Politecnico di Torino

Corso di Laurea Magistrale in
INGEGNERIA ELETTRONICA

INTEGRATED SYSTEMS ARCHITECTURE

Lab Report 2

Professors:
Prof. Guido Masera
Prof. Maurizio Martina

Students:
Christian Vespo 292674
Claudio Raccomandato 290190
Pietro Romeo 269010

Anno Accademico 2021/2022

Contents

1 LAB 2 : Digital arithmetic Assignment **2**

2 Digital Arithmetic and Logic synthesys **2**
2.1 Modelsim Verification 2
2.2 Performance Analisys with Design Compiler 3

3 Fine Grained pipeline and optimization **4**

4 MBE multiplier Architecture **6**
4.1 Dadda Tree 6
4.2 Partial Product Generator 8
4.3 PP selectors 9
4.4 Booth Recoding Units 11
4.5 Modelsim Verification 12

5 Architectures comparison **13**

1 LAB 2 : Digital arithmetic Assignment

The goal of this Lab is to analyze and understand a floating point multiplier, and ultimately design an unsigned multiplier, one of the fundamental blocks of a Floating point multiplier.

2 Digital Arithmetic and Logic synthesys

In this section we'll seek to understand how the provided Floating Point Multiplier works, and how its second stage impacts the performance. As a first step, Quartus Modelsim was used in order to visualize the architecture. The result can be seen in Figure 1.

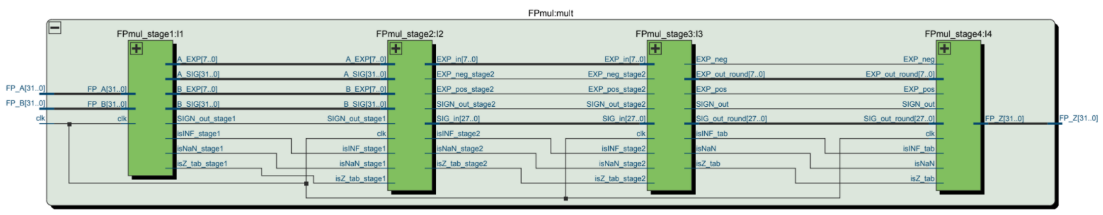


Figure 1: Full FP multiplier block scheme

2.1 Modelsim Verification

In order to verify the proper behavior of the Floating Point multiplier, a testbench was created. Also, a new "wrapped" top level entity was made to ease the process. The testbench is organized in a similar way as that of Lab 1, meaning it is comprised of the testbench itself and a Clock Generator, a Data Maker and a Data Sink. This way, the output results are conveniently stored in a text file. As shown in Figure 2, the multiplier works as intended, since the obtained results match the expected ones.

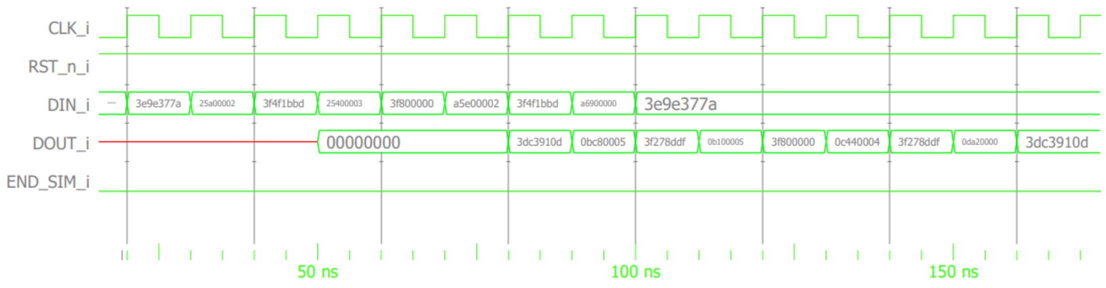


Figure 2: FPmul testbench results on ModelSim wave window

2.2 Performance Analisys with Design Compiler

After the model verification, performance is evaluated with Synopsys Design Compiler. The first evaluation is computed without changes, while for the second and the third the second stage was forced to use specific adders, respectively CSA and PParch. The most relevant results are reported in Table 1.

	Max Frequency [MHz]	Area [μm^2]
Default	641	4120
CSA	230	4905
PParch	641	4153

Table 1: Performance comparison

Max frequency was obtained using the same method as Lab 1, namely through a few iterations. The reported area refers to the architecture at maximum frequency. It is quite clear from the table that the default architecture and the one using PParch display the same results. With the *report_resources* command, it can be verified that in fact the default architecture employs PParch.

3 Fine Grained pipeline and optimization

In order to understand how a pipeline stage would affect the architecture's performance, a "wrapped" second stage was created in VHDL, including both the second stage and the output registers and flip-flops. The structure obtained this way is represented in Figure 3.

Through the *optimize_registers* command in Design compiler, retiming was then performed in order to place these register in a way that would reduce the critical path as much as possible. Other optimizations have been obtained with the use of the *compile_ultra* command. Area and maximum frequency obtained with such optimizations have been reported in Table 2.

	Max Frequency [MHz]	Area [μm^2]
optimize_registers	1176	4751
compile_ultra	613	4100
optimize_registers and compile_ultra	1315	5317

Table 2: Performance with retiming

Performing retiming yielded the expected result: the critical path delay was roughly halved. *compile_ultra* on the other hand, did not perform retiming, meaning the added register stage did not necessarily improve the architecture. It must be noted that *compile_ultra*, in an effort to reduce the area, has implemented a Radix 8 multiplier (while the optimize_registers employs Radix 4). This choice allowed to reduce the size of the multiplier, at the cost of inserting adders or subtracters to calculate partial products. When using both commands, we obtain the highest maximum frequency, but also the highest area.

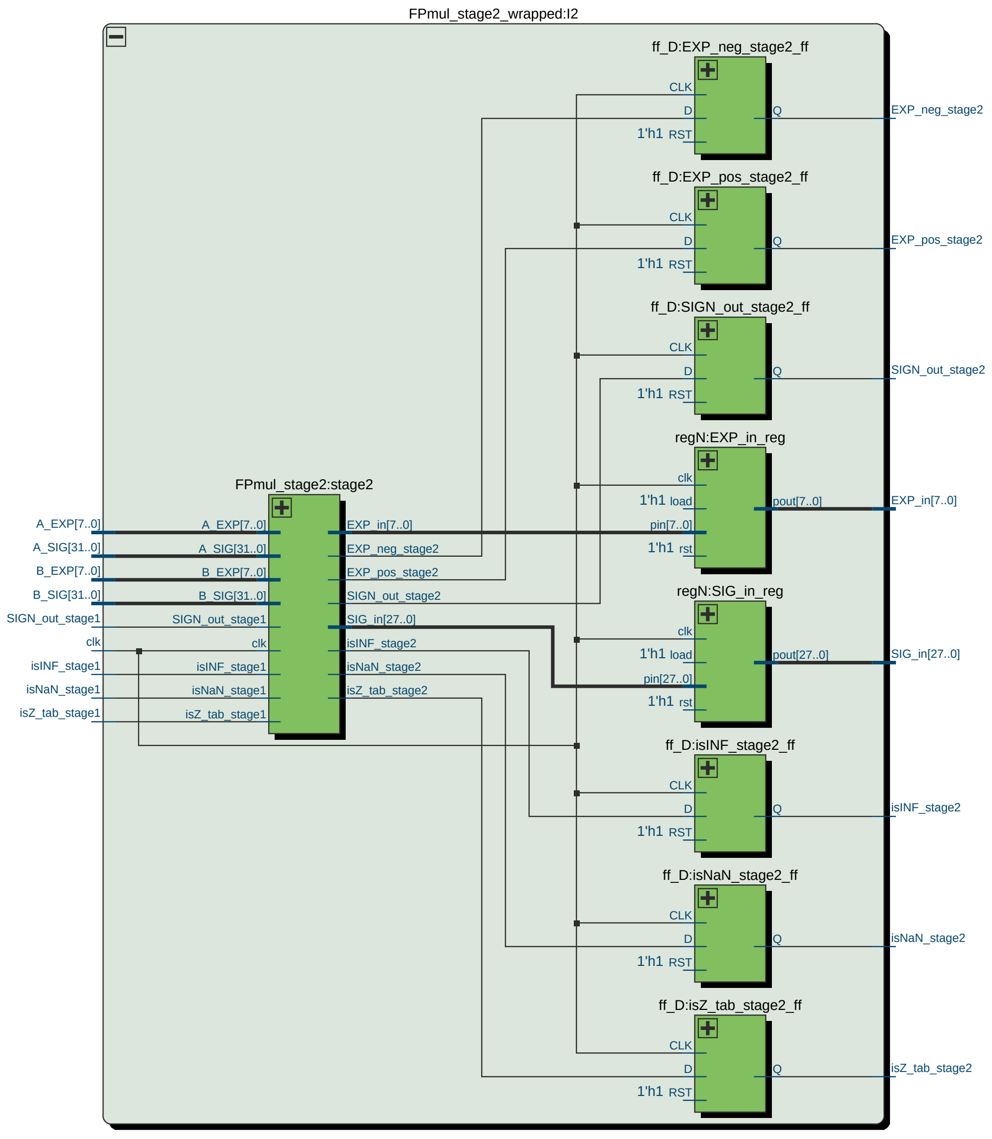


Figure 3: Second stage with added registers

4 MBE multiplier Architecture

Now that the architecture has been fully understood and the impact of the second stage has been evaluated, the unsigned multiplier architecture was developed according to the specifications.

Given the constraint of not using adders/subtracters to generate partial products, the Radix choice was limited to either 2 or 4. A higher radix would in fact have required to calculate multiples of the multiplicand which would not be possible to achieve with shifts.

Also, a full tree architecture was chosen in order not to modify the timing of the Floating Point Multiplier. Given the gargantuan size of the Full Tree, Radix 4 was chosen so as to work with twelve partial products instead of the twenty four that would have resulted from a Radix 2 implementation.

Figure 4 represents the block diagram that was implemented.

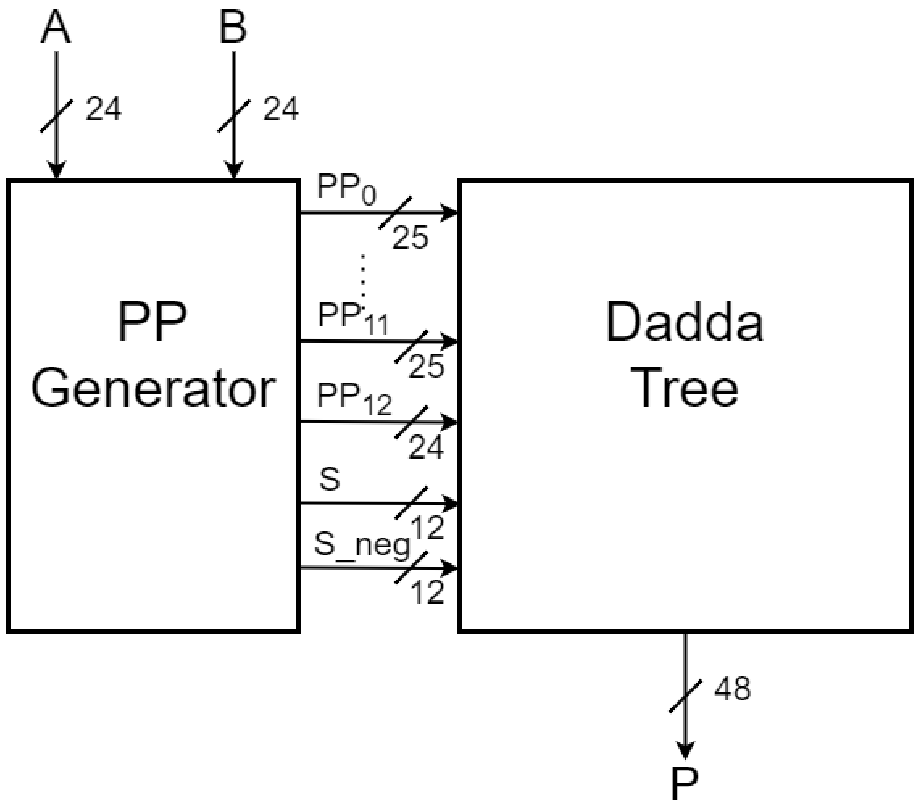


Figure 4: Full Tree MBE architecture

4.1 Dadda Tree

The sign extension bits have been optimized by replacing them with a single 1 followed by $\overline{S}$, as explained in the sign_extension_booth_multiplier_Stanford.pdf file. For the first partial product, however, the bits for the sign extension have been replaced with a 1 followed by the sum of 1 and $\overline{S}$, whose equivalent circuit has been simplified with a half adder and a nor gate. Also, given the presence of '1's in the Tree, the Half adders that took them as one of their inputs have been reduced to an inverter and a wire. The Tree is fully represented in Figure 5. Zooming in allows to distinguish blocks and inputs. Only the inputs essential to understand the pattern have been represented.

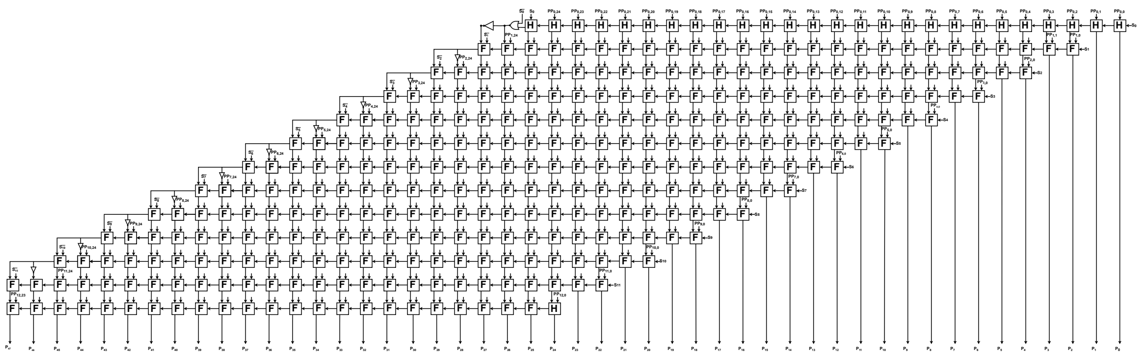


Figure 5: Full Dadda Tree

Moreover the implementation of each half adder is shown in Figure 6. In this way two NOR gates have been used instead of an XOR, using the signal of COUT.

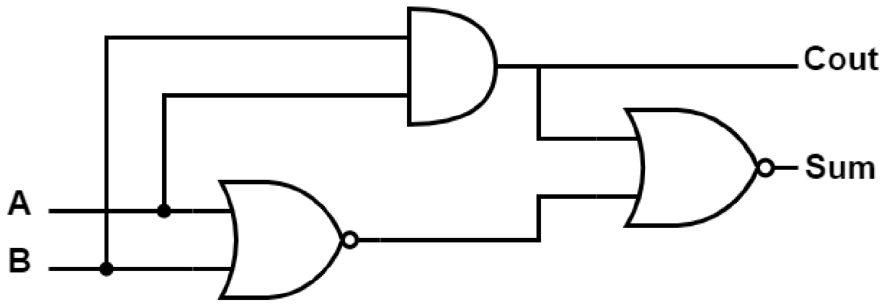


Figure 6: Optimized Half adder

Instead the representation of the Full Adders is shown in Figure 7.

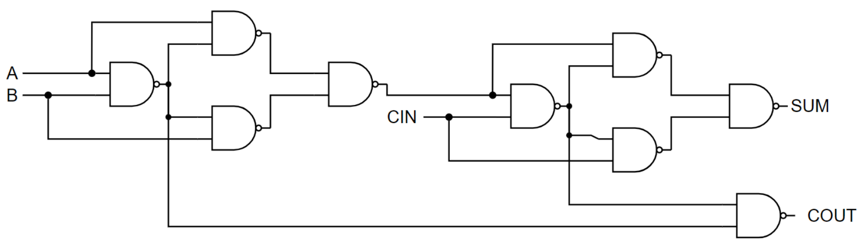


Figure 7: Full adder

In this representation using only NAND gates, it is possible to use internal signals for the computation of the outputs, in this way fewer logic gates are implemented.

4.2 Partial Product Generator

Partial product generation has been implemented in two stages, as represented in Figure 8: the first stage being comprised of thirteen BRUs (Booth Recoding Units) and the second being comprised of thirteen Partial Product selectors. All the parts have been realized in NAND and NOR gates, in order to be able to reduce the number of gates as much as possible, and thus reduce the total area.

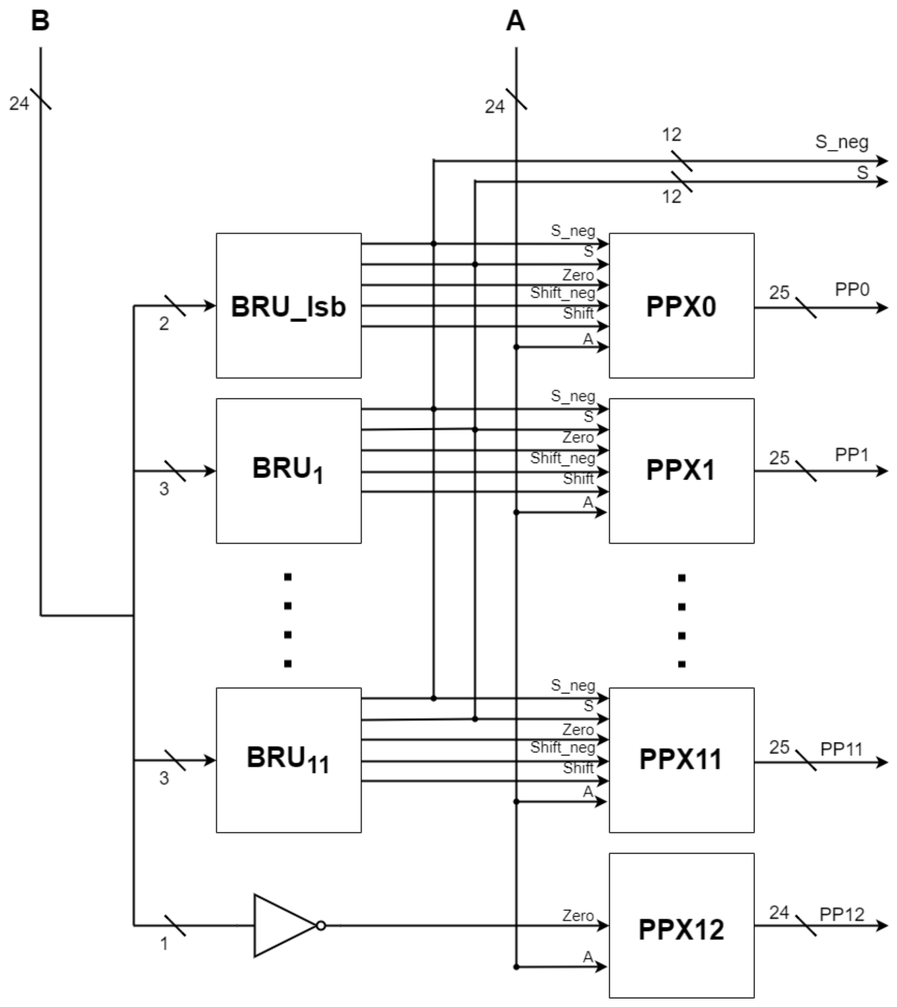


Figure 8: PP generator

4.3 PP selectors

Given the high amount of components required to realize the PP generator, several optimizations have been implemented in order to reduce the number of gates. Firstly, the multiplexers that are traditionally used to select partial products have been simplified in account of them having three inputs (either zero, A or 2A) instead of four. Since the thirteenth partial product selector only receives Zero as input signal, PPX12 was further simplified.

The first twelve Partial Product selectors are composed by two blocks: a block is dedicated to the LSB and for the other bits except the MSB, it is represented in Figure 9, with the corresponding representation in logic gates in Figure 10. The second block is dedicated to the MSB and represented in Figure 11.

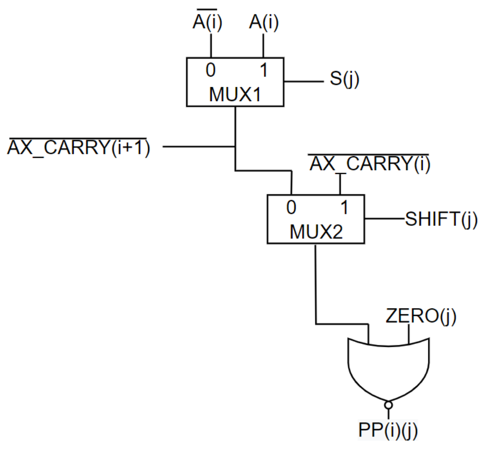


Figure 9: PP selector for LSB and the other bits except the MSB

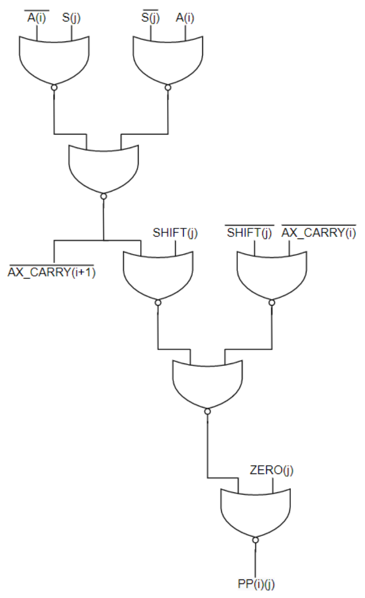


Figure 10: PP selector for LSB and the other bits except the MSB with logic gates

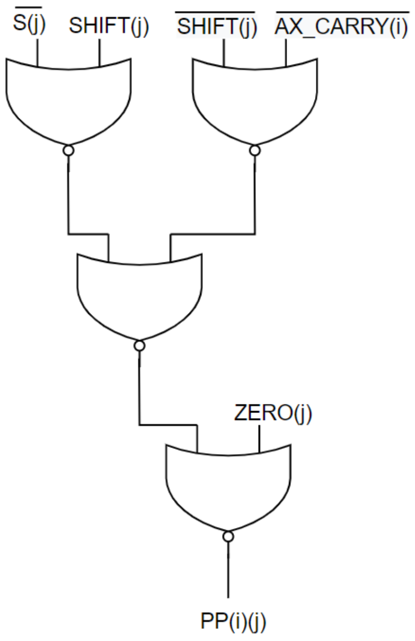


Figure 11: MSB of PP selector

Instead for the last Partial Product selector there is a only block, represented in Figure 12.

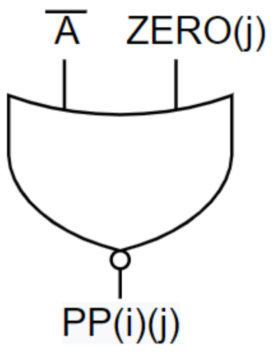


Figure 12: LSB of PP selector

4.4 Booth Recoding Units

The BRUs were simplified as well, exploiting the few Don't Cares in their truth tables. Also, the two BRUs that receive the most significant bits and the two least significant bits have been further simplified, since they behave like a BRU with fixed zeros at some of the inputs. Lastly, it must be added that the BRUs provide two inverted signals (Shift_neg and S_neg). These signals could have been generated locally where needed, but it was preferred to generate them only once to reduce the number of inverters. The behavior of BRU1 is reported in table 3. The truth tables of the BRUs at the most and least significant bits can be derived directly from the same table. Their actual RTL implementation is represented in Figure 13.

B(3)	B(2)	B(1)	S	Shift	Zero	PP
0	0	0	0	-	1	0
0	0	1	0	0	0	+A
0	1	0	0	0	0	+A
0	1	1	0	1	0	+2A
1	0	0	1	1	0	-2A
1	0	1	1	0	0	-A
1	1	0	1	0	0	-A
1	1	1	0	-	1	0

Table 3: BRU1 truth table

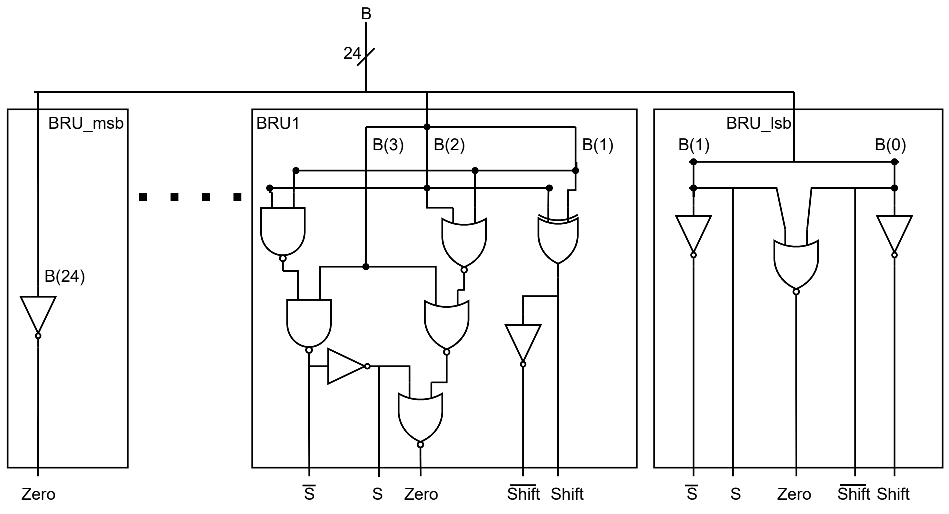
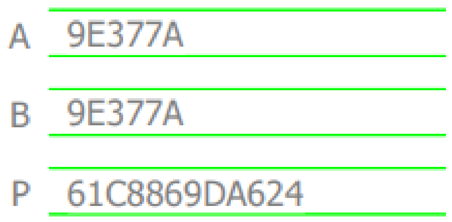


Figure 13: BRU RTL architecture

4.5 Modelsim Verification

In order to verify the proper behavior of the MBE multiplier radix 4, a testbench was created. As shown in Figure 14, the multiplier works as intended, since the obtained results match the expected ones.



```
A  9E377A
B  9E377A
P  61C8869DA624
```

Figure 14: MBE radix 4 testbench results on ModelSim wave window

5 Architectures comparison

To conclude, the developed MBE unsigned multiplier's performance has been evaluated with Design Compiler. For the sake of completeness, it was simulated with *optimize_registers*, *compile_ultra*, and both. Table 4 represents all the results obtained in this Lab, and will aid in their comparison.

	Max Frequency [MHz]	Area [μm^2]
Default	641	4120
CSA	230	4905
PParch	641	4153
optimize_registers	1176	4751
compile_ultra	613	4100
optimize_registers and compile_ultra	1315	5317
MBE mul (compile_ultra)	209	4734
MBE mul (optimize_registers)	714	8594
MBE mul (optimize_registers and compile_ultra)	781	7807

Table 4: Performance comparison

Despite all optimizations that were implemented, the developed MBE multiplier architecture does not have any remarkable advantage over the pipelined architectures analyzed in Section 3, and in fact it is worse in terms of both Frequency and Area. While the development of the MBE multiplier was a useful exercise, it is now clear that Design Compiler realizes better architectures in far less time.