Politecnico di Torino

III Facoltà di Ingegneria

# Lab 2
# Integrated Systems Architecture

Master degree in Electronics Engineering

Authors: Group 25

Francesco Bono, Claudia Golino, Mattia Mirigaldi

# Contents

CHAPTER 1

# Digital arithmetic and logic synthesis

The first part of this laboratory session consists in the verification and synthesis of the floating point pipeline multiplier coming from "opencores.org". The overall structure of the multiplier can be appreciate in the picture 1.1

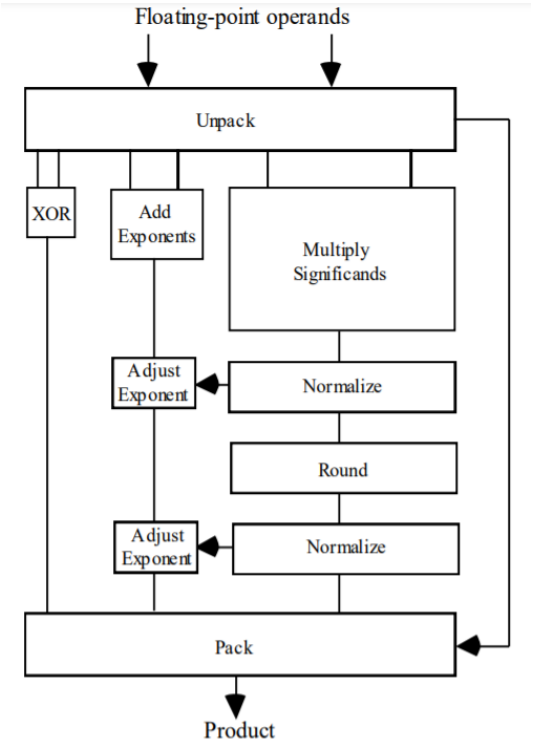


Figure 1.1: Schematic blocks of the FP mul

## 1.1 FP pipeline multiplier verification

In order to verify the FP pipeline multiplier a simple testbench has been created ("*tb_mul.vhd*") that instantiates the components "*FP_mul*" and "*datamaker*", the latter one gives in input to "*FP_mul*" the data contained in the file "*fp_samples.hex*".
The FP pipeline multiplier has been modified according to the lab request by adding registers to the

inputs. To verify the $FP\_mul$ the square values of the data contained in "$fp\_samples.hex$" has been evaluated, a screenshot of the waveforms can be seen in figure 1.2.

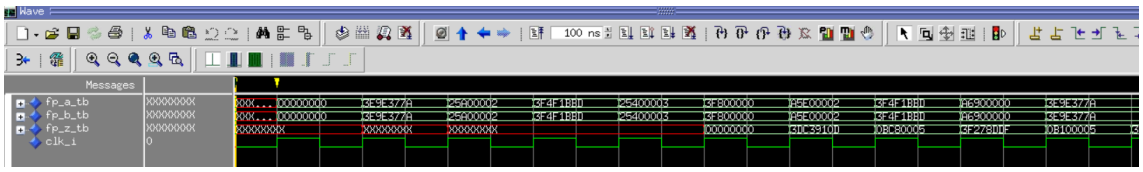


Figure 1.2: Waveforms simulation $FPpipelienmultiplier$

## 1.2 FP pipeline multiplier synthesis

Synopsys Design Compiler handles the behavioural description of adders and multipliers by directly inferring the component in the netlist. The component can be forced to a specific architecture with the command *set_implementation*, where the architecture is taken from a collection of ready-to-use blocks (called Design Ware).
In the lab assignment is asked to firstly flatten the hierarchy of the design and then force the multiplier architecture to be implemented with a carry save (CSA) and parallel-prefix (PPARCH) architecture. The synthesis scripts used are:

- "$syn\_mul\_flatten.src$" for the unflatten design
- $syn\_mul\_flatten\_csa.src$ for the unflatten design with multiplier forced to a carry-save architecture
- $syn\_mul\_flatten\_pparch.src$ for the unflatten design with multiplier forced to parallel-prefix architecture

Report summary of the logic synthesis,table 1.1:

| | Frequency [MHz] | Area | Power consumption [µW] |
|---|---|---|---|
| *Unflatten FP mul* | 223.71 | 3984.946 | 918.33 |
| *Unflatten FP mul with CSA* | 223.21 | 4806.08 | 962.82 |
| *Unflatten FP mul with PPARCH* | 223.71 | 3759.37 | 879.72 |

Table 1.1: Logic synthesis results

# CHAPTER 2

# Fine-grain Pipelining and optimization

## 2.1 Registers optimization

In this step a pipelined multiplier has been implemented. Instead of writing by hand the multiplier the Design Compiler capability of placing and optimization of the registers inside the circuit is exploited. At the output of the second stage a set of registers is added.
Three ways of compilation:

- Compile + Optimize_registers (C + O)
- Compile_ultra (CU)
- Compile_ultra + Optimize_registers (CU + O)

For each case we found maximum frequency, area and power consumption. The results are shown in the table 2.1:

| Synthesis | Frequency [MHz] | Area [$\mu m^2$] | Power consumption [µW] |
|---|---|---|---|
| C + O | 1052 | 5547.2 | 6709 |
| CU | 704 | 4303.6 | 3308 |
| CU + O | 1176 | 4750 | 6537 |

Table 2.1: Synthesis Pipelined w/ register optimization

From the table we notice that the maximum frequency is increased pipelining from 223 MHz to 1 GHz at the cost of an increased area and power consumption.
Applying the compile_ultra command instead of compile + optimize_registers the maximum frequency decrease, although higher than the frequency in the not-pipelined.
The command optimize_registers after the compile_ultra gives the best result. The frequency in this test reached the peak of 1.2 GHz with area and power consumption lower than in the first case. We can assert that even after the compile_ultra the Design Compiler is able to better optimize the registers inserted.

## 2.2 MBE-based multiplier

The last step of this laboratory is to use a MBE multiplier instead of the behavioural operator "*" in the Significands multiplier in Stage2 of the Floating point multiplier.

A 24 bits-MBE based multiplier for unsigned data was chosen to implement the Significand of the Floating Point multiplier: a $32x32$ bits multiplication was unnecessary because only 28 bits (from bit 20 to bit 47) of the multiplication were assigned to the output significand value of Stage 2.
For this reasons a 24-bits multiplier was chosen and the 28 MSBs of the result were assigned to the significand.

In order to explain the structure of the MBE multiplier, it's given below an example of how a multiplication works.
The result $P$ of the multiplication is given by $P = A \cdot B$.
In the binary form: $P = A \cdot (2^0 + 2^2)$, where in this case $B = 101_2$.
Here 2 partial productions can be highlighted: $p_0 = A \cdot 2^0$ and $p_1 = A \cdot 2^2$.
A final addition between partial products have to be done in order to retrieve the value of $P = p_0 + p_1$.
In our design the partial products have been be generated with no adder/subtractor, but with an encoder and the adder plane rely on a Dadda-tree, as shown respectively in paragraph 2.2.1 and 2.2.2.

### 2.2.1 Modified Booth Enconding

MBE is a Radix-4 approach wherewith partial products are produced. They are generated by:

- dividing the multiplier in 3 bit slices (with $b_{-1} = 0$), where two consecutive slices feature a 1-bit overlap, as in figure 2.1.

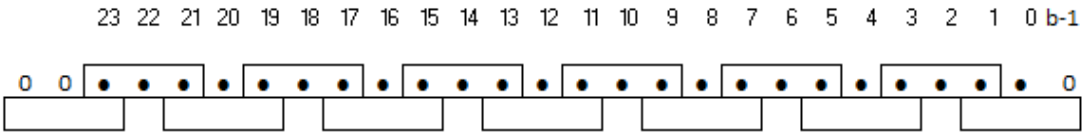


Figure 2.1: Bit slicing of the multiplier

- each triplet of bits is exploited to encode the multiplicand according to table 2.2.

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

Table 2.2: Modified Booth Encoding.

The partial products are added in an MBE-based multiplier using a Dadda tree structure. Furthermore a simple and effective technique is used to reduce the number of adders required for covering partial products: the sign extension.

### 2.2.2 Dadda tree structure

Dadda tree is a structure for adding multiple operands, in the case of the multiplier for adding partial products. The data has to be aligned so the partial terms with the same weight are in the same column.
With this structure partial addition by columns with Half-Adders (HAs) and Full-Adders (FAs): several levels of HA/FA are needed to compress the structure with n operands (partial products in this case) up to two operands.
When the tree has reduced the operands to two operands, they are added together with a fast two-operand adder. In this project the final adder was implemented with a simple "+" in VHDL to give the synthesizer the opportunity to choose the adder that gives better overall performance. In general the fast adder can be implement as preferred, by considering the constraints of the design.

**Reduction tree**
Dadda allocation of adders is As-Late-As-Possible (ALAP), meaning that at each level $j$ of reduction the minimum number of FAs and HAs is allocated in order to have the maximum number of operands allowed at the next level $l_{j-1}$.
In general at level $j$ of the reduction tree the maximum number of elements is:

$$l_j = \frac{3}{2} \cdot l_{j-1} \tag{2.1}$$

with $j$ is the level's number.
Considering a 24×24 bits multiplier the maximum number of operands for each level is reported below:

- $l_0 = 2$;
- $l_1 = 3$;
- $l_2 = 4$;
- $l_3 = 6$;
- $l_4 = 9$;
- $l_5 = 13$;
- $l_6 = 19$:
- $l_7 = 28$;

As a consequence, we need eight levels to compress 24 operands to two operands by the means of HAs and FAs.

**Sign extension**
In order to reduce the number of half adders and full adders sign extension bits in the Dadda-tree must be simplified so the height (maximum number of items to be added in any column) of the dot is reduced. In the Fig.2.2 we have an example on a 16x16 bits multiplication of the position of the dots and sign extension bits in order to have an height reduction.

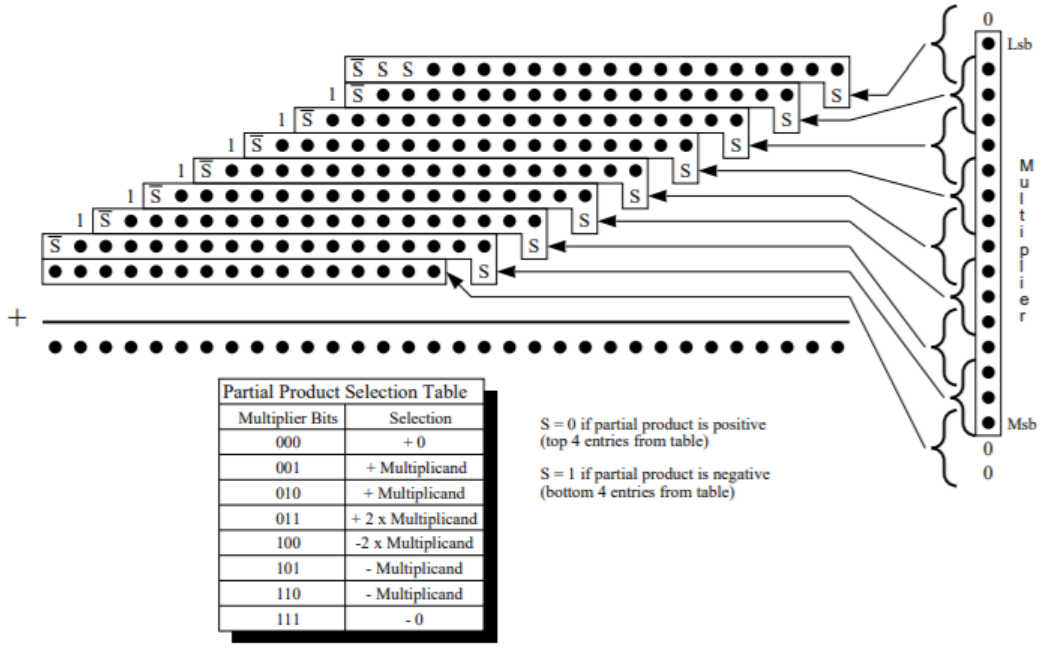


Figure 2.2: Complete 16 bit Booth 2 multiplication with height reduction.

In our case the multiplication is on 24 bits, so the height reduction will bring to a total of 13 row and 48 columns (fig.2.3). As a consequence,by applying the Dadda tree structure, we need six levels to compress 24 operands to two operands, by the means of a total of 264 FAs and 33 HAs.

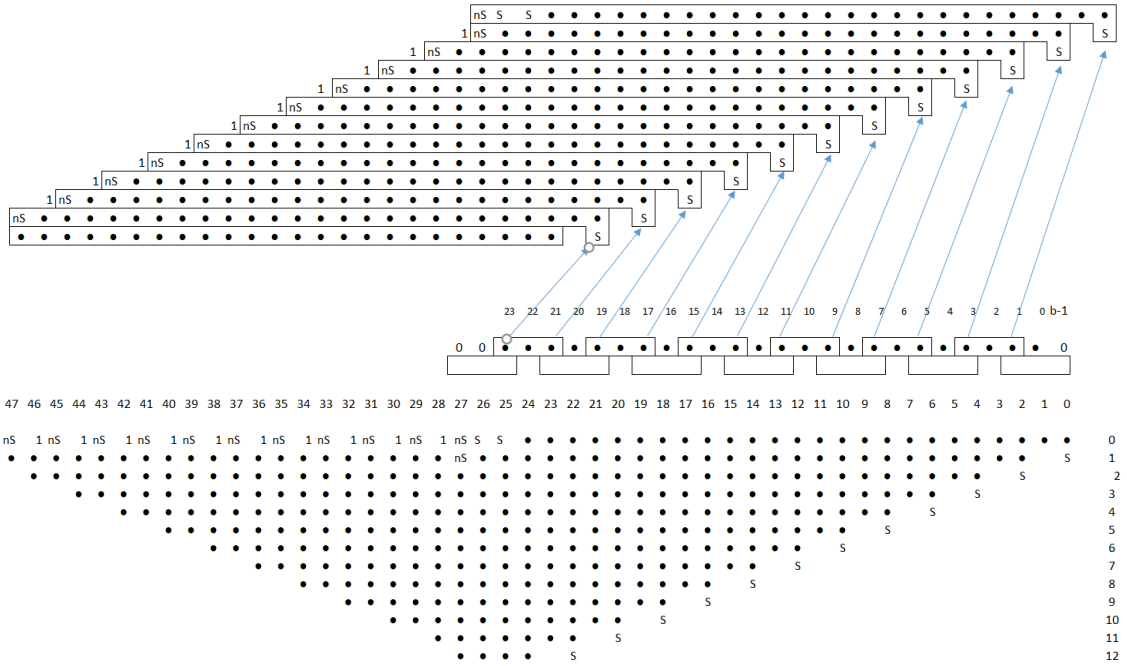

Figure 2.3: Complete 24 bit Booth 2 multiplication with height reduction.

Partial product alignment is shown with dot notation in fig.2.4.

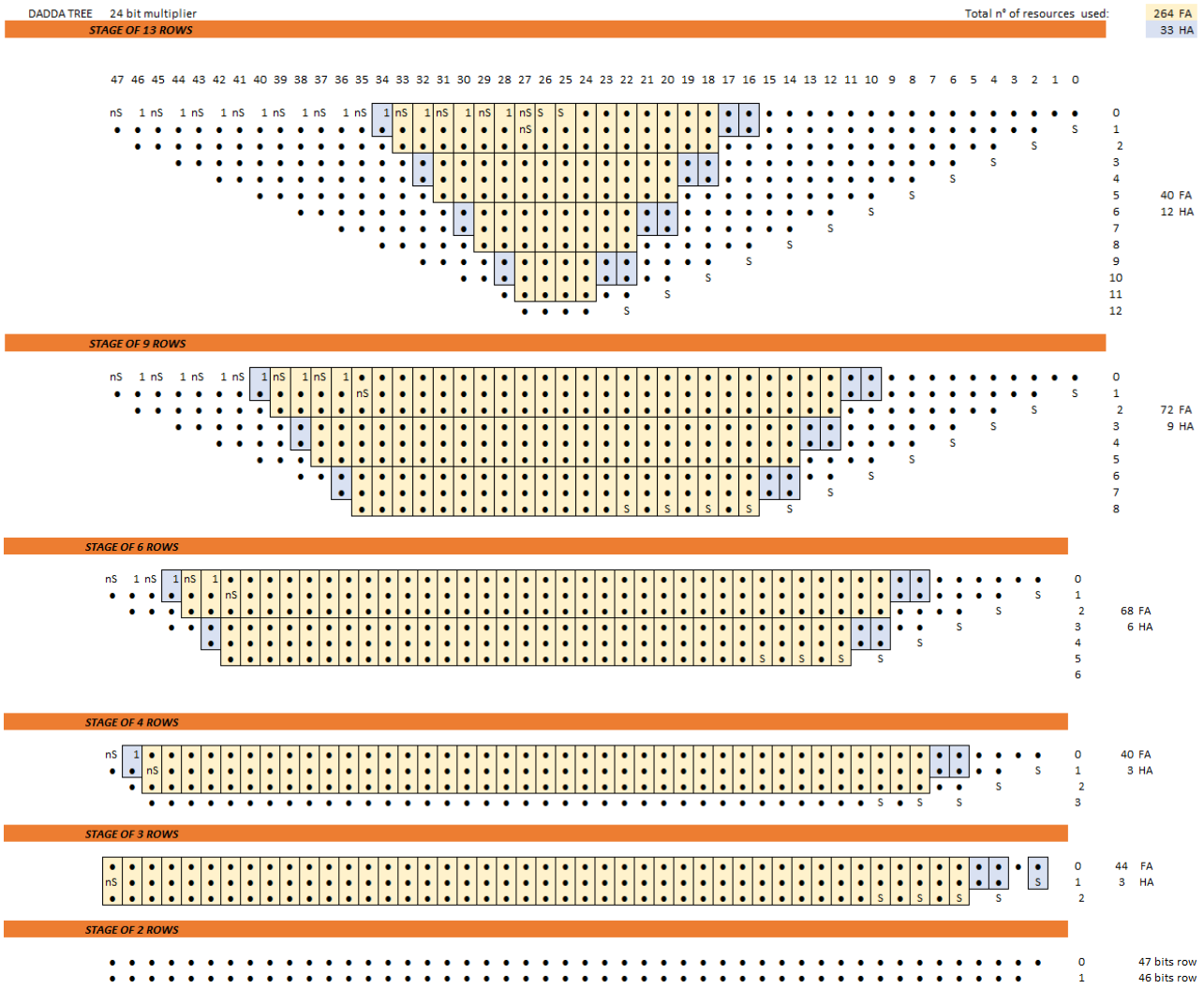


Figure 2.4: Dadda tree with adders assignment.

### 2.2.3 Synthesis

After the verification via simulation of the correct behaviour of the multiplier (Fig. 2.5), the design is synthesized in order to find the maximum operating frequency and its area, showed in table 2.3. Moreover it was decided to do a synthesis of the pipelined FP mul of point 2.1, with the MBE-based multiplier in order to do a comparison of the obtained results.

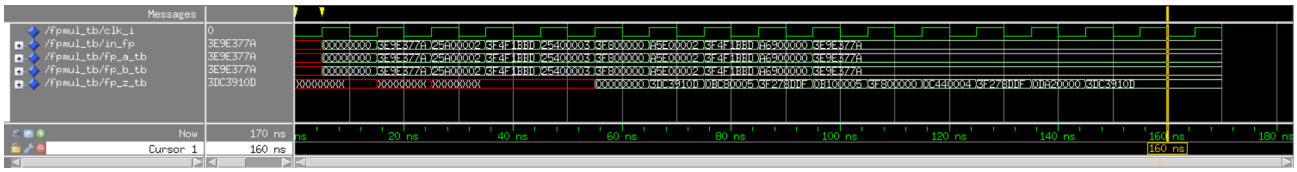


Figure 2.5: Simulation waveforms of the FP mul with MBE

| Minimum period [ns] | Max frequency [MHz] | Total cell area | Power consumption [mW] |
|---|---|---|---|
| 3.94 | 253.8 | 5025.8 | 1.45 |

Table 2.3: Synthesis results

The pipelined mul was firstly compiled without optimization to compare the results with the synthesis, then a register optimization was applied. The types of compiling applied are listed below, with the abbreviations used for the synthesis results:

- Compile + Optimize_registers (C + O)
- Compile_ultra + Optimize_registers (CU + O)

For each case it was found maximum frequency, area and power consumption. The results are shown in the table 2.4:

| Synthesis | Minimum Period [ns] | Frequency [MHz] | Area [$\mu m^2$] | Power consumption [mW] |
|---|---|---|---|---|
| C + O | 0.78 | 1282 | 6751.6 | 10.31 |
| CU + O | 0.81 | 1234 | 6744.7 | 9.64 |

Table 2.4: Synthesis and optimization of the Pipelined FP mul with MBE

## 2.3 Final results

Below is shown a table summarizing all the results obtained in this laboratory (Table 2.5).

| Architecture and synthesis | Frequency [MHz] | Area | Power consump. [µW] |
|---|---|---|---|
| Unflatten FP mul | 223.7 | 3985 | 918.3 |
| Unflatten FP mul with CSA | 223.2 | 4806.1 | 962.8 |
| Unflatten FP mul with PPARCH | 223.7 | 3759.4 | 879.7 |
| Pipelined FP mul (Compile + Optimize) | 1052 | 5547.2 | 6709 |
| Pipelined FP mul (Compile Ultra) | 704 | 4303.6 | 3308 |
| Pipelined FP mul (Compile Ultra + Optimize) | 1176 | 4750 | 6537 |
| FP mul with MBE | 253.8 | 5025.8 | 1451 |
| Pipelined FP mul with MBE (Compile + Optimize) | 1282 | 6751.6 | 10310 |
| Pipelined FP mul with MBE (Compile Ultra + Optimize) | 1234 | 6744.7 | 9644 |

Table 2.5: Synthesis results

Summarizing all the results coming from all the different strategies we can give a glimpse about the trade-off between frequency, area and power consumption. The first strategy adopted was centered in the use of multiplier available on Design Ware, therefore ready-to-use units. The overall performance are satisfying even though the frequency obtained is the lowest between all of them. The key point is the ease of use that allows to have a comparison respect to the own design.

The second implementation is the pipelined multiplier using a set of registers optimized by the Design Compiler. The main advantage is the highest frequency reached, more than five times higher than the first case. The principal drawback is the high power consumption that reaches 6709 uW in the

Compile + Optimize case.

The third method is to use an MBE multiplier instead of a given unit, used in the first method with the Synopsys compiler. The frequency is higher, even if with a slight margin. The power consumption is comparable to the one of the ready-to-use units, but the area increases and it is similar to the pipelined multiplier. It's a good result compared to the first method in terms of maximum operating frequency, but can be improved by the use of a clever synthesis of the pipelined multiplier: the registers optimization brings to an higher frequency with the respect to the standard pipelined FP MUL and area-power data are coherent with the design.