



Politecnico di Torino

Integrated Systems Architectures

Digital Arithmetic Laboratory 2 report

GROUP13

Naouras Latiri
(287376)

Zeineb Ben Othmen
(287874)

Hugo Bouisson
(287419)

December 19, 2021

Contents

1	Assignment	2
1.1	Digital arithmetic and logic synthesis	2
1.1.1	Floating point implementation	2
1.1.2	Fine-grain Pipelining and optimization	4
1.1.3	MBE multiplier for unsigned data on 32 bit	5
1.1.4	Dadda tree	5
2	Conclusion	8

ABSTRACT - This document presents a study of digital arithmetic circuits. During this laboratory, Synopsys Design Compiler has been forced to use specified Design Ware adders and multipliers. Then, timing, resources and area reports have been generated and compared.

1 Assignment

1.1 Digital arithmetic and logic synthesis

1.1.1 Floating point implementation

The floating point multiplier implementation is a 4 stage pipelined multiplier with 32 bits operands. In order to verify its correct behavior, we have written a testbench. The inputs were taken from fpsamples.hex and then we compared the results we have obtained with the file prod.hex which contains the correct results.

Then, we have modified the code to add 2 registers in input to store the 2 operands and again we have tested it with the same testbench to confirm the correct behavior. The advantage is that now we have sequential inputs. So as a result the latency is expected to increase by one clock cycle due to the presence of the additional registers for each input operands.

Testbench :

The first step consists in creating a testbench to verify the correctness of the floating point implementation. To do so, we used the following modules :

- DUT : program under test
- DATA MAKER : read the samples from the file fpsamples.hex
- DATA SINK : write all the result generated by the DUT inside the file fpreults.hex
- CLOCK GENERATOR : generate the clock signal for all the modules

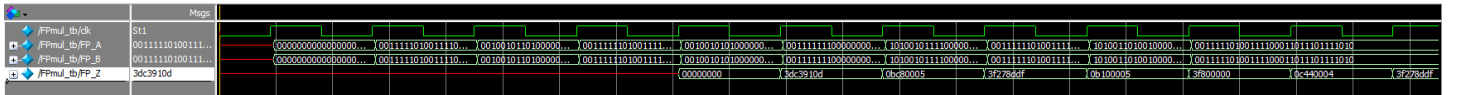


Figure 1.1: Result with file *FPMul.vhd*, floating point multiplication

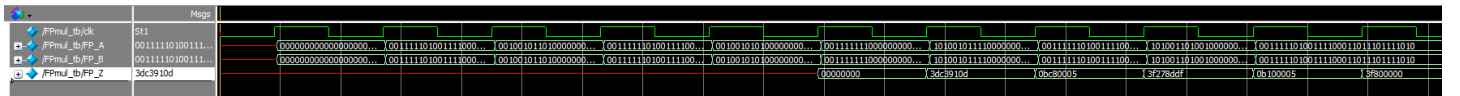


Figure 1.2: Result with file *FPMul_with_reg_IN.vhd*, floating point multiplication with added registers to the input

As we can see, we get the same result for both simulations, but with the input register we have to wait one more clock cycle to get the result.

Synthesis :

During this step, several synthesis were performed at the aim to better analyse the differences between various implementation and constraints. Three different synthesis scripts were elaborated:

- *syn_first.tcl*
- *syn_csa.tcl*
- *syn_pparch.tcl*

In the first one, we have forced Design Compiler to flatten the hierarchy and to synthesize the multiplier leaving all the implementation choices to the Synopsis tool in order to find the maximum frequency and the area. In the second script, we have forced Design Compiler to flatten the hierarchy and to implement the significands multiplier in Stage2 as a CSA multiplier. In order to perform this operation, these commands were generated:

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

And in the last script, we have forced Design Compiler to flatten the hierarchy and to implement the significands multiplier in stage2 as a PPARCH multiplier. The commands generated here:

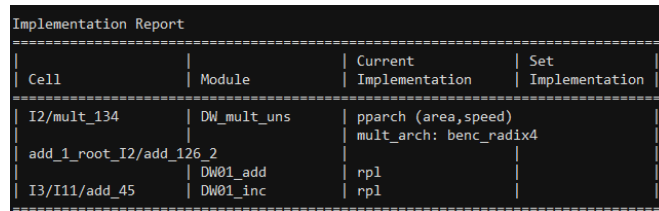
- *ungroup -all -flatten*
- *set_implementation DW02_mult/pparch [find cell *mult*]*
- *compile*

Logic Synthesis :

As it can be seen in table 1, the performance is significant when the synthesizer is allowed to choose the implementations by itself. When referring to the resources report, the synthesizer has chosen to implement the parallel prefix multiplier. When synthesizing the PPARCH, the area is decreased, which leads to conclude that at the beginning the synthesizer has optimized the area and the time while, when forcing it to implement it, the optimization is performed in terms of only the area.

	Design Compiler choice	CSA	PPARCH
maximum frequency [MHz]	714.2857	217.3913	588.2353
area [μm^2]	4106.242	4807.684	3962.069

Table 1: Post synthesis report



```

Implementation Report
=====
| Cell          | Module      | Current Implementation | Set Implementation |
|-----|-----|-----|-----|
| I2/mult_134   | DW_mult_uns | pparch (area,speed)    |                     |
|               |               | mult_arch: benc_radix4 |                     |
| add_1_root_I2/add_126_2 | DW01_add    | rpl                     |                     |
| I3/I11/add_45 | DW01_inc    | rpl                     |                     |
=====

```

Figure 1.3: Resources report synthesizer

1.1.2 Fine-grain Pipelining and optimization

In this part, we have added a register at the output of the significands multiplier and several registers such that the timing of the `fpstage2struct.vhd` is correct. Then, using Design Compiler, we have to flatten again the hierarchy and synthesis in order to find the maximum frequency and the area.

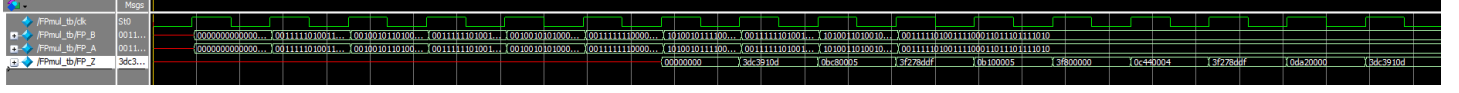


Figure 1.4: Result with file `FPmul_with_reg-OUT.vhd`, floating point multiplication with added registers in output

As we can see, we get the same result as before, but now, with added register in output (and added registers such that the timing for the whole stage2 is correct) we have to wait 2 additional clock cycles with respect to the classic multiplier.

Logic Synthesis :

During this step, we've forced the Design Compiler to flatten then we've generated the commands `optimize_register` and `compile_ultra`.

	compile + Optimize register	Compile_ultra
maximum frequency [MHz]	1333.33	1111.11
area [μm^2]	3900.091	3661.223

Table 2: Post synthesis report

The table 2 shows that the synthesis with `optimize_registers` is quite twice the one performed with just `compile` in the previous section, which means that the timing has been optimized. However, the area has increased with respect to the the synthesis with `compile_ultra`.

1.1.3 MBE multiplier for unsigned data on 32 bit

Now, we have tried to implement a 32 bit 2-multiplicands unsigned multiplier based on Dadda tree and a modified booth encoding. This type of encoding uses a radix-4 approach. So with A the multiplicand and B the multiplier, each partial product is generated taking 3 bits slices of B with one bit overlapped for 2 consecutive slices. In this way B is extended on 35 bits, adding a 0 in the position -1, 33 and 34. Then each slice encodes the multiplicand according to the following expression:

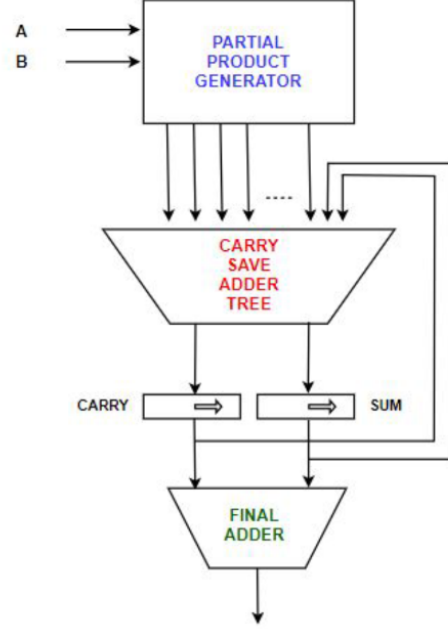


Figure 1.5: Multiplier

1.1.4 Dadda tree

In the Dadda tree we use some full adders (FA) and half adders (HA) to reduce the sum of all partial products. In the following figure we have the result of partial product according to the sign, the weight and the bits of B :

$$d_{j+1} = \left\lfloor \frac{3}{2} d_j \right\rfloor ; j \mid d_j \geq n \ \& \ d_{j-1} < n$$

The Dadda allocation is as late as possible (ALAP), meaning that for each level we need to allocate the minimum number of FA and HA. Each level has a specific maximum height :

$B_{2i+1}B_{2i}B_{2i-1}$	Partial product i
000	0
001	+A
010	+A
011	+2A
100	-2A
101	-A
110	-A
111	0

Finally we get the following height tree :

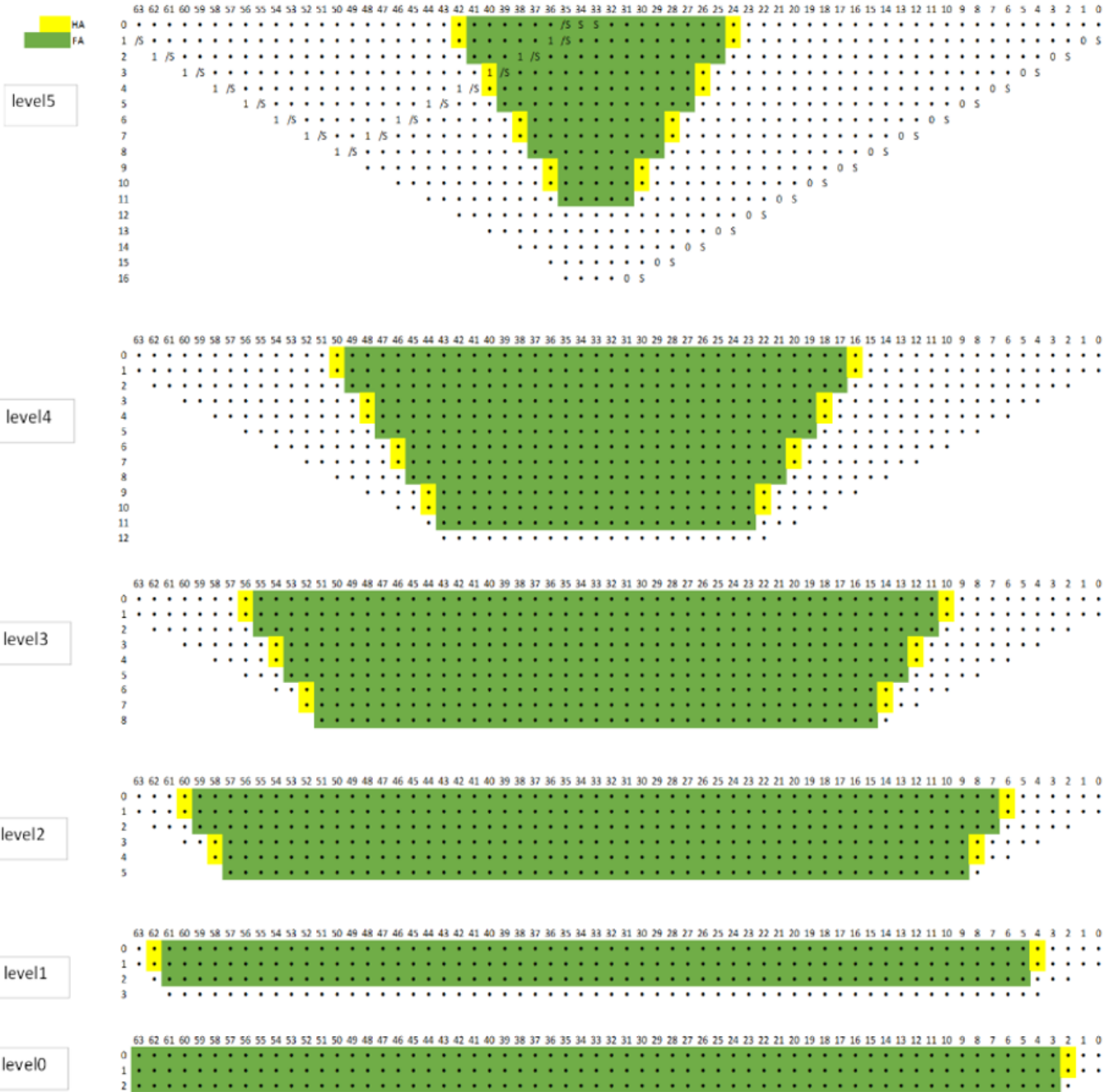


Figure 1.6

In order to test the implementation we have modified the stage 2 by adding the MBE code into it. Then using the same testbench we have performed the test of the architecture.

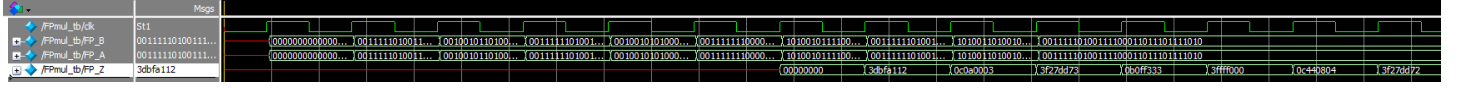


Figure 1.7: Result with file *FPmul_with_reg_IN_MBE.vhd*, floating point multiplication

Logic Synthesis :

Finally, we've performed the synthesis of the *MBE* in order to retrieve the maximum frequency and area.

In the table 3, we have almost summarized all the results, where *Design Compiler Choice* (DCC), *CSA* and *PPARCH* have been chosen for the multiplier with input registers; *Compile + optimize_registers* and *Compile_ultra* for the implementation with output registers and then, *Compile* for the MBE implementation.

Overall, it's remarkable that the best performance is obtained with the synthesis using the command *optimize_register*. As it can be noticed, the synthesis implementing *CSA* witnesses the worst case in terms of timing, with frequency equal to 217.4 MHz. On the other hand, the *MBE* has the worst area with value 6322.3 μm^2 .

	DCC	CSA	PPARCH	C+OR	Compile_ultra	MBE
maximum frequency [MHz]	714.2857	217.3913	588.2353	1333.33	1111.11	238.09.
area [μm^2]	4106.242	4807.684	3962.069	3900.091	3661.223	6322.288

Table 3: Post synthesis report

2 Conclusion

During this laboratory we have implemented different architectures in order to create a floating point multiplier using pipelined implementation. Through the synthesizer, we have seen that depending on the architecture (CSA, PPARCH) or command (`compile`, `compile-ultra`, `optimize_registers + compile`) executed we get advantages in term of performances or in area.

Referring to the results elaborated above, it seems that the best performance is obtained when the synthesizer is free to choose the implementation using the command *compile*. The *optimize_registers* command contributes to achieve the best performance in terms of period, but not the area.