Politecnico di Torino

III Facoltà di Ingegneria

Integrated systems architecture

Laboratory 2

Laurea Magistrale in Ingegneria Elettronica

Orientamento: Sistemi Elettronici

Group n. 9

Authors:

Favero Simone S270686
Micelli Federico S270456
Spanna Francesca S278040

Repository Github: github.com/simomaiden/ISA_2020_Group_09_Lab_2

Index

Digital arithmetic and logic synthesis

1.1 Verification of the pipelined multiplier

The first step consists of designing a testbench to verify the correct behavior of a given multiplier implementation.
From the hardware description code, it is possible to understand its structure, which is composed by four different pipelined stages, as shown in the figure below.

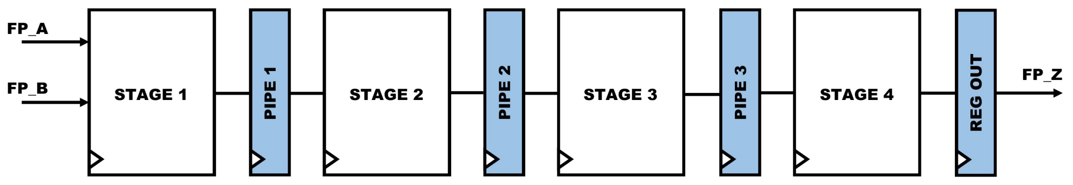

Figure 1.1: Pipelined multiplier scheme

A hierarchical mixed verilog-VHDL testbench has been developed. It relies on an input samples file, which contains ten 32-bits samples expressed in hexadecimal notation. The aim of the testbench is to evaluate and check the squares of the input data.
For the sake of simplicity, output data are converted back to hexadecimal notation.

The defined testbench contains the following entities:

- **Clk_rst_gen**: its purpose is to generate a clock signal with a given period, after having emulated the initial reset condition.
- **Read_data**: it reads the input samples from *fp_samples.hex* and provides them to the device under test, with a periodicity equal to the clock cycle. It is also employed to manage the signals RESULT_AVAILABLE and END_SIM; the former is used to write the results with the correct timing, while the purpose of the latter is to stop the simulation when the end of input samples file is reached.
- **FPmul**: it is the device under test that represents a pipelined multiplier.
- **Write_results**: this module is used to write on an external file, *results_vhdl.txt*, the output results of the multiplier. The writing operation is performed when the signal RESULT_AVAILABLE is asserted: at the beginning of the simulation it is necessary to wait the correct amount of latency before writing the first result.

- **Error_check**: its functionality is to compare, one by one, the results provided by the multiplier with respect to the ones reported in the file *fp_prod.hex*. A feedback of the comparation is reported in the file *error_check.txt*. In case of error, both expected and computed values are provided.

The testbench structure is implemented according to the following scheme.

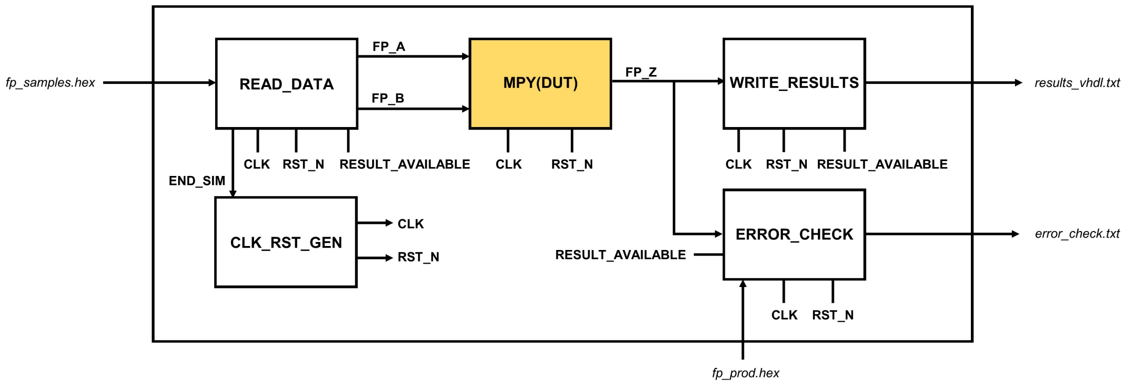

Figure 1.2: Testbench block diagram

A simulation launched on ModelSim generated results reported in the following table.

Row	***fp_prod.hex***	***results_vhdl.txt***
0	000000000	00000000
1	3dc3910d	3dc3910d
2	0bc80005	0bc80005
3	3f278ddf	3f278ddf
4	0b100005	0b100005
5	3f800000	3f800000
6	0c440004	0c440004
7	3f278ddf	3f278ddf
8	0da20000	0da20000
9	3dc3910d	3dc3910d

No errors have been reported in the file *error_check.txt*, therefore the multiplier model works correctly.

1.2 Input register implementation

As visibile in the figure 1.1, a layer of registers is not present at the input of the first stage. However, it is strongly suggested to implement an interfacing barrier of registers in order to separate the designed system from the external environment.

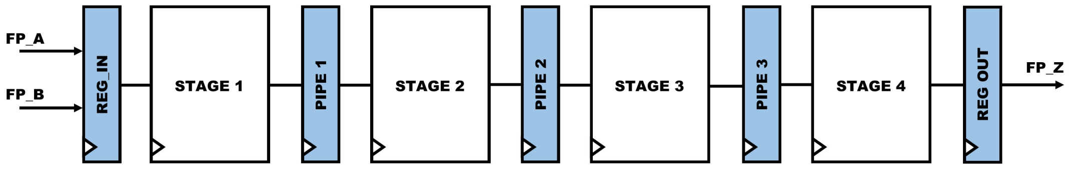

Figure 1.3: Pipelined multiplier with additional input register scheme

In order to implement that, the file *fpmul_pipeline.vhd* has been modified: a process has been declared to provide input samples with the correct timing.

```
PROCESS(clk)
BEGIN
    IF RISING_EDGE(clk) THEN
      FP_A_in1 <= FP_A;
          FP_B_in1 <= FP_B;
    END IF;
END PROCESS;
```

Due to this modification, the overall latency has been increased by one clock cycle, therefore the availability of the results is postponed by the same delay.
As in the previous section, the simulation did not report any error during the execution.

1.3 Synthesis and implementaton of the multiplier

Synopsys design compiler can be exploited in order to synthesize the architecture of the pipelined multiplier on a given cell library.
After having analyzed and elaborated the VHDL files, a clock signal with a period equal to 0 ns is declared in order to let the CAD optimize the design and provide an estimation of the maximum achievable clock frequency as a negative slack.

It is important to highlight that the provided slack value is only to be considered as a starting point to test the design compiler in order to find the maximum achievable frequency. As a consequence, an iterative process has to be performed until a null slack is shown in the timing report.
As requested, before starting the compilation, the hierarchy has been flatten and the following final results have been obtained.

Basic architecture		
$T_{ck,\ min}$ [ns]	**$F_{ck,\ max}$ [MHz]**	**Area [um^2]**
1.55	645.2	4059

After having flatten the hierarchy, it is possible to specify to the design compiler the desired implementation for the entities involved in the architecture. In particular, the significand multiplier defined in the stage 2 has been forced to be implemented using a CSA based architecture or a parallel prefix based one.
This choice has consequences on the maximum clock frequency and the area occupation, as shown in the tables below.

CSA based architecture		
$T_{ck,\,min}$ [ns]	$F_{ck,\,max}$ [MHz]	Area [um^2]
4.48	223.2	4806

Parallel prefix based architecture		
$T_{ck,\,min}$ [ns]	$F_{ck,\,max}$ [MHz]	Area [um^2]
1.56	641	4097

It is possible to notice a strong similarity between all parameters of the basic and PP architectures. Therefore, due to the timing constraints, it is supposed that the design compiler has chosen the fastest implementation for the significand multiplier, which coincides with the parallel prefix.
As a matter of fact, the CSA based architecture is characterized by a slower clock frequency, with a decrement factor equal to 2.8.

A further consideration is related to the area of the circuit, which is larger in the case of CSA based architecture, implying more complexity and a higher cost.

1.4 Fine-grain pipelining

In this section it is requested to introduce an additional layer of registers inside the floating point multiplier, implementing a fine-grain pipeline.

In particular, according to the specifications, a first register needs to be introduced at the output of the significand multiplier in the stage 2. However, it is important to ensure the correct timing during the data flow. For this reason all signals that are involved in the connection between stages 2 and 3 must be delayed by the same amount.

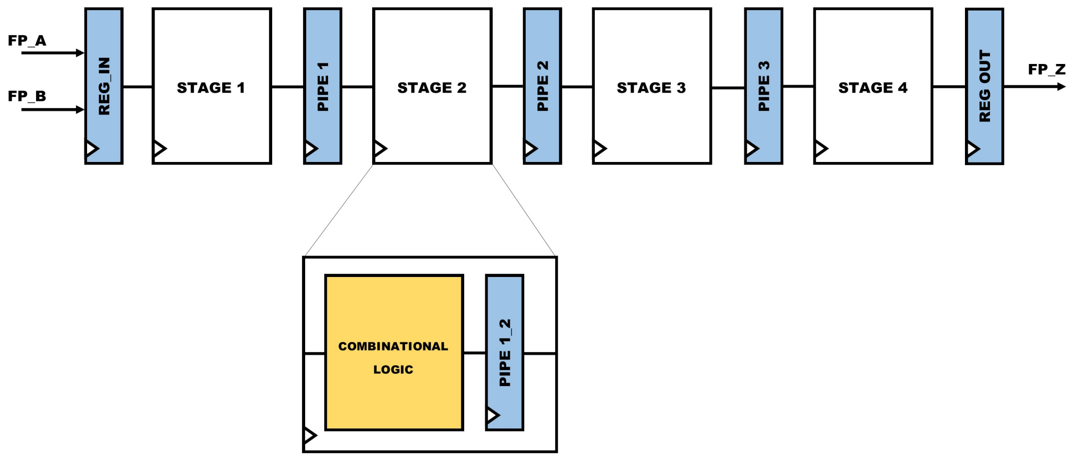

Figure 1.4: Fine- grain pipelined multiplier

Therefore, *fpmul_stage2_struct.vhd* has been modified: four additional signals and the following process have been declared.

```
PROCESS(clk)
        BEGIN
                IF RISING_EDGE(clk) THEN
                        SIG_in_int <= SIG_in_int_copy;
                        EXP_in_int <= EXP_in_int_copy;
                        EXP_pos_int <= EXP_pos_int_copy;
                        EXP_neg_int <= EXP_neg_int_copy;
                END IF;
END PROCESS;
```

Before proceeding with the next steps, the correct behavior of the fine-grain pipelined multiplier has been verified with a proper testbench, that takes into account the additional introduced latency.

When an architecture that contains registers is considered, the Synopsys design compiler is able to optimize registers position, applying the universal retiming technique. This method allows to potentially reduce the critical path, maintaining the timing of the original system unchanged.

This operation can be executed after the compilation procedure, using the command *optimize_registers*.
Similar and additional optimizations can be obtained exploiting the ultra mode of the design compiler.

compile + optimize_registers		
$T_{ck,\ min}$ [ns]	**$F_{ck,\ max}$ [MHz]**	**Area [um^2]**
0.80	1250	4897

compile ultra		
$T_{ck,\ min}$ [ns]	**$F_{ck,\ max}$ [MHz]**	**Area [um^2]**
1.50	667	4479

As noticeable in the tables above, the *optimize_registers* command, issued after a basic compilation, leads to higher performances in terms of frequency. However, also a higher value of the area is obtained with this method.

It is noticeable how a direct request of an optimization method, as for the case of *optimize_registers* command, is able to lead to a better performance, differently from the other case. Indeed, when the command *compile_ultra* is issued, several optimizations are provided automatically by the design compiler.

As a matter of fact, it is important to understand which is the best optimization method to be implemented from case to case, and force the design compiler to apply it to the architecture under analysis.

MBE multiplier design

The purpose of this chapter is to replace the behavioral significand multiplier, defined in the stage 2 of the floating point multiplier, with a designed MBE fully combinational multiplier that operates with unsigned data.

The design has been developed according to the following specifications:

- **Radix-4 approach**: useful to reduce the adder tree depth.
- **Modified Booth Encoding**: used to simplify the expression of partial products.
- **Dadda-tree method**: it is an ALAP approach for the allocation of half adders and full adders in the compression tree.
- **Stanford sign extension**: it allows to significantly reduce the overall amount of operators needed in the compression tree.

It is important to notice that, since this multiplier will be employed in the provided floating point structure, a 24-bits parallel architecture would be enough to evaluate the significand multiplication. However, it has been choosen to maintain the extension on 32 bits already defined in the initial behavioral description.

The general structure of the implemented operator is reported in the following scheme.

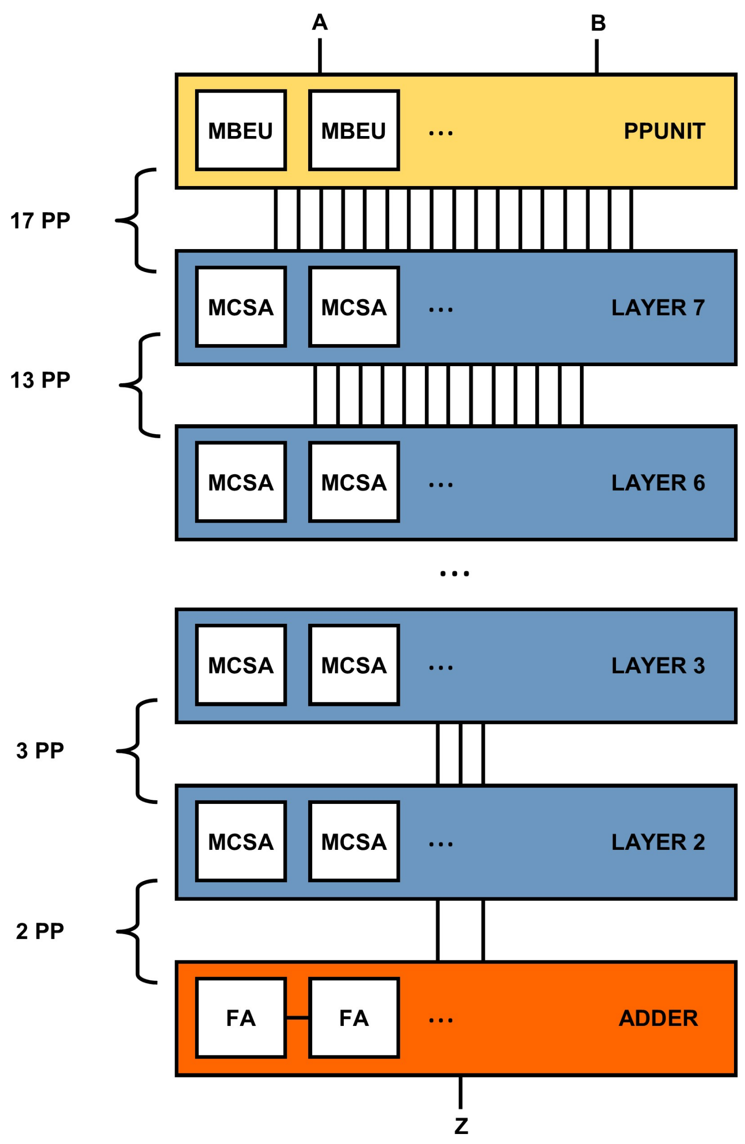

Figure 2.1: MBE multiplier general scheme

The whole architecture has been developed following a hierarchical approach, as described in the following sections.

2.1 Partial products unit

The yellow block, defined as PPUNIT in the figure 2.1, is meant to generate the seventeen initial partial products, according to the Booth encoding approach.
Each one of them is obtained by means of a MBEU entity, represented in the following figure.

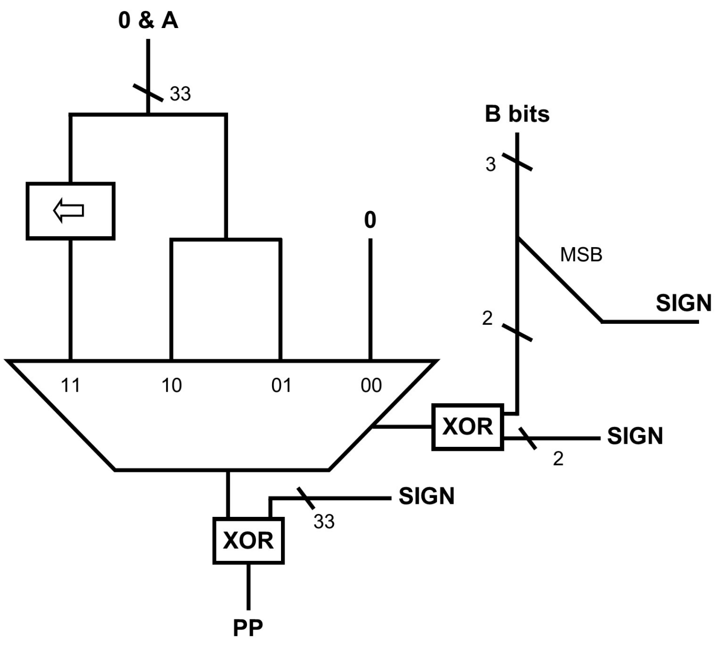

Figure 2.2: MBEU scheme

This unit receives as inputs the multiplicand A and a sequence of three bits from the multiplier B. It is meant to provide as output the correct encoded version of A, taking into account the possible complementation due to the sign bit.

The selection performed by the multiplexer has been obtained from the reported table:

b_{2j+1} b_{2j} b_{2j-1}	P_j
000	0
001	A
010	A
011	2A
100	-2A
101	-A
110	-A
111	0

Moreover, the sign bit is an additional output of the structure, since it is needed for the application of the Stanford sign extension, as reported in the figure 2.3. The extended partial products represent the output of the PPUNIT.

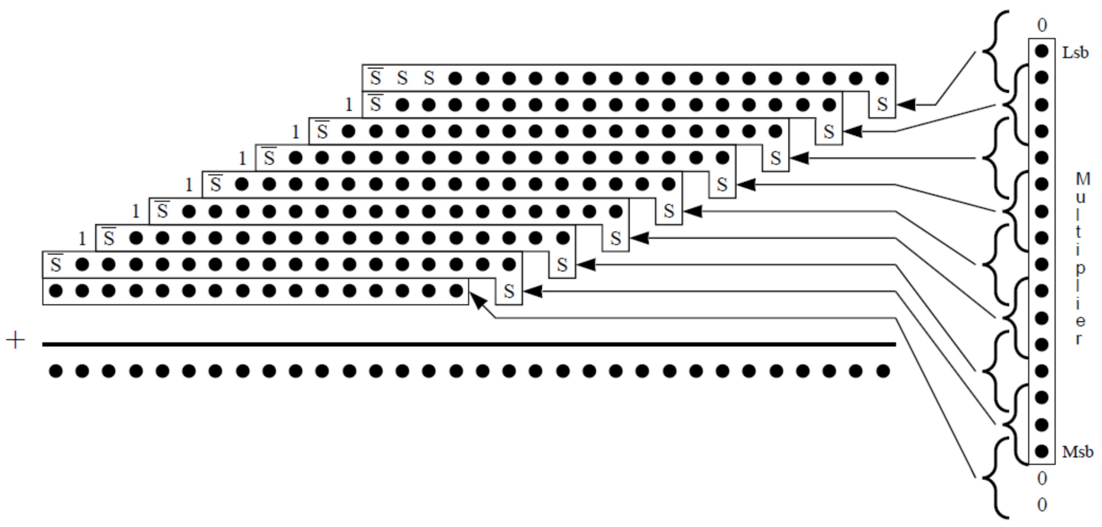

Figure 2.3: Sign extension of partial products on 16 bits

2.2 Dadda-tree

The first step for the Dadda-tree implementation is the definition of the d_j coefficients, that represent the maximum number of operands for each level.
The lowest layer is initialized considering $d_2 = 2$; the next ones are computed by means of the following formula, until d_j reaches a value equal or higher with respect to the initial number of operands:

$$d_{(j+1)} = \text{floor}(3/2 \cdot d_j)$$

It can be noticed that the factor 3/2 is related to the compression ratio of the full adder, the basic element used in the compression between two layers.
The evaluated d_j coefficients are reported in the following table.

d_j	Number of operands
d_2	2
d_3	3
d_4	4
d_5	6
d_6	9
d_7	13
d_8	19

After defining the d_j coefficients, the first seventeen partial products, generated by the PPUNIT, have been organized using the dot notation on a spreadsheet file, taking into account the correct weight of the bits. Then, full adders and half adders have been introduced in order to reduce the maximum number of operands to thirteen. Part of the spreadsheet file is represented in the following figure.

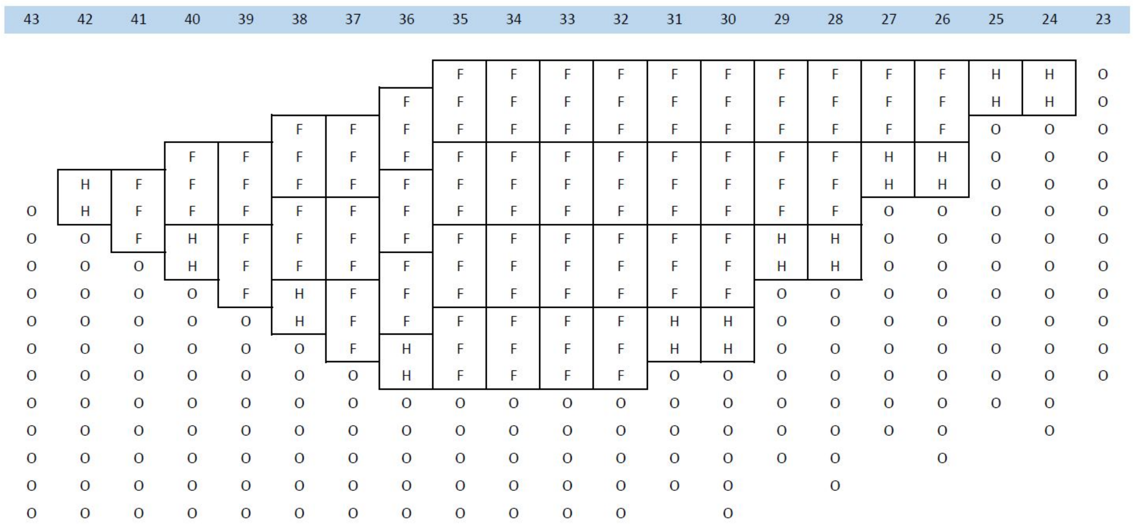

Figure 2.4: Layer 7 allocation scheme

The same approach has been repeated for the next levels to further reduce the number of operands until reaching a value equal to 2, related to the last Dadda-tree layer.

In order to maintain a hierarchical approach, each level has been described as a unique entity, identified by the blue blocks in figure 2.1.

By observing the several defined layers, a common computational structure can be recognized, as highlighted in the figure below; due to the huge amount of operators to be instantiated, it is useful to rely on the definition of Carry Save Adders that are able to work with multiple bits.

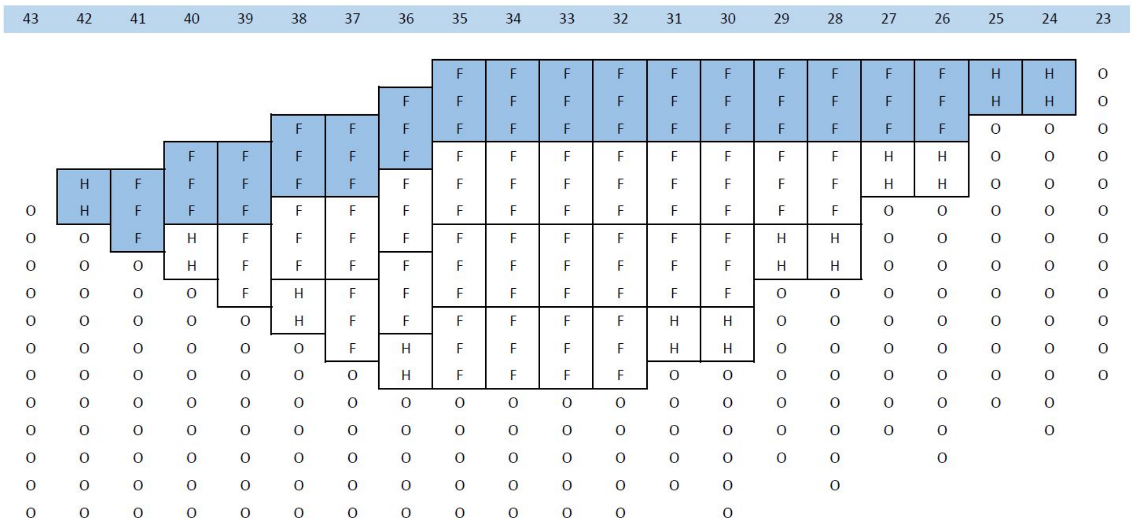

Figure 2.5: Common structure

It is possible to describe this MCSA unit as a generic entity, composed by three half adders and an arbitrary number of full adders. Since the number of full adders needs to be customized, a generate statement has been employed in the hardware description.

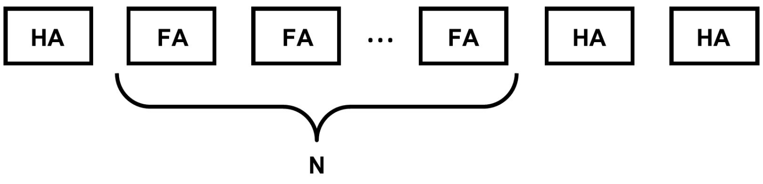

Figure 2.6: MCSA

In each layer, a correct number of MCSA units has been employed to provide the correct number of operands for the next layer.
The output of each layer includes the outputs of MCSA units and bits that are not involved in the compression operation.

The last two operands are represented in carry stored form, so in order to obtain a unique result expressed on 64 bits, a two operands adder needs to be employed. Therefore it has been choosen to implement it with a behavioral hardware description.

It can be observed that the Dadda-tree approach tends to move the computational cost to the the last step. For this reason the two operands adder has to work with a parallelism of 64 bits.
To overcome this issue, a faster adder, relying on a specific implementation, could be introduced.

2.3 Simulation and Synthesis

The behavioral multiplier defined in stage 2 has been replaced with the designed MBE multiplier.

```
sig_mpy: Significand_Multiplier PORT MAP(A => A_SIG,
                                         B => B_SIG,
                                         Z => prod);
```

The same testbench developed for the structure described in section 1.4 has been employed in order to verify the correct functionality of the architecture. The ModelSim simulation did not produce any error, providing the correct results.

Row	***fp_prod.hex***	***results_vhdl.txt***
0	000000000	00000000
1	3dc3910d	3dc3910d
2	0bc80005	0bc80005
3	3f278ddf	3f278ddf
4	0b100005	0b100005
5	3f800000	3f800000
6	0c440004	0c440004
7	3f278ddf	3f278ddf
8	0da20000	0da20000
9	3dc3910d	3dc3910d

The next step consists in performing the synthesis of the actual design, with the aim of finding the maximum operating frequency and the corresponding area.
The same iterative approach, defined in the section 1.3, has been used also in this case, obtaining the results below.

compile + optimize_registers		
$\mathbf{T_{ck,\ min}}$ **[ns]**	$\mathbf{F_{ck,\ max}}$ **[MHz]**	**Area [um^2]**
0.72	1339	2754

compile ultra		
$\mathbf{T_{ck,\ min}}$ **[ns]**	$\mathbf{F_{ck,\ max}}$ **[MHz]**	**Area [um^2]**
0.78	1282	2748

It can be noticed that the results of the different compile operations are comparable. The performance obtained with the *optimize_registers* command are slightly better, with a consequence on area, which slightly increases.

2.4 Conclusions

All the obtained results are summarized in the following tables.

With fine grain pipeline				
Description	**compile + optimize_registers**		**compile_ultra**	
	$T_{ck, min}$ [ns]	**Area [um^2]**	**$T_{ck, min}$ [ns]**	**Area [um^2]**
Behavioral	0.80	4897	1.50	4479
MBE	0.72	2754	0.78	2748

Without fine grain pipeline		
Implementation	**$T_{ck, min}$ [ns]**	**Area [um^2]**
CSA	4.48	4806
PPARCH	1.56	4097

In general it can be noticed that the implementation of the fine grain pipelining is able, considering any description, to lead to better performances. These results are achievable providing that a proper positioning of pipeline registers is applied. This condition is obtained through the command *optimize_registers*, since it forces the compiler to apply a retiming operation.

In this particular scenario, the introduced pipeline can potentially double the working frequency of architecture. As a drawback, the area estimation increases due to the allocation of additional registers.

The MBE multiplier introduction is able to improve the performance with respect to the simple behavioral description. This is due to several aspects of the hardware implementation: first of all, the radix-4 approach allows to reduce the number of computational steps. Further more, thanks to the Modified Booth Encoding, the generation of initial partial products is sped up. From the acquired data, the maximum frequency can increase of a factor 1.1.

Since the MBE multiplier is a feedforward structure, pipeline levels can be introduced in order to further improve performances. Also in this case the retiming operation can be exploited to balance correctly the different delays related to combinational paths.

A huge advantage can be noticed in terms of area, which for the MBE case is reduced of a factor 1.8 with respect to the behavioral description. This decrement is obtained thanks to the ALAP approach carried by the Dadda-tree and the Stanford sign extension, which allow to use the minimum number of allocated elements.