Politecnico di Torino

Integrated Systems Architecture

A.A. 2020/2021

Prof. Guido Masera
Prof. Maurizio Martina

Digital arithmetic

17 Dec 2020

Antona Gaspare	275336
Barrera Alessandro	275337
Liu Huicai	273375

Link to GitHub: `https://github.com/Dragosk97/ISA-laboratories.git`

Contents

1 Introduction 2

2 Floating point multiplier 3

2.1 Test-bench and simulation 4

2.2 Multiplier architectures 4

2.3 Fine-grain Pipelining and optimization 5

3 MBE multiplier with the Dadda tree 7

3.1 Modified Booth Encoding and its reduction 7

3.2 Dadda tree 8

3.3 Design implementation 9

3.3.1 MBE units 9

3.3.2 Sign extension 10

3.3.3 Signal rearrangement into modified dot matrix 10

3.3.4 Dadda tree implementation 12

3.3.5 Final adder 13

3.4 Simulation 14

3.5 Synthesis and results 14

4 Conclusions 16

1 Introduction

In the first laboratory experience, in the implementation of the filter, the addition and multiplication operators are described as behavioural in VHDL. On the contrary, the aim of this second laboratory is to analyze which performances are achievable by using predefined architecture provided by Design Compiler for the significands multiplication in a floating point multiplier, like the **carry-save (csa)** and **parallel-prefix (pparch)**. Several optimization were also applied to the behavioural multiplier using Design Compiler commands **optimize_registers** and **compile_ultra**. In addition, a MBE multiplier for unsigned data is implemented with the purpose of substituting the behavioural multiplier.

2 Floating point multiplier

For this laboratory a floating point multiplier is used. As shown in figure 1, the multiplier is made by several blocks and in this case they are divided into 4 stages of pipeline. After verifying the correct functionality of the given VHDL description, the input registers are added to the original architecture and the verification is performed again.

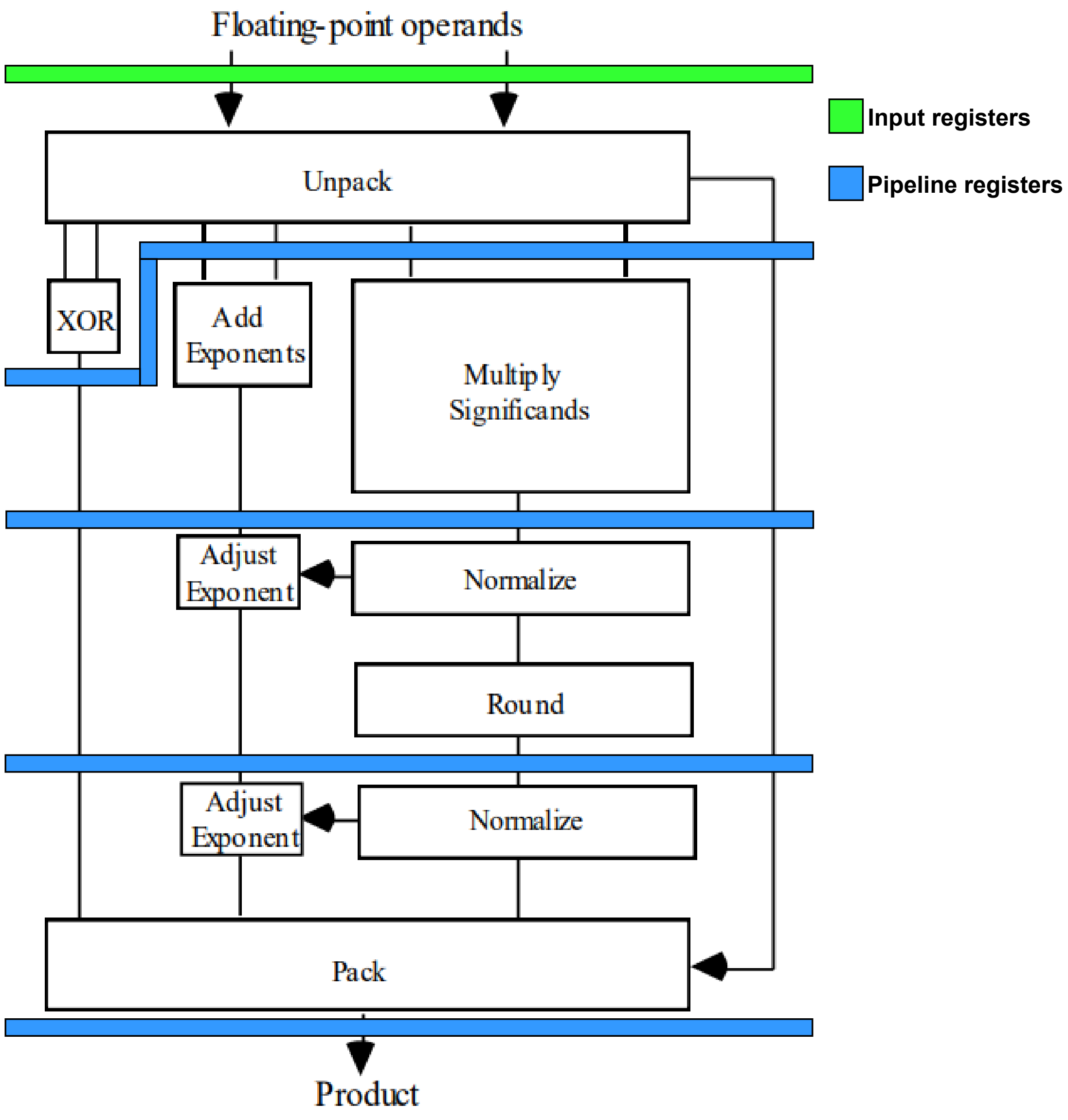


Figure 1: Blocks of floating point multiplier

2.1 Test-bench and simulation

In order to prove that the circuit works correctly, a simulation is needed. In this case, the input file and the reference output file were supplied in hexadecimal notation, therefore the VHDL test-bench entities were modified to comply with this notation. This was achieved by employing the **hread()** and **hwrite()** functions in the *data_maker.vhd* and *data_sink.vhd* files respectively. Moreover, the **data_sink** entity is designed in such a way as to have the output writing only in correspondence to a valid output. In order to obtain this behaviour, the constants **N_pipe** and **N_samples** are used to set the number of pipeline stages and the number of samples to be elaborated. The variable **count** is used as a counter which controls the output writing. This design made possible to change the number of pipeline stages and correctly write the results without many modifications.

The simulation is performed with ModelSim by exploiting a top-level Verilog module test-bench, which instantiates the device-under-test along with the VHDL entities **data_sink**, **data_maker** and **clk_gen**. These units are connected together in order to read and write the data using files. After the simulation, the output file *fp_out.hex* is generated and has to be compared with the reference file *fp_prod.hex.*

This task is accomplished by the Python script *output_check.py.* Besides reading the two files to be compared line by line, a case conversion is needed because of the fact that the **hwrite()** function uses capital letters, while the reference file has lowercase letters. This is simply achieved by the use of the string method **lower()**.

Following this procedure, all the architectures were tested and resulted correct.

2.2 Multiplier architectures

The architecture was synthesized on Synopsys Design Compiler three times, each one using a different block for the multiplication of the significands:

- without specifying any architecture for the multiplier, a behavioural description is used in VHDL;
- implementing the multiplier as a carry-save-adder multiplier by using the command `set_implementation DW02_mult/csa [find cell *mult*]`;
- implementing the multiplier as a parallel-prefix multiplier by using the command `set_implementation DW02_mult/pparch [find cell *mult*]`.

These architectures are synthesized using customized scripts, named *freq_area.scr*, *freq_area_CSA.scr* and *freq_area_PPARCH.scr* respectively. In order to have the possibility to use the `set_implementation` command, each script flattens the hierarchy with the command `ungroup -all -flatten`. It removes hierarchical boundaries and also timing is improved by reducing the levels of logic.The maximum frequency and the corresponding area were found and the results are shown in table 1. Here it is possible to highlight that the carry save adder multiplier has the greater clock period so the worst performance and also the occupied area is the greatest one probably due to a major complexity of this implementation. While the behavioural description and the parallel-prefix multiplier are very similar in performance and in area.

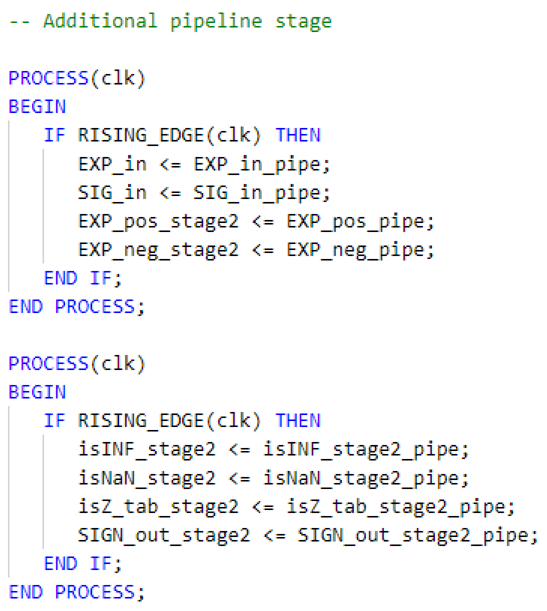
Version	Clock period ns	Maximum frequency MHz	Area μm^2
Behavioural description	1.56	641.0	4047.7
Carry-save-adder multiplier	4.3	232.6	4861.4
Parallel prefix multiplier	1.59	628.9	4011.1

Table 1: Results from the synthesis.

2.3 Fine-grain Pipelining and optimization

In order to increase the operating frequency, the significands multiplier in behavioural description can be pipelined, since generally it is the slowest block. This is achieved by adding one register at the output of the significands multiplier. Also other registers have been inserted in order to respect the correct timing. At this point, it is possible to exploit Design Compiler capability of optimizing registers position, namely re-timing, by synthesizing the architecture:

- with the `compile` and the `optimize_registers` commands;
- with the `compile_ultra` command instead of the previous two.



```
-- Additional pipeline stage

PROCESS(clk)
BEGIN
   IF RISING_EDGE(clk) THEN
      EXP_in <= EXP_in_pipe;
      SIG_in <= SIG_in_pipe;
      EXP_pos_stage2 <= EXP_pos_pipe;
      EXP_neg_stage2 <= EXP_neg_pipe;
   END IF;
END PROCESS;

PROCESS(clk)
BEGIN
   IF RISING_EDGE(clk) THEN
      isINF_stage2 <= isINF_stage2_pipe;
      isNaN_stage2 <= isNaN_stage2_pipe;
      isZ_tab_stage2 <= isZ_tab_stage2_pipe;
      SIGN_out_stage2 <= SIGN_out_stage2_pipe;
   END IF;
END PROCESS;
```

Figure 2: Additional registers inside the stage 2.

In figure 2 it is possible to notice that intermediate signals *_pipe* are created in order to not change the output interface.

The `optimize_registers` command performs the re-timing of sequential cells for a pipelined design. In particular, this technique consists in the re-positioning of the internal registers in order to reduce the critical path. But the input and the output registers are not touched in order to have the same interface and make possible the accessibility from the outside.

To have several optimization, it is possible to enable the *ultra* mode in Design Compiler using the command `compile_ultra` instead of `optimize_registers`+`compile`. This command works in a different way respect to the previous one by optimizing all the components at lower level. So, it guarantees the maximum achievable optimization of the structure.

In order to do these optimizations, two different scripts are implemented: *optreg_script.scr* where the command `optimize_registers` is exploited and *compultra_script.scr*, where `compile_ultra` is used. The results are summarized in table 2.

Optimization	Period *ns*	Maximum frequency *MHz*	Area μm^2
Behavioural description	1.56	641	4047.7
optimize_registers	0.78	1282.1	5180.6
compile_ultra	1.5	666.7	4198.5

Table 2: Results from fine grain pipelining and optimization

It is possible to understand that the performance of the unsigned multiplier in behavioural description is improved thanks to the fine-grain pipelining applied by using the `optimize_registers` command. The clock period is reduced by about a half and consequently the maximum frequency doubles. The cost paid is a considerable increase of area in both combinational and non combinational contributions.

With the optimization of `compile_ultra` command, the performance is slightly better than the very first implementation as the frequency has a small increase, as well as the area.

The maximum frequency obtained from the command `compile_ultra` is lower than the one from the `optimize_registers`. By opening the **Design Compiler® Optimization Reference Manual**, it is possible to understand that the command does not perform register retiming. In order to do that, there are two possible ways:

- by using the command `optimize_registers`;
- by using the `-retime` option supported by `compile_ultra`, which enables Design Compiler to perform local retiming moves to improve worst negative slack. This capability, called "adaptive retiming", optimizes an entire design. It works best with general non-pipelined logic.

3 MBE multiplier with the Dadda tree

An interesting thing to do is to substitute the unsigned multiplier with something different from the architectures proposed by Synopsys Design Compiler. The multiplier chosen is based on the 24-bit Modified Booth Encoder and a proper Dadda tree. For the final sum a behavioural adder is picked. The choice of 24 bits is due to the fact that the fractional part of the significand in a single precision floating point multiplier is on 23 bits.

3.1 Modified Booth Encoding and its reduction

Instead of using AND gates to get the partial products, a Booth encoder is used to reduce the hardware utilization. In particular, the MBE is based on a Radix-4 approach as shown in table 3, where a is the multiplicand and b is the multiplier. The partial product p_j is produced based on the 3-bit set of b.

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

Table 3: Modified Booth Encoding

It is interesting to notice that in a 24-bit multiplier with the Modified Booth Encoder, only 13 partial products will be produced instead of 24. But the disadvantage is that all the operands could be signed numbers, therefore they should be extended up to 48 bits with the proper sign. This cancels the benefit of the reduction of the number of operands.

A simple and effective technique to reduce the number of adders required in order to cover the partial product sign extension bits is presented in [1] and the final result is shown in figure 3. The number of dots in each row is 25, since it could be zero, equal to the multiplicand a or twice as much, except for the last row, reason why it is limited to 24 bits. The S, 1 and $\bar{S}$ are due to the sign extension. From table 3, it is easy to notice that the sign of the partial product is negative when $b_{2j+1} = 1$. This information can be used for the sign reduction, indeed the S of each partial product corresponds to the MSB of each triplet of the multiplier. Finally, the dots of each row should be negated if the corresponding S is equal to 1. In order to perform the sum of all the partial product, a Dadda tree is exploited. It will be explained in the following section.

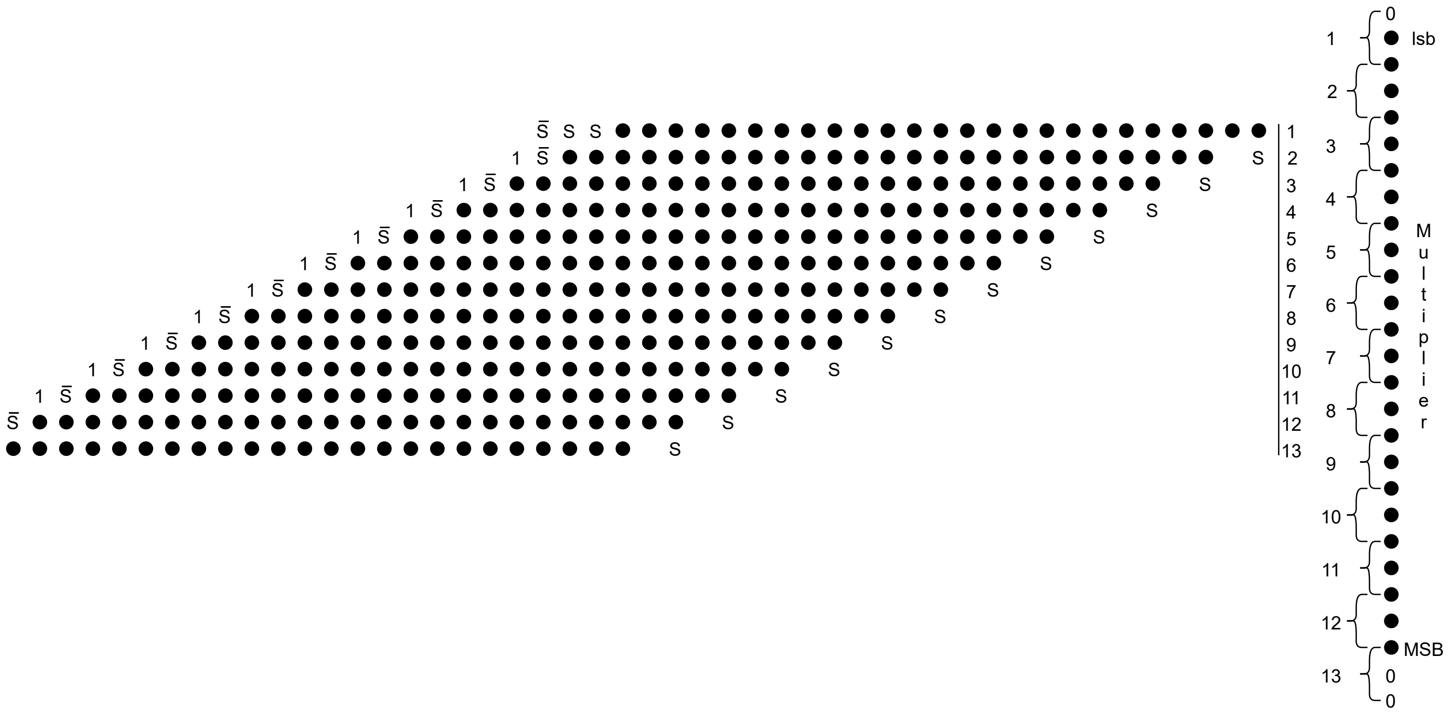


Figure 3: Results of the Reduced Extension Modified Booth Encoder

3.2 Dadda tree

The designed multiplier exploits MBE in order to generate partial products, which have to be added together. The Dadda tree is a possibility to reduce the dot matrix previously composed until only two rows are left, which will be the operands of the final adder.

This reduction is obtained by allocating full adders and half adders as needed in order to compress the maximum number of rows as predicted by the following expression:

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

where l_j is the maximum number of rows at level j and $l_0 = 2$. This relation is obtained considering the maximum compression factor of 3/2 due to the full adders.

The Dadda tree follows an As Late As Possible (ALAP) approach, meaning that the full adders and the half adders are allocated in the minimum amount in order to obtain the required l_j. It is important to notice that, when evaluating the number of rows in the next level, also the carry-out from the previous weight must be taken into account. In other words, when elaborating a certain column, the expression to be evaluated is $d = r + c - l_j$, where r is the number of rows, c carry-out bits from the full adders and half adders of the previous weight and l_j maximum number of elements in the next level.

If $d \leq 0$ no compression is needed, otherwise the total reduction in this column must be equal to d. This value is needed in order to predict how many full adders and half adders are necessary to provide the correct reduction, considering that the firsts reduce by 2 bits and the others by 1 bit. As a consequence, the number of full adders n_{FA} and half adders n_{HA} can be easily computed starting from the value of $d > 0$ as:

$$n_{FA} = \left\lfloor \frac{d}{2} \right\rfloor,$$
$$n_{HA} = d \bmod 2.$$

In the presented work, a 24-bit MBE multiplier is designed, providing a dot matrix with 13 rows to be fed to the Dadda tree. This corresponds to the following sequence of maximum number of rows:

$$l_5 = 13, \quad l_4 = 9, \quad l_3 = 6, \quad l_2 = 4, \quad l_1 = 3, \quad l_0 = 2.$$

It is possible to notice that 5 reduction stages are needed.

3.3 Design implementation

The multiplication operation was organized into five steps: the Modified Booth Encoding to generate the partial products, the sign extension, the rearrangement of the signals in order to prepare the dot matrix, the Dadda tree reduction and the final addition.

The final design is described in the VHDL file *MBE_mult.vhd* obtained by the Python script *dadda_algorithm.py*, which implements the Dadda tree reduction and exploits the template file *MBE_mult_template.vhd*. In the latter, the remaining VHDL descriptions are included.

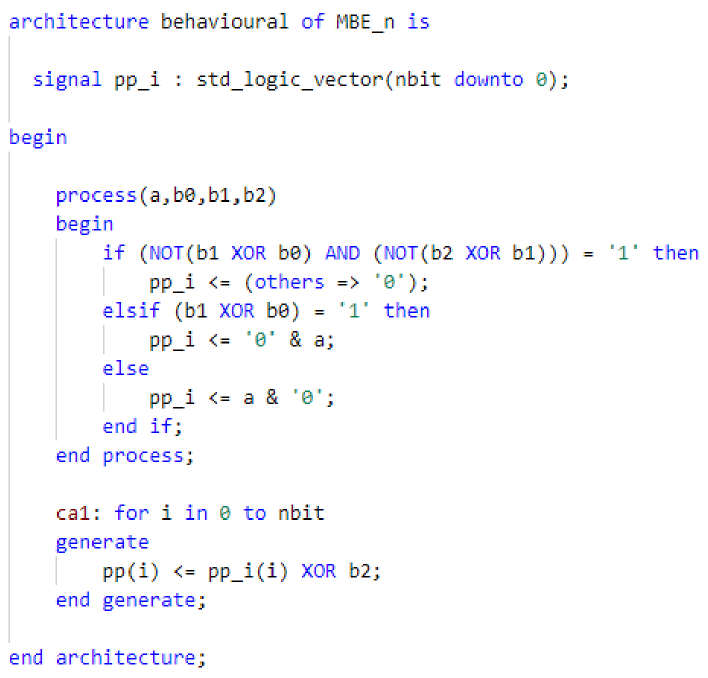
3.3.1 MBE units

As said before, each partial product is generated based on 3 bits of the multiplier, table 3. For this purpose, the following logic statements are used:

$$q_j = \begin{cases} 0 \ \text{ if } \ (\overline{b_{2j} \oplus b_{2j+1}})(\overline{b_{2j+1} \oplus b_{2j}}) \\ a \ \text{ if } \ (b_{2j} \oplus b_{2j-1}) \\ 2a \ \text{ if } \ (\overline{b_{2j} \oplus b_{2j+1}})(b_{2j+1} \oplus b_{2j}) \end{cases}$$

It is interesting to notice that q_j would be on 25 bits and it is not the final partial product, since it can only be positive. The partial product pp_j, namely the dots represented in figure 3, is obtained by $pp_j = q_j \oplus b_{2j+1}$.
A problem occurs: a shift register cannot be exploited for the generation of all the partial product since it would require 13 clock cycles and this is not wanted for the unsigned multiplier. Therefore, a basic block with as inputs the multiplicand a and

3 bits of the multiplier $b_{2j+1}, b_{2j}, b_{2j-1}$ and as output pp_j, is described in VHDL, *MBE.vhd*, as showing in figure 4. Then, this block is instantiated 13 times in order to obtain all the partial products at once.



```
architecture behavioural of MBE_n is

  signal pp_i : std_logic_vector(nbit downto 0);

begin

    process(a,b0,b1,b2)
    begin
        if (NOT(b1 XOR b0) AND (NOT(b2 XOR b1))) = '1' then
            pp_i <= (others => '0');
        elsif (b1 XOR b0) = '1' then
            pp_i <= '0' & a;
        else
            pp_i <= a & '0';
        end if;
    end process;

    ca1: for i in 0 to nbit
    generate
        pp(i) <= pp_i(i) XOR b2;
    end generate;

end architecture;
```

Figure 4: Description of the MBE unit

In VHDL, a signal called **mbe_out** is created, which is an **array** of **std_logic_vector**. Each element of the array is a standard logic vector of 25 bits and here the corresponding partial product is saved as shown in figure 5.

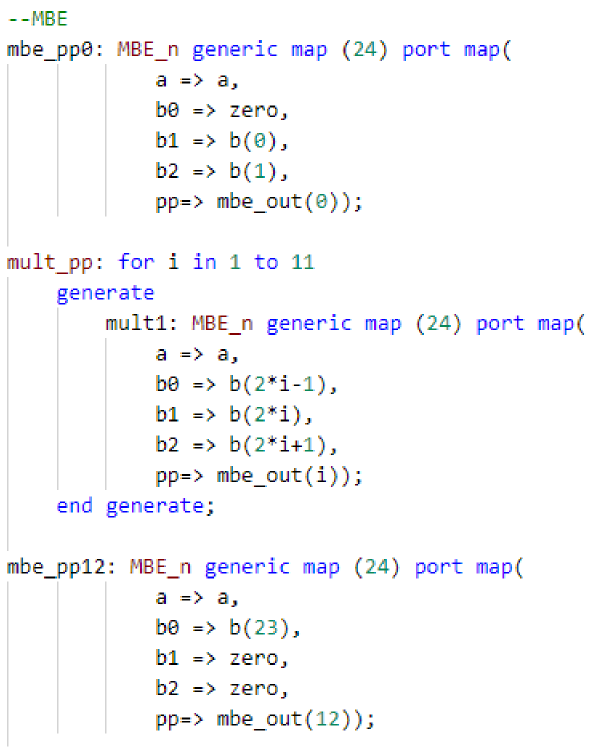
3.3.2 Sign extension

As described in section 3.1, in order to perform the sign extension, a particular reduction method could be exploited, so the signal **mbe_out** is not enough. All the '1', S, $\overline{S}$ should be added in their proper position, therefore a new array signal **in_dadda** is used, where each element is a standard logic vector of 29 bits. This width is necessary for the worst case corresponding to the first partial product.

As shown in figure 6, the S needed to perform the +1 in a 2's complement conversion, is inserted to the last position of each row. The real weight of each bit will be taken in consideration in the signal **dadda_i** during the rearrangement as explained in the following section.

3.3.3 Signal rearrangement into modified dot matrix

In order to simplify the connections within the Dadda tree, the input signals collected in the array **in_dadda** had not to be rearranged into the matrix shown in figure 3,

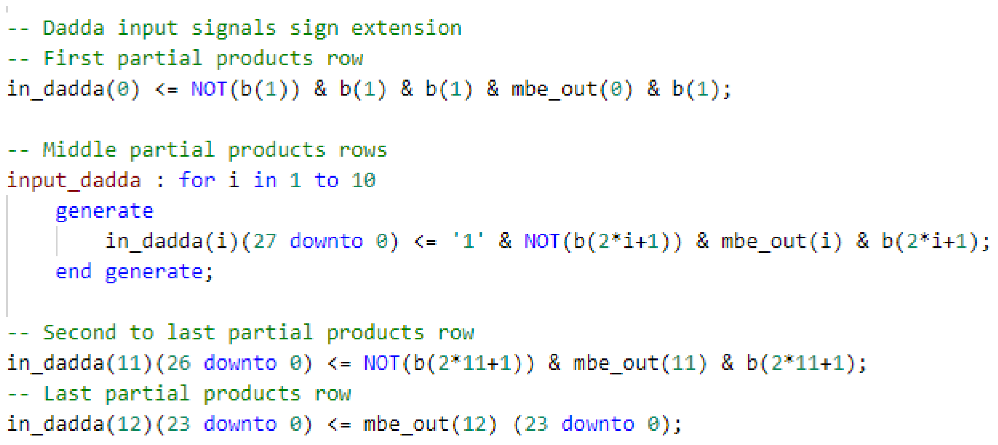


```
--MBE
mbe_pp0: MBE_n generic map (24) port map(
            a => a,
            b0 => zero,
            b1 => b(0),
            b2 => b(1),
            pp=> mbe_out(0));

mult_pp: for i in 1 to 11
    generate
        mult1: MBE_n generic map (24) port map(
            a => a,
            b0 => b(2*i-1),
            b1 => b(2*i),
            b2 => b(2*i+1),
            pp=> mbe_out(i));
    end generate;

mbe_pp12: MBE_n generic map (24) port map(
            a => a,
            b0 => b(23),
            b1 => zero,
            b2 => zero,
            pp=> mbe_out(12));
```

Figure 5: MBE units instances



```
-- Dadda input signals sign extension
-- First partial products row
in_dadda(0) <= NOT(b(1)) & b(1) & b(1) & mbe_out(0) & b(1);

-- Middle partial products rows
input_dadda : for i in 1 to 10
    generate
        in_dadda(i)(27 downto 0) <= '1' & NOT(b(2*i+1)) & mbe_out(i) & b(2*i+1);
    end generate;

-- Second to last partial products row
in_dadda(11)(26 downto 0) <= NOT(b(2*11+1)) & mbe_out(11) & b(2*11+1);
-- Last partial products row
in_dadda(12)(23 downto 0) <= mbe_out(12) (23 downto 0);
```

Figure 6: Sign extension

but in a version where all the dots are positioned as high as possible, that is to say with the lowest possible row index.

As **in_dadda** is an array of **std_logic_vectors**, it can be visualized as a rectangular matrix, in such a way that **in_dadda(x)(y)** indicates the element in the row x and column y. It is worth noticing that this matrix is not full, because not all the elements in the array have the same length. It is easier to visualize the transformation if the column index is growing from right to left.

The transformation needed to extract the dot matrix which will be fed as input to the Dadda tree consists into shifting the rows to the left and inserting the LSB S bit in the underlying row, obtaining the matrix in figure 3. Then, the columns have to be shifted upward as much as possible. The final matrix is shown in figure 7.

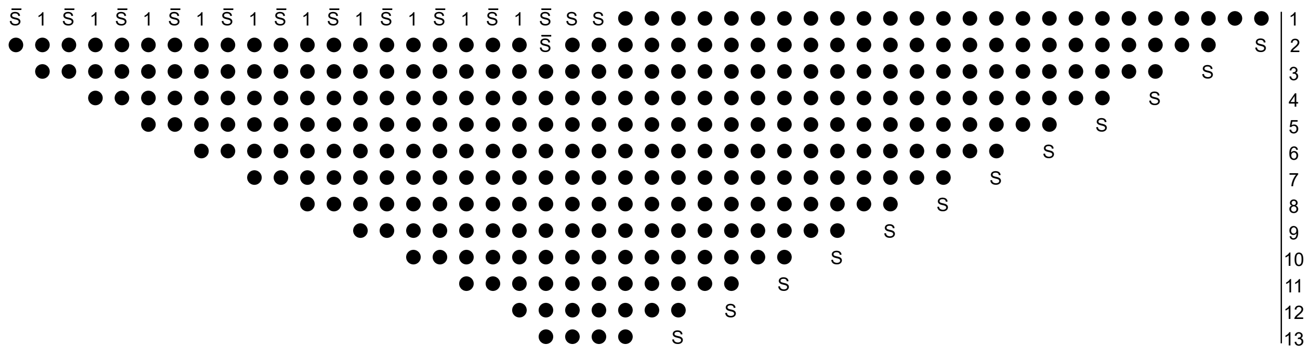


Figure 7: Input dot matrix to the Dadda tree

All the signals within the Dadda tree are collected in the instance **dadda_i** of an array of arrays of **std_logic_vector** signals. In particular, it is used in such a way that the signal **dadda_i(x)(y)(z)** indicates the bit at reduction stage x, in the row y and column z. The column is also corresponding to the weight of the bit. It is worth noticing that the stage with $x = 0$ is the starting dot matrix from which the reduction has to be performed, shown in figure 7, obtained with the above mentioned rearrangement.

3.3.4 Dadda tree implementation

The Dadda tree is implemented as a netlist by allocating **FAs** and **HAs** in order to reduce the number of rows. As VHDL resulted inconvenient, this is achieved by exploiting the Python script *dadda_algorithm.py*, which decides whether to insert a full adder, a half adder or to propagate the unprocessed bits between two consecutive stages according to the algorithm in section 3.2.

For this purpose, two nested for loops are used to pass each stage, evaluating column by column the expression:

```
diff = row_num[stage][col] + num_carry - row_target[stage+1]
```

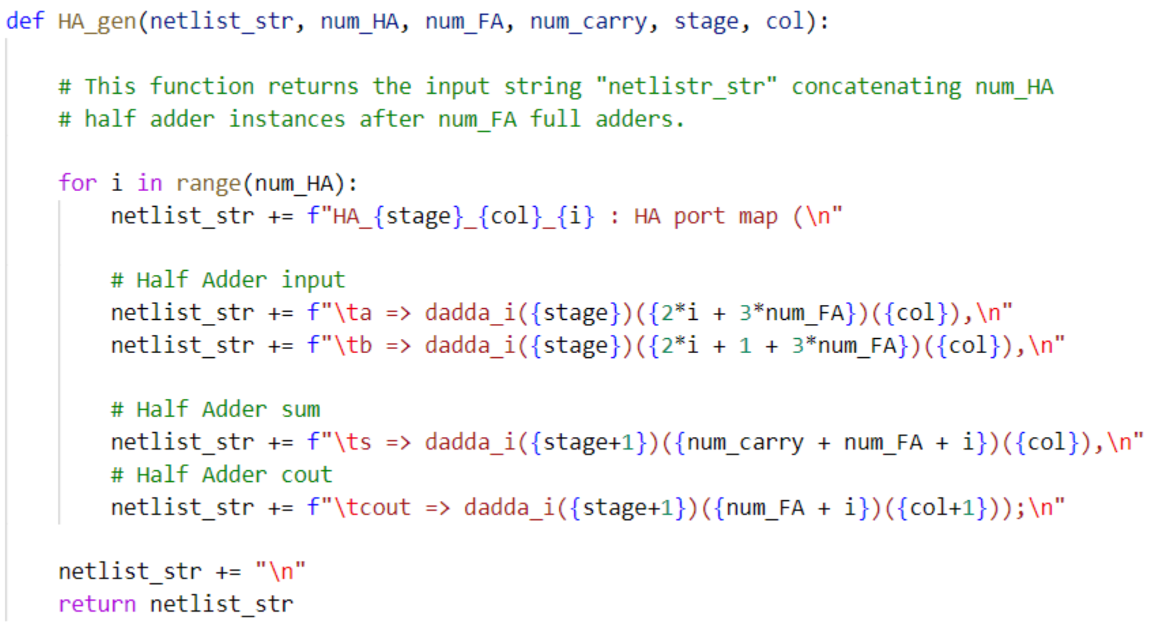
where:

- **row_num** is a list of lists such that **row_num**[**x**][**y**] is the number of rows at stage x and column y. The first element **row_num[0]** is initialized according to the starting matrix of the Dadda shown in figure 7;
- **row_target** is a list containing the required maximum number of rows per stage as predicted by the Dadda algorithm shown in section 3.2;

- **num_carry** is used to take into account the carry-out bits of the adders of the previous weight.

The variable **diff** is used to retrieve the number of needed FAs and HAs. This information is used by the functions **FA_gen()**, **HA_gen()** and **unproc_prop()**, which are defined in the module *generate_lib.py*. These functions produce the VHDL instances of FAs, HAs or signal assignments respectively, which are appended in the **nestlist_str** string.

In order to correctly map the bits between the two consecutive stages, the following policy has been adopted: when **num_FA** full adders are instantiated, the sum bits are positioned with the lowest row index after the first **num_carry** bits. On the other hand, the carry-out bits are positioned in the next weight with the lowest row index possible. Similarly is done for the half adders, taking into account the already positioned FAs. And lastly, the same is done for the unprocessed bits propagation. For example, in figure 8 the **HA_gen()** function definition is shown.



```python
def HA_gen(netlist_str, num_HA, num_FA, num_carry, stage, col):

    # This function returns the input string "netlistr_str" concatenating num_HA
    # half adder instances after num_FA full adders.

    for i in range(num_HA):
        netlist_str += f"HA_{stage}_{col}_{i} : HA port map (\n"

        # Half Adder input
        netlist_str += f"\ta => dadda_i({stage})({2*i + 3*num_FA})({col}),\n"
        netlist_str += f"\tb => dadda_i({stage})({2*i + 1 + 3*num_FA})({col}),\n"

        # Half Adder sum
        netlist_str += f"\ts => dadda_i({stage+1})({num_carry + num_FA + i})({col}),\n"
        # Half Adder cout
        netlist_str += f"\tcout => dadda_i({stage+1})({num_FA + i})({col+1}));\n"

    netlist_str += "\n"
    return netlist_str
```

Figure 8: **HA_gen()** definition.

When the reduction algorithm is terminated, **netslist_str** is inserted in the template file *MBE_mult_template.vhd*, exploiting the Python library *jinja2* in order to produce the final file *MBE_mult.vhd*.

3.3.5 Final adder

After the Dadda tree only two rows of 48 bits are remained, but it is interesting to notice that there are 2 bits in each column except the column of weight 1. This position could be filled by a zero and then a sum on 48 bits can be performed. But it is also possible to add a HA for the LSBs in order to get directly the final LSB and to perform the sum of two numbers on 47 bits. The second alternative has been chosen. It is possible to notice that it does not introduce any further delay since this HA works in parallel to the Dadda tree. The final adder is described in behavioural, so its choice is left to Design Compiler.

3.4 Simulation

In order to verify the correct behaviour of the designs, as usual, a simulation is performed with ModelSim by exploiting a top-level Verilog module test-bench. The three VHDL entities **data_sink**, **data_maker** and **clk_gen** are modified in a proper way.

Before simulating the complete multiplier, the **MBE_unit** was verified. This is achieved by exploiting the same verification flow, with the difference that the input signals were not read by a file, but generated by a 3-bit counter and fed as input to the b_i input ports. This procedure made possible to test all the combinations of the multiplier set of three bits.

A MATLAB script is used for the generation of two files containing the input samples and the correct outputs for the verification of the MBE multiplier. The input file is given to ModelSim in order to run the simulation and a new output file is generated. Finally, the outputs of MATLAB and ModelSim are compared using the usual Python script *output_check.py*.

After demonstrating the correct behaviour of the multiplier, it is inserted inside the floating point unit in the place of the behavioural multiplier and it was simulated again. After this step, the circuit is given to Synopsys as described in the following section.

3.5 Synthesis and results

The architecture was synthesized on Synopsys Design Compiler using a customized script, named *mbe_syn.scr*. The synthesis was repeated until the slack became equal to 0. After different attempts the minimum clock period is set to 2.78 ns, corresponding to a maximum clock frequency equal to 359.7 MHz. It was also found the total area corresponding to the maximum frequency equal to 4647.8 μm^2, in table 4.

Version	Period	Maximum frequency	Area
	ns	MHz	μm^2
MBE	2.78	359.7	4647.8
MBE optimize_registers	0.79	1265.8	6965.5
MBE compile_ultra	1.46	684.9	4845.9

Table 4: Results from the synthesis

It is possible to highlight that the `optimize_registers` command is used to improve the performance thanks to the fine-grain pipelining. Indeed, the clock period is reduced a lot and consequently the maximum frequency becomes bigger. The cost paid is a considerable increase of area, both combinational and non combinational one.

On the other hand, with `compile_ultra` command, the performance is better than the first MBE implementation because the frequency significantly increases, but also the area worsen a little.

4 Conclusions

It is interesting to compare all the results obtained by the synthesis.

Version	Period	Max frequency	Area	Area · Delay
	ns	MHz	μm^2	$\mu m^2 \cdot ns$
Behavioural description	1.56	641.0	4047.7	6314.4
Carry-save-adder multiplier	4.3	232.6	4861.4	20904.0
Parallel prefix multiplier	1.59	628.9	4011.1	6377.6
Behavioural optimize_registers	0.78	1282.1	5180.6	4040.9
Behavioural compile_ultra	1.5	666.7	4198.5	6297.8
MBE	2.78	359.7	4647.8	12920.9
MBE optimize_registers	0.79	1265.8	6965.5	5502.7
MBE compile_ultra	1.46	684.9	4845.9	7075.0

Table 5: Comparison

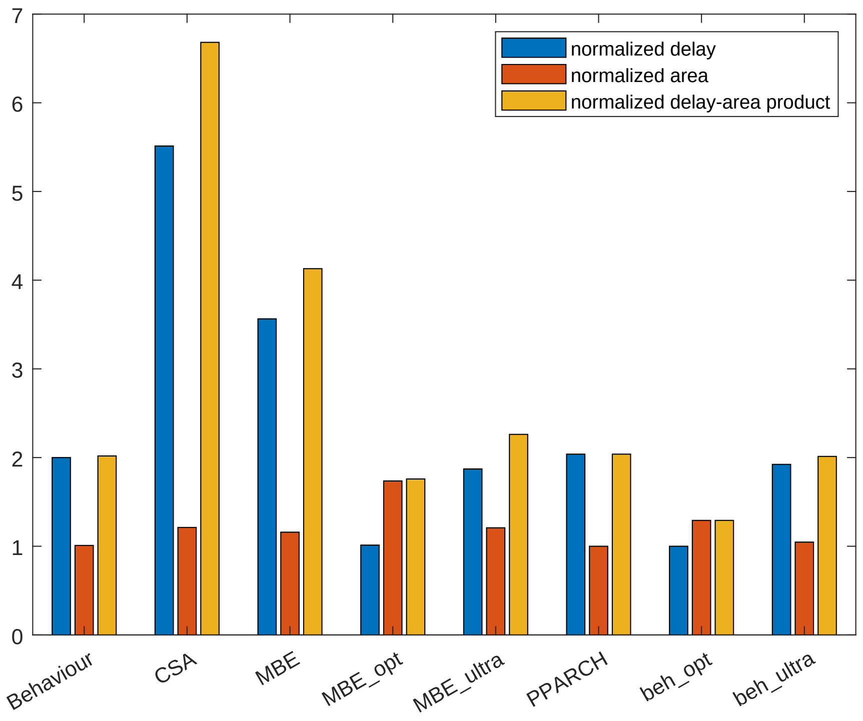


Figure 9: Comparison

The MBE multiplier gives worse performance compared to the behavioural operator, without any optimizations. In fact, the clock period increases and the frequency in this case is lower, probably due to the complexity of the MBE implementation that also uses more area, as shown in table 5. The smallest architecture is the parallel-prefix multiplier. The most performing implementation is obtained by using the `optimize_registers` command in the behavioural description. Without the use of the re-timing technique, the best solution is represented by the MBE multiplier synthesized with the `compile_ultra` command.

The overall best architecture according to the area-delay product is the behavioural multiplier synthesized with `optimize_registers` command. In figure 9 a visual representation of the results is shown, where the values are normalized with respect to the minimum one, i.e. $0.78\,ns$ for the delay and $4011.1\,\mu m^2$ for the area, while the delay-area product is obtained from the normalised values.

References

[1] M. Roorda. “Method to reduce the sign bit extension in a multiplier that uses the modified Booth algorithm”. In: *Electronics Letters* 22.20 (1986), pp. 1061–1062. DOI: `10.1049/el:19860727`.