that the reading timing is still correct.
Having verified that the design has been synthesized with *Synopsys Design Vision* with, in addition the the standard commands used previously (*set_implementation* command not included), the new command *optimize_registers*. This command performs a retiming of the registers in order to optimize the critical paths and to improve the maximum working frequency. Some area changes are expected compared to the original versions as a new set of registers has been added.
After performing a few synthesis cycles to get to the point where the slack is zero (or almost zero), the obtained maximum frequency is $667MHz$ while the occupied area has been found out to be equal to $4070\mu m^2$.
For the second step of advanced optimizations, a command the performs a much wider set of optimizations, in addition to the fine-grain pipelining via retiming, has been used. So, instead of invoking the *optimize_registers* command followed by the *compile* command, the *compile_ultra* command has been used. This command is used particularly for strict timing constrained circuits, where the timing must be optimized as much as possible to reach the best performance. The *compile_ultra* command, in addition to the normal *compile* routines, has the following features:

- Additional high-effort delay optimization algorithms
- Advanced arithmetic optimization
- Integrated datapath partitioning and synthesis capabilities
- Finite state machine (FSM) optimization
- Advanced critical path resynthesis
- Register retiming
- Support for advanced cell modeling
- Advanced timing analysis

While not every feature is useful for our design, we can be sure that *Synopsys Design Vision* will perform every optimization it can apply to our circuit. Like before, the synthesis has been performed a few times until the slack time was minimized to zero. At that point the obtained maximum frequency

is $690MHz$ while the occupied area has been found out to be equal to $4303\mu m^2$.
We can compare the two optimizations (*optimize_registers + compile* vs *compile_ultra*) by looking at the following table:

	compile	*optimize_registers* + *compile*	*compile_ultra*
Critical path time	$1.6ns$	$1.5ns$	$1.45ns$
Max. frequency	$625MHz$	$667MHz$	$690MHz$
Area	$3999\mu m^2$	$4070\mu m^2$	$4303\mu m^2$

As we can see every iteration is better than the previous one, and, in particular, the best results are obtained with the *compile_ultra* command. By employing the *optimize_registers + compile* commands, the critical path has improved so that the maximum frequency has increased by $42MHz$ (+6.7%) while the Area has increased by $71\mu m^2$(+1.8%). The circuit generated with the *compile_ultra* command has seen an improvement in maximum frequency of $65MHz$ (+10.4%) and an Area increase of $304\mu m^2$ (+7.6%) when compared to the initial version, which is a much larger increase. Comparing the two optimized versions, the improvement in maximum frequency from the first to second one is of $23MHz$ (+3.5%) while the Area increase is of $233\mu m^2$ (+5.7%).

0.3.2 Optimization with an Advanced MBE Multiplier

In this last part of the laboratory we had to design an unsigned Modified Booth Encoded (MBE) multiplier to obtain the significands' multiplication. Doing this we can generate all the partial products without the need of adders/subtractors and after that add them together using a Dadda-tree structure. We have to precise that even if in the previous behavioral description the multiplication is done working on 32 bits, in this case we can reduce the parallelism of the multiplier to the minimum needed that is equal to 24 bits. This lets us to minimize the complexity and the dimension of our final circuit while maintaining its correct behavior. To do this we need also some little modifications of the *fpmul_stage2_struct.vhd* file that allows us to correctly add the 24 bit version of multiplier.
We can start analyzing the modified booth encoding that is used to generate all the partial products (PPs). This technique is an extension of the standard Radix-2 approach where more bits are analyzed simultaneously to obtain a better reduction of PPs. MBE can be seen as a Radix-4 approach because it produces half PP compared to the Radix-2 one. This is achieved by dividing the b operand in different groups of 3 bits with a 1-bit overlap, considering one $'0'$ before the LSB and the needed number of zeros after the MSB to complete the final grouping. The grouping needed in our case is reported in the following figure 1:

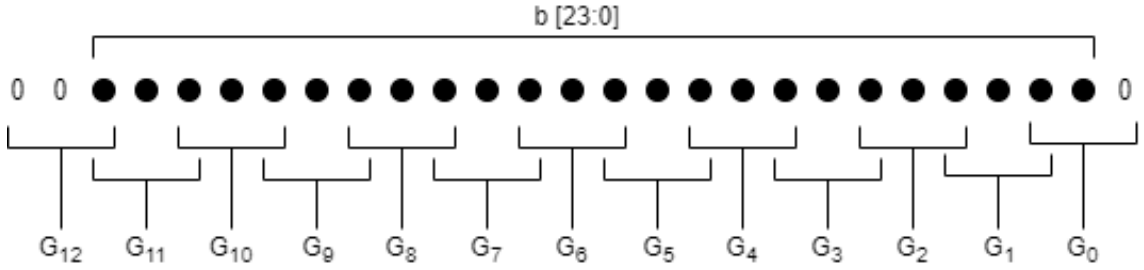

Figure 1: Grouping of the bits of operand b to create the partial products

From every group we can derive a different partial product that can assume one of the following

values, depending on the values of grouped bits:

Group of b bits (b_{2i-1},b_{2i},b_{2i+1})	Value of partial product p_i
000	0
001	$+a$
010	$+a$
011	$+2a$
100	$-2a$
101	$-a$
110	$-a$
111	0

We can summarize this table using one general equation to calculate all the PPs that is based on the CA2 notation:

$$p_i = (q_i \oplus b_{2i+1}) + b_{2i+1} \qquad \textit{where} \qquad q_i = \begin{cases} 0 & \textbf{if} \quad \overline{(b_{2i} \oplus b_{2i-1})} \cdot \overline{(b_{2i} \oplus b_{2i+1})} \\ a & \textbf{if} \quad (b_{2i} \oplus b_{2i-1}) \\ 2a & \textbf{if} \quad \overline{(b_{2i} \oplus b_{2i-1})} \cdot (b_{2i} \oplus b_{2i+1}) \end{cases} \tag{1}$$

We have to notice that the final + is not a logic OR but a sum that is needed to obtain the correct result using the CA2 notation. This equation, indeed, is based on the sign inversion of q_i (that is always generated positive) when necessary, in particular when the 3rd bit of the group (b_{2i+1}) is equal to $'1'$. One example of the calculation of the first partial product is reported in figure 2:

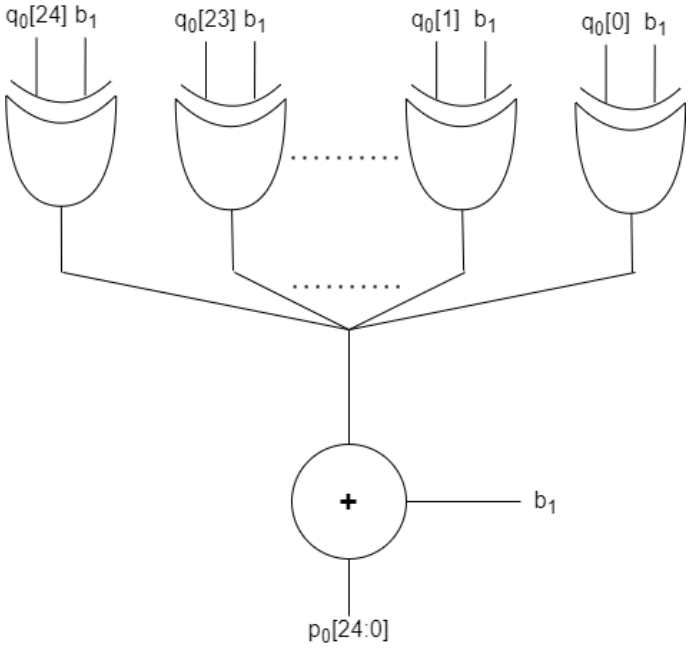

Figure 2: Generation of the first partial product

If we repeat this calculation for all the groups and we use a dot notation to represent the results of XOR operations we can obtain the following figure 3 where all the PPs are present:

Figure 3: All the partial products after the Booth encoding

In this picture the S_i terms refers to the 3rd bit of every group and in particular to the bits b_{2i+1}. To have a correct addition of all these PPs we need to apply a sign extension. To minimize the number of resources needed in the following step of addition we can optimize this process. Considering again only the first PP we can see the two cases of possible sign extension:

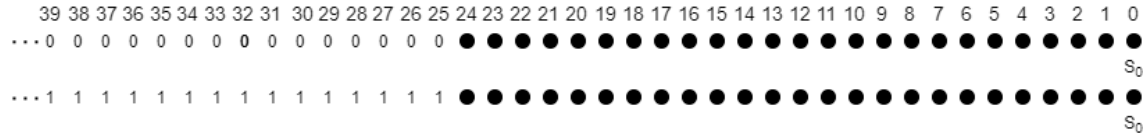

Figure 4: Two possible cases of sign extension

We can notice that by adding $'1'$ at the position after the MSB of the original PP (2^{25} in this case) we can easily pass from the case of negative extension to the one of positive extension. So we can extend every PPs with $'1'$ (like a negative sign extension) and after that add one in correct position for the PPs that has to be positive. Knowing that the positive PPs are the ones where $S_i = b_{2i+1} = 0$ we can always add the value $\bar{S}_i$ with the correct weight. The obtained result is reported in figure 5:

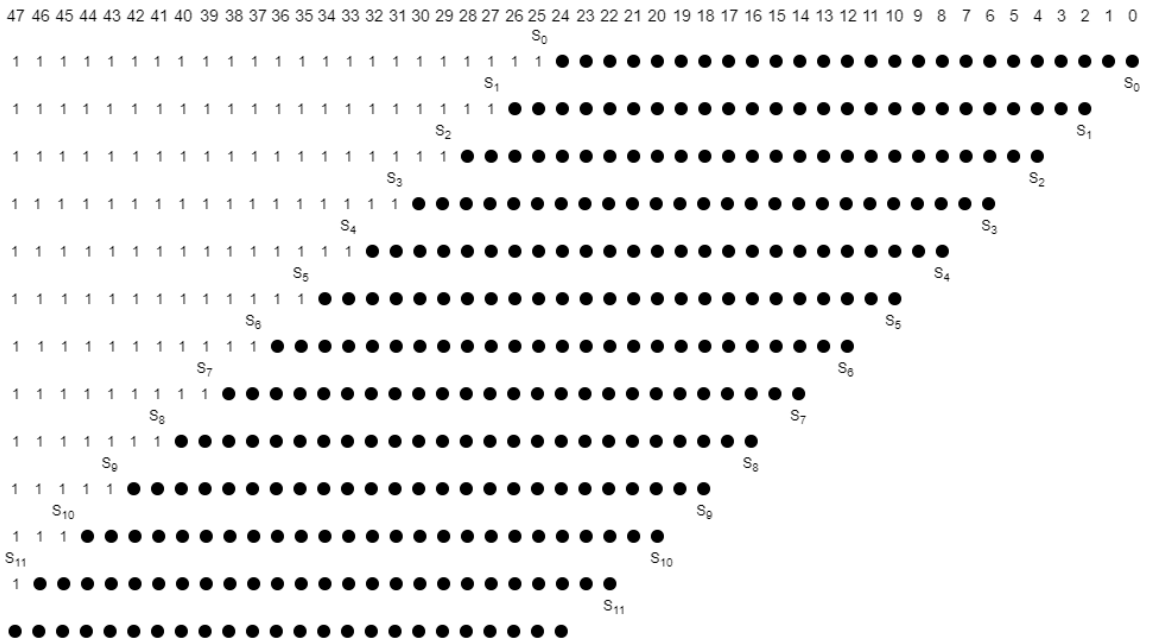

Figure 5: Generic sign extension of all partial products

This structure works properly but can be optimized by summing in advance the sign extension bits and reducing the total height with some simple combinations. The final version, where all the PPs are ready to be summed using the Dadda tree, can be seen in figure 6:

Figure 6: Partial products after all the reductions ready to be summed

We can now start describing the steps needed to obtain the correct structure of the Dadda tree used to add all the previous obtained PPs. Starting from the theory and considering that we start with 13 PPs we have calculated the number of levels for the Dadda tree approach, in particular we have found that we need 6 levels to obtain the complete reduction. The height d_i of each level i has

been also calculated using the following equation:

$$d_{i+1} = \left\lfloor \frac{3}{2} d_i \right\rfloor \tag{2}$$

The last step to obtain the final structure is to correctly reduce the height at every level. To do this we can use half and full adders to combine bits with the same weight but we have to take into account also the generated sum and carry bits. Considering that the Dadda tree can be seen as an ALAP approach we always have to instantiate the minimum number of half and full adders needed to pass from one level to the next one. In the following pictures the procedure to obtain the needed reductions is reported in a graphical way where we can see also the instantiated adders:

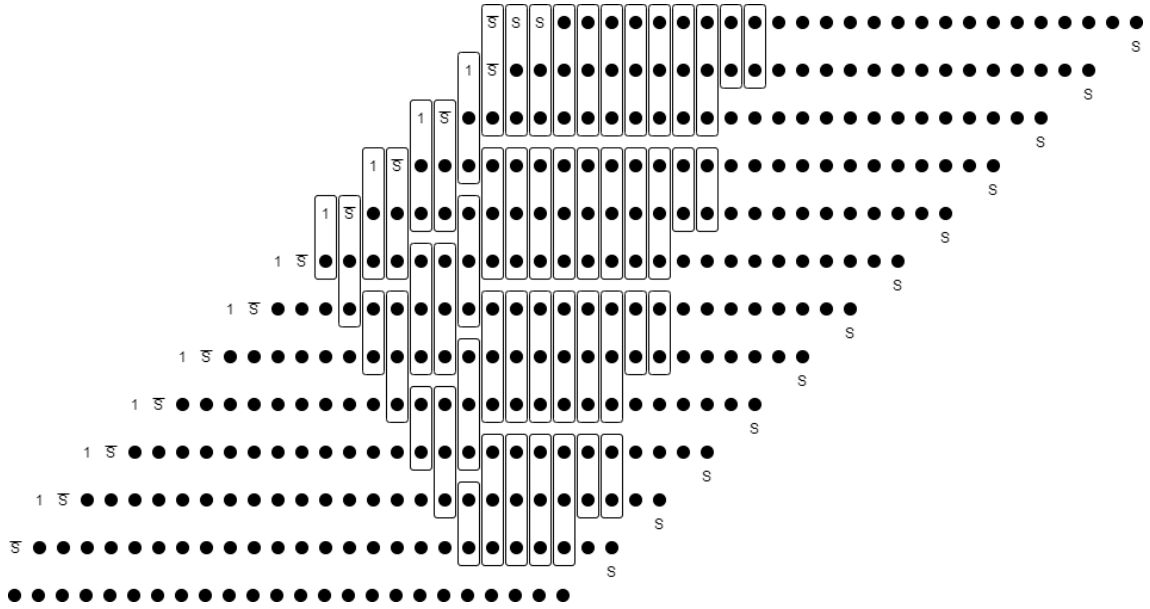

Figure 7: 6th level of Dadda tree and the HA/FA needed to pass at level 5

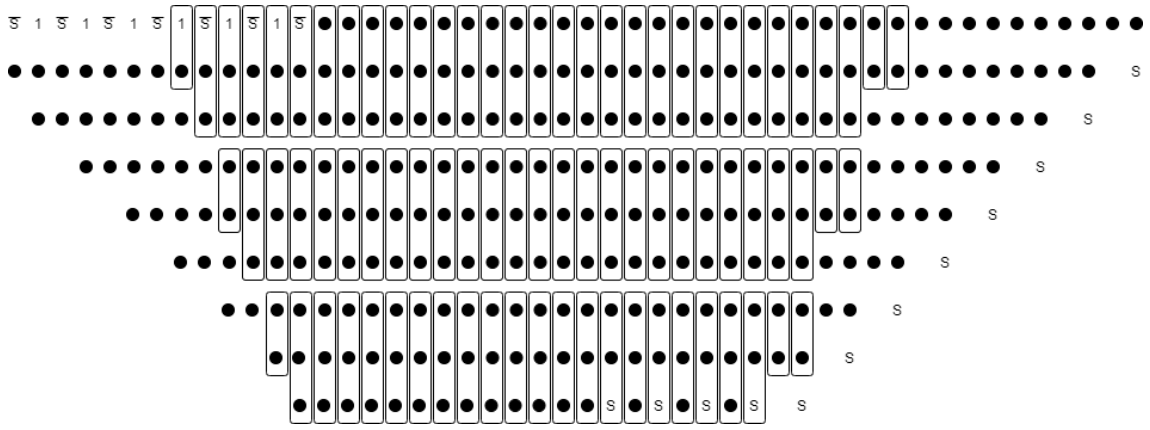

Figure 8: 5th level of Dadda tree and the HA/FA needed to pass at level 4

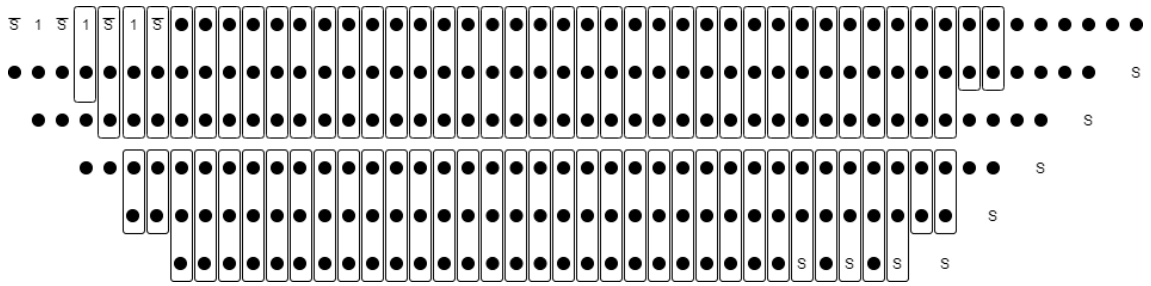

Figure 9: 4th level of Dadda tree and the HA/FA needed to pass at level 3

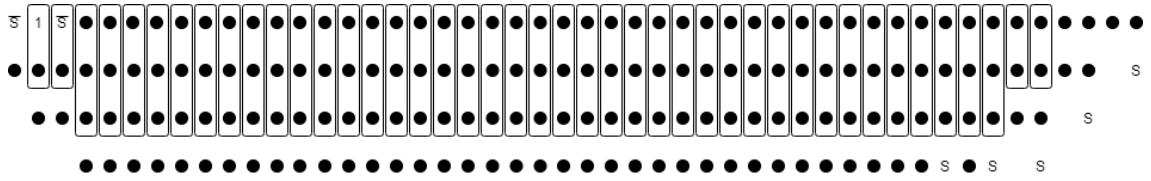

Figure 10: 3rd level of Dadda tree and the HA/FA needed to pass at level 2

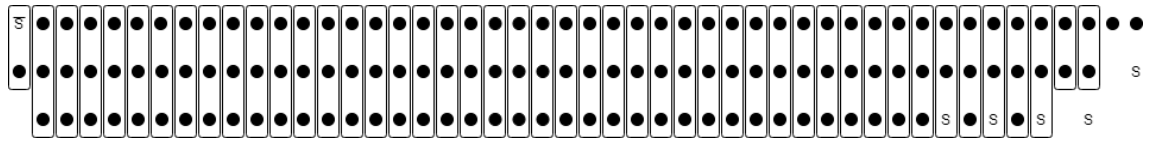

Figure 11: 2nd level of Dadda tree and the HA/FA needed to pass to the last level

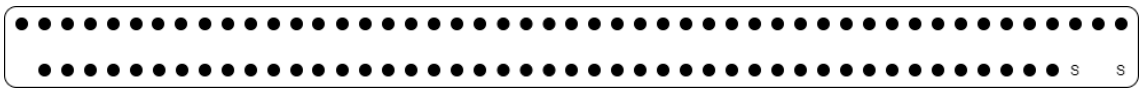

Figure 12: Last level of the Dadda tree where we have obtained 2 final results

We can see that in the last level we have obtained only 2 rows that can be summed using a standard adder working on 48 bits, that in our case is implemented behaviorally. In the following table a summary of the previous procedure and of the needed resources is reported:

Level	Height (d)	Number of HA	Number of FA
6	13	12	40
5	9	9	72
4	6	7	67
3	4	4	39
2	3	3	43
1	2	0*	0*

** in this case we implemented the addition behaviorally and so we don't know how many FA and HA are used if any.*

Using this multiplier for the significand we simulated again the complete architecture of the floating point operator and we checked its correct behavior. After that we have synthesized again using the

compile_ultra command to find the maximum frequency and the corresponding area occupation. Using the same procedure described before we have obtained a null slack at the frequency of $662MHz$ and in this situation we have an occupied area of $5087\mu m^2$. The following table helps us to compare the results obtained with this last procedure and the ones obtained using the behavioral multiplier (in both cases using the *compile_ultra* command):

	Behavioral multiplier	MBE multiplier
Critical path time	$1.45ns$	$1.51ns$
Max. frequency	$690MHz$	$662MHz$
Area	$4303\mu m^2$	$5087\mu m^2$

From the results we can notice that the best solution is the one with the behavioral multiplier for both timing and area occupation. The bigger difference is on the area occupation that in the MBE solution is greatly increased (about +18%). This is due to the fact that with the multiplier described in a structural way the synthesizer cannot apply many improvements or use an optimized multiplier from the library. For the same reason we have also a lower increase in critical path delay (about +4%) but in this case the final result is good and comparable with the one obtained using the *optimize_registers + compile* commands.

0.4 Conclusion

All the obtained results are reported together in the following table:

	basic	*CSA*	*PPARCH*	*optimize_registers*	*compile_ultra*	*MBE multiplier*
Critical path time	$1.6ns$	$4.49ns$	$1.6ns$	$1.5ns$	$1.45ns$	$1.51ns$
Max. frequency	$625MHz$	$222.72MHz$	$625MHz$	$667MHz$	$690MHz$	$662MHz$
Area	$3999\mu m^2$	$4806\mu m^2$	$3987\mu m^2$	$4070\mu m^2$	$4303\mu m^2$	$5087\mu m^2$

The best overall results are obtained with the *compile_ultra* command that performs all the possible optimizations. Considering the area occupation we have that the best solution is the one using the PPARCH multiplier, this is also due to the fact that in that case some intermediate register are missing.