Politecnico di Torino

III Facoltà di Ingegneria

Digital Arithmetic
Integrated Systems Architecture

Master's degree in Electronic Engineering

Authors: group 24

Campanella Andrea, Iacovelli Gianluca, Pala Stefano

Contents

CHAPTER 1

Lab 2: Digital Arithmetic

1.1 Architectures

This first section will examine the various architectures that have been designed and synthesized for the floating point multiplier.

1.1.1 Classic Architecture

The starting point for the design was the 32-bit floating point pipelined multiplier, for which vhdl code was provided. The outline scheme is shown in Figure 1.1, it is divided into 4 main stages:

- Stage 1: The two input floating point numbers are unpacked.
- Stage 2: Multiplication of significands, treated as integers, is carried out.
- Stage 3: The rounding of significands is carried out.
- Stage 4: The calculation of the exponent is carried out and finally the number is again transformed into a floating point through the packaging block.

Studying the VHDL source, it can also be noticed that there are always registers between the stages which sample the signals on the rising edge of the clock. This is the reason why the functional stages correspond to the pipeline ones. Moreover it is possible to observe that in the second stage the type of multiplier used to multiply the significands is not specified, in fact there is a process in which the operation is done with the symbol "*" which leaves the synthesizer the choice of hardware implementation.

Testbench

Before starting with the various hardware modifications, the correct functioning of the supplied circuit was simulated using Modelsim. For this purpose, a testbench was created in verilog with blocks for input generation, DUT and output saving. The input file used is the one provided for both input A and B so the multiplier performs the square of the number in question. The numbers were represented on 8 hexadecimal bits, so conversion to binary was performed before feeding them to the DUT. Finally, by comparing the results obtained with those provided as an example, it was possible to confirm the

Github repository: https://github.com/campandrea/Lab_ISA/

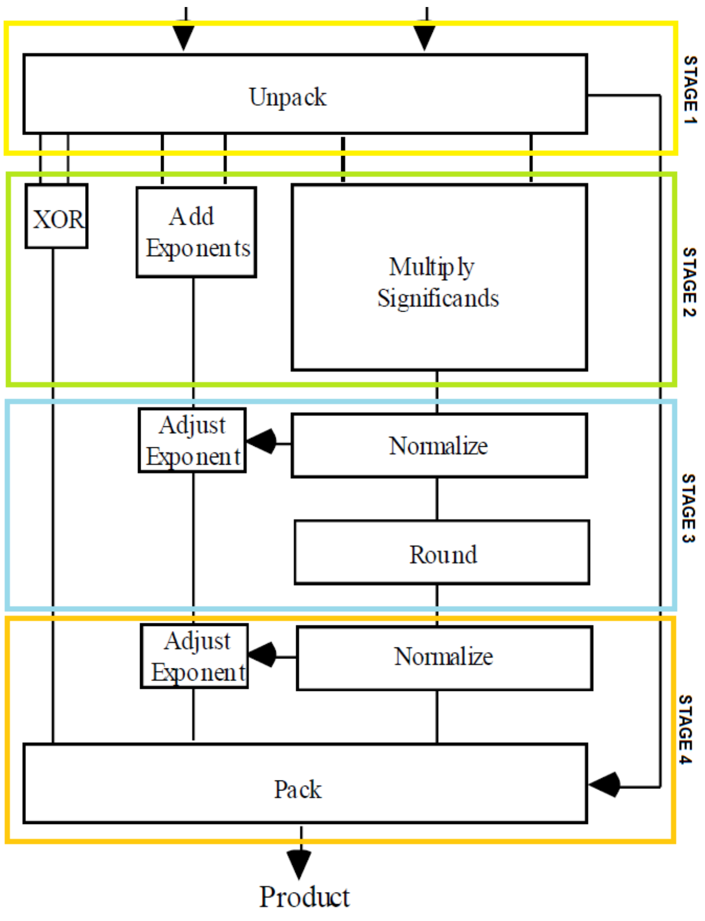

Figure 1.1: Structure of the floating point multiplier

functioning of the circuit. Furthermore, by studying the timing generated by Modelsim, found in Figure 1.2, it can be seen that the latency of the circuit is indeed as assumed, i.e. four clock strokes.

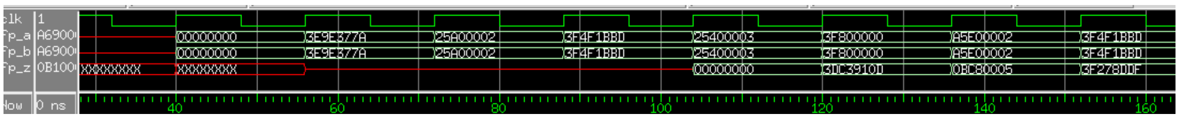

Figure 1.2: Timing of the standard implementation

Afterwards, the VHDL code was modified so that registers were added to the input signals; this correction helps to decouple the timing of the inputs, with possible glitches, from the internal timing of the circuit, thus avoiding unexpected errors. This modification was followed by a second testbench, which is functionally the same as the previous one, and again showed that the device under test worked correctly. As can be seen in Figure 1.3 the only difference with the previous implementation is that the latency of the circuit increases by moving to five clock cycles.

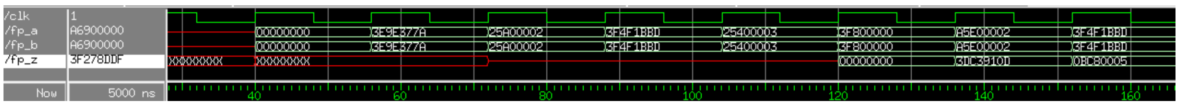

Figure 1.3: Timing of the standard implementation with input registers

1.1.2 Carry Save Adder (CSA)

The Carry Save Adder is a 3 inputs and 2 otputs digital device, where the carry-in terminal has been replaced by a third general input. In fact, this device can be also used to sum in parallel three or plus different binary numbers, without lost of efficiency and excessive growing of area.
The basic single block is shown in the image below:

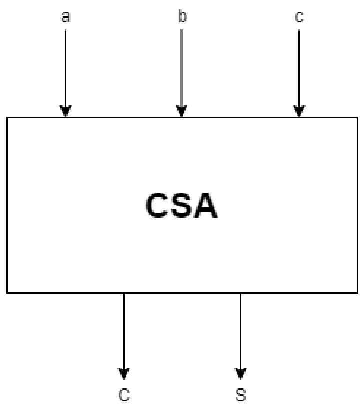

Figure 1.4: Carry Save Adder basic block

In general, the *carry save adder* block has no difference in terms of logic gates if compared to the *full adder*, but they are used in a different way.
The *CSA* allows to compute some addition parts in parallel, and after, the partial results are combined in order to reach the complete summation of the numbers in input. The image below shows an example of the *CSA tree*:

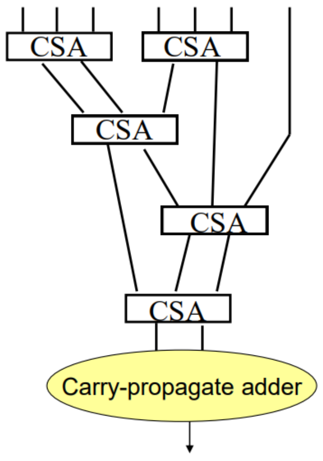

Figure 1.5: Carry Save Adder tree

There are a lot of algorithms that exploit the power of this device, and there are several approaches used to speed up the execution of an addition or a multiplication, the mains are the Wallace and Dadda ones.
The adder present in the second stage of the multiplier was forced to be synthesized as CSA and in order to do that the following commands were used in the synthesis process:

- ungroup -all flatten
- set_implementation DW02_mult/csa [find cell *mult]

1.1.3 Parallel Prefix Architecture (PPARCH)

The *PPARCH* analysed in this lab has been previously improved by using two particular techniques: the *carry look ahead* and the *parallel prefix.* The first one uses a combination of *generate* and *propagate* to compute the carries faster than a *ripple carry adder* architecture. Generate and Propagate are computed by combinating the binary input signals:

- Generate $G_i = A_i \cdot B_i$
- Propagate $P_i = A_i \oplus B_i$

By using these signals is possible to compute the carries and the partial sums:

$$C_{i+1} = G_i + P_i \cdot C_i$$
$$S_i = P_i \oplus C_i$$

In particular, it's possible to unroll the carries equation, in order to speed up the computation:
$C_0 = C_0$
$C_1 = G_0 + P_0 \cdot C_0$
$C_2 = G_1 + P_1 \cdot C_1 = G_1 + G_0 \cdot P_1 + P_1 \cdot P_0 \cdot C_0$
and in general:
$C_i = G_{i-1} + P_{i-1} \cdot C_{i-1}$
All these computations require a lot of hardware to be computed and there is an issue called a *Parallel Prefix* problem: the generate-propagate network can be very long and this can develop problem in terms of the critical path. In fact, a several algorithm has been invented in order to improve the performance of this newtork, for example: Kogge-Stone , Ladner-Fishner, Brent-Kung. Synopsys can arbitrarily use one these optimizations in order to minimize the hardware required to develop the generate-propagate carry generation network.
The adder present in the second stage of the multiplier was forced to be synthesized as PPARCH and in order to do that the following commands were used in the synthesis process:

- ungroup -all flatten
- set_implementation DW02_mult/pparch [find cell *mult]

1.1.4 Fine grain pipelining

The next aim is to modify the VHDL file by inserting a *pipeline stage* in stage 2 output, as shown in the following image:

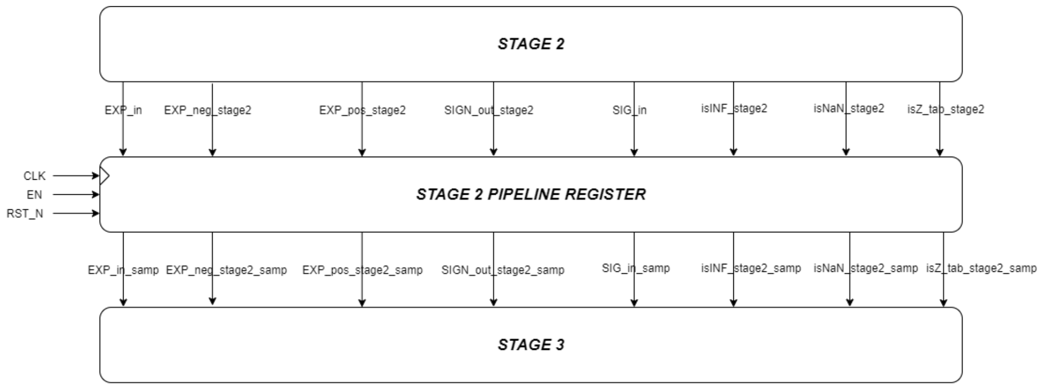

Figure 1.6: Fine grain pipelining block diagram

The output register has been implemented in the VHDL file by using several registers in order to ensure the correct behavior and timing. After the VHDL manipulation, two different synthesis processes were made by using different commands per time: in the first run, the compile command was launched after *optimize_registers* command and in the second run *compile_ultra* was used rather than *optimize_registers* and *compile*. The Synthetizer elaborates the structures in a different way, by using some algorithms instead of other ones, according to the choosen command.

Testbench

To verify the correct behaviour, a Modelsim simulation was launched prior to the synthesis, which again showed proper functioning. As can be seen in Figure 1.7 the latency of the circuit is correctly increased by changing to six clock cycles.

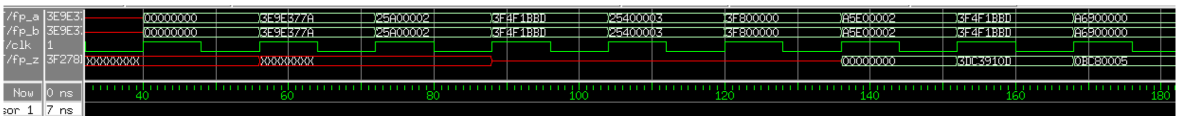

Figure 1.7: Timing of the fine grain pipelining implementation

1.1.5 Modified Booth Encoding (MBE)

The Modified Booth Encoding Multiplier is an extension of the Radix-2 approach, where a Radix-4 approach is used in order to produce half partial products. In general, considering **a** and **b** as two unsigned numbers in the following form:

$$\mathbf{a} = \sum_{i=0}^{n-1} a_i 2^i \qquad \mathbf{b} = \sum_{i=0}^{n-1} b_i 2^i$$

The product **c** is obtained:

$$\mathbf{c} = \sum_{i=0}^{n-1} \sum_{i=0}^{n-1} a_i b_i 2^{i+j}$$

Radix-2 approach produces **n** partial products in the form:

$$p_j = \begin{cases} 0 & \text{if } \overline{b}_j \\ a & \text{if } b_j \end{cases}$$

On the other hand MBE produces only **n/2** partial products because more bits are analyzed simultaneously. The multiplier is divided in 3-bit slices (with $b_{-1} = 0$), where two consecutive slices feature a 1-bit overlap. Then, each triplet of bits is exploited to encode the multiplicand according to the subsection 1.1.5. The expression to evaluate p_j is $p_j = (b_{2j+1} \oplus q_j) + b_{2j+1}$ where

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

Table 1.1: Modified Booth Encoding

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

A total of 16 combinatorial blocks were used to evaluate all partial products that have to be summed. An overview of the structure with a dot view can be seen in Figure 1.8. A Dadda Tree was used to

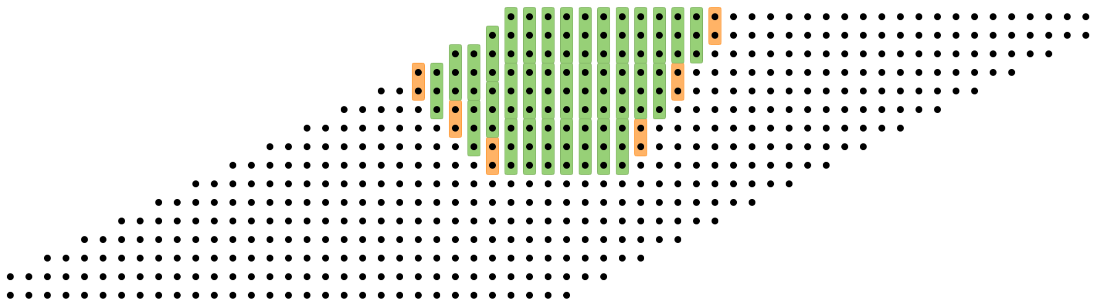

Figure 1.8: Partial products representation

cover the bits in order to use an optimal number of stages and thus high speed in performing the operations. The approach is to reduce the height at each stage until a predetermined number of bits is reached using half-adders and full-adders. The heights follow the sequence:

$$d_1 = 2 \qquad d_{j+1} = \lfloor (1.5 d_j) \rfloor$$

The initial height of the structure is 16, so the sequence will be:

$$d_1 = 2,\ d_2 = 3,\ d_3 = 4,\ d_4 = 6,\ d_5 = 9,\ d_6 = 13,\ d_7 = 16$$

Since the next step has a height of 13, all columns that have a higher number of bits must be reduced. Starting from the right, on the first column that has 14 bits a half-adder has been inserted that starting from two bits generates a new bit on the same column and a carry bit on the next column. The adjacent column that still has 14 bits, considering the carry from the previous half-adder, requires a full-adder which, from three bits, generates a sum bit on the same column and a carry bit on the next column. This reasoning has been repeated on the whole structure and the final result can be observed in Figure 1.8 where 6 half-adders (in orange) and 33 full-adders (in green) have been used. The same mechanism is applied to each stage to reduce the number of bits while respecting the sequence. In Figure 1.9 it is possible to observe the coverage of each stage up to a height of two bits, where thanks to a adder the final result can be computed. The number of half-adders and full-adders for each stage

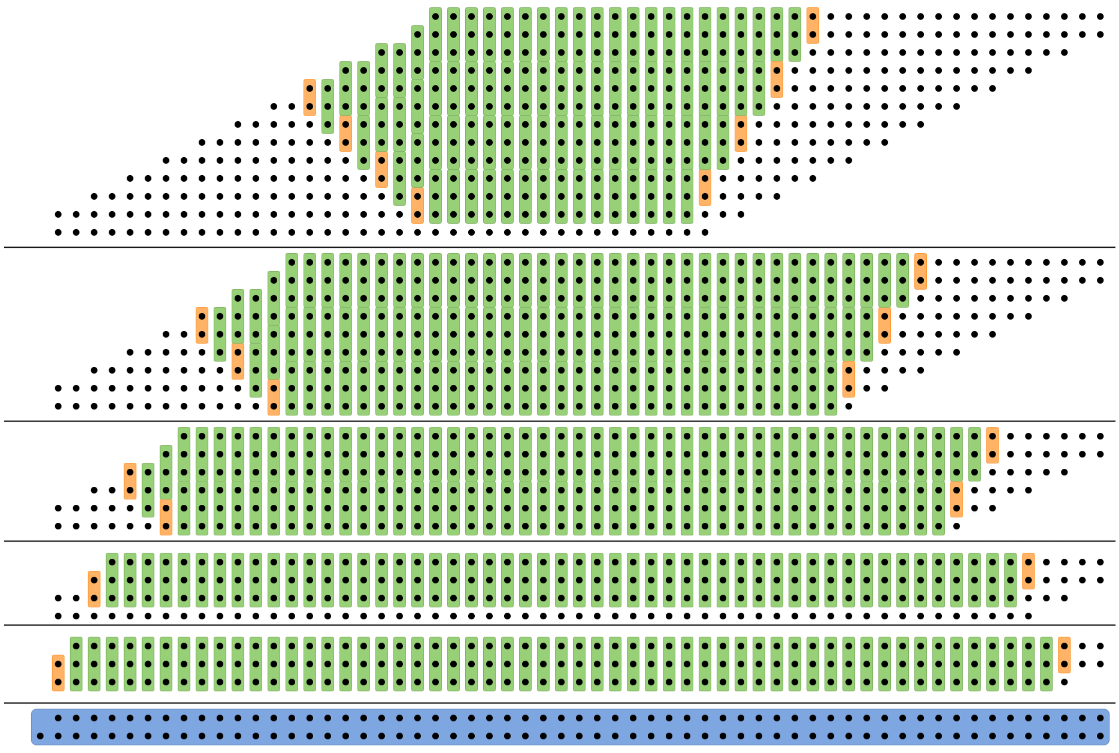

Figure 1.9: Dadda Tree

are summarized in Figure 1.1.5. The last stage can be implemented in different ways; to get an idea of the final result it has been considered as a ripple-carry adder. Once the structure was defined a

stage	HA	FA
1	6	33
2	8	84
3	6	105
4	4	90
5	2	51
6	2	55
7	2	58
Tot	30	476

Table 1.2: Number of adders used

program written in C++ [2] , was used to generate the vhdl code related to the MBE multiplier.

[2]Source: here

Testbench

The correct functioning of the circuit was verified using the previously written testbench. Comparing the results obtained with those expected has confirmed the correctness of the implementation and when looking at the timing no differences were noticed compared to Figure 1.3.

1.2 Synthesis

For all the architectures described, synthesis was carried out with the Synopsys software in order to find the maximum operating frequency and the corresponding area. To find these results, the clock period was first set to zero, and then this value was increased until slack was null.

1.2.1 Results

The results obtained from the various syntheses are shown in Table 1.3.

Architercture	Minimum period (ns)	Area (µm^2)
standard	1.56	4047.7
carry save adder	4.28	4851.6
parallel prefix adder	1.45	4088.4
fine grain pipelining	0.79	4924.7
fine grain pipelining (compile_ultra)	1.52	4188.2
modified Booth encoding	1.94	5360.4
modified Booth encoding (compile_ultra)	1.64	5327.7

Table 1.3: Syntesis Results

A first observation that can be made is about the standard architecture: the results show values not so far from the other architectures, a sign that leaving freedom to the synthesiser is not always a bad choice.
The combined solution, on the other hand, in which the implementation of the adder, which is present in the multiplier of the second stage, is imposed, can lead to totally different and sometimes counter-productive results; in fact, by choosing a carry save adder the minimum period increases a lot, about 175%, and with it the area, while choosing a parallel-prefix adder we have a slight improvement in frequency, about 8%, keeping the area about the same as the standard implementation.
The solution with fine grain pipelining is the one with the smallest clock period. However, two completely different results are obtained: by forcing retiming (command `optimize_register`) with consequent classical compilation, or by using only the command `compile_ultra`, which enables various optimization mechanisms such as automatic ungrouping to optimize the timing of the circuit. In the first case a marked reduction in the clock period is observed, about 50%, at the expense of an increase in the area of 22%. In the second case there is only a 3% reduction in the clock period and also a slight increase in the area, but less than in the previous case. It could then be concluded that, in this particular situation it is not convenient to use this technique unless there are strict constraints on the area to be occupied. On the other hand, it is very efficient in terms of performance to add registers and force the synthesiser to perform retiming.
Finally, implementing the second stage multiplier with the modified Booth Encoding technique, no special advantage is noticed, on the contrary, the clock period increases by 24% and the area by 32%, a result probably due to the higher combinatorial path for the addition of the Dadda tree. Slightly better results can be obtained by using the command `compile_ultra`.

In conclusion, it can be said that to improve circuit performance the best strategy is to let the synthesiser choose the implementation of the multiplier and add a registers stage to give more margin in the application of retiming at the cost of paying in terms of area and latency. If this solution is not applicable, then implementing the multiplier with a parallel prefix adder brings better advantages than implementing it with a carry save adder or modified Booth encoding multiplier in terms of both area and maximum operating frequency.