Politecnico di Torino

III Facoltà di Ingegneria

Laboratory 2
Digital arithmetic

Master degree in Electrical Engineering

Authors: Group 21

Dilillo Nicola S284963
Moncalvo Stefano S290315
Carrano Lorenzo S281565

December 19, 2021

Contents

CHAPTER 1

Prototype

1.1 Introduction

In order to better and easily simulate the architecture, two scripts have been used:

- A bash script to control the simulation from high level: it is in charge of invoking the C script with the right command line argument during the proper simulation phase, and of running Modelsim.
- A C script in charge of performing various operations during different steps of the simulation, manipulating different files.

For simplicity, a single value is assigned to both operands each simulation cycle, de facto using the multiplier as a square-evaluation circuit.

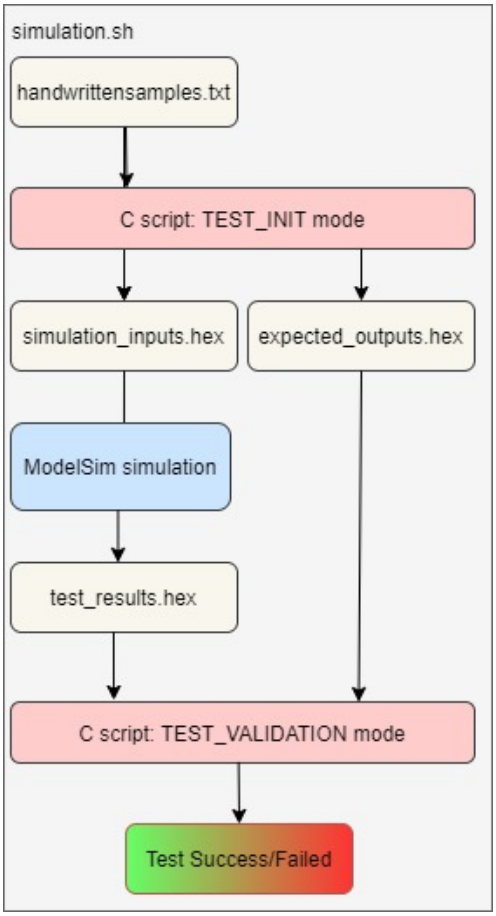
1.2 General simulation flow

The C script can accepts an additional command line parameter:

- -i corresponds to **TEST_INIT** mode: the script reads the handwrittensamples.txt file, in which desired inputs in decimal format are stored, e.g. 12.29487, -0.9872 etc, and generates two other files as output, both storing data in hexadecimal encoding: simulationinputs.hex, which contains the inputs for the Modlesim simulation expected_outputs.hex, which contains the outputs that are expected to be generated by the Modelsim simulation.
- -v corresponds to **TEST_VALIDATION** mode: the script executes a diff command between the self-generated expected_outputs.hex file and the file generated by the Modelsim simulation using a pipe to execute it in background and take back its output, then processing it in order to establish if the two files are identical, Test Successful, or different, Test Failed.

The flow of the simulation is the following:

1. the bash script executes the C program passing -i as command line parameter in order to generate the input vectors for the simulation and the file storing the expected results.
2. Modelsim is launched and the effective output file is generated.
3. the C program is executed passing -v as command line parameter in order to compare the simulation results with the expected ones.



1.3 C script - Key Aspects

Since IEEE754 is the standard encoding that is normally used to store floating point data inside a flash memory, to convert a floating point value in its equivalent hexadecimal encoding it's sufficient to use the **union** C data type. **Union** data type allows to store multiple-encoding data variables in the same memory location, thus it is sufficient read a value from the handwrittensamples.hex file and store it as a float variable inside the previously declared memory location, and then accessing it as a 32-bit data variable, using the C standard type **uint32_t**.

```
union IEEE754conv
{
    float f;
    uint32_t ieee754Value;
};
```

The previously described operation is trivially implemented by the following procedure, that is very convenient both in terms syntax and in terms of CPU time, since what is done is just a couple of memory accesses:

```
uint32_t convFloatinIEEE754(float val)
{
        union IEEE754conv conv;
        conv.f = val;
        return conv.ieee754Value;
}
```

1.4 Simulation Results

Simulating the architecture with the procedure described above, it has been confirmed that it behaves in the expected way, since the results it produces are coincident with the expected ones generated by the C script. Tests have been made with both positive and negative numbers, with different orders of magnitudes for the modules, on both Ubuntu and Centos Linux OS. In case a new simulation has to be performed on a different Linux system, a bash script, compile.bash, is provided in order to quickly recompile binaries.

CHAPTER 2

Testbench

2.1 Design modification

Before proceeding with the simulation some modifications have been added to improve the testability:

- Added registers to inputs and a reset signal, to achieve a better behavior.
- Added a VIN and VOUT signals that have the purpose to start the operation and to say when it's over.
- Added registers at the output of the significands multiplier, while maintaining the correct timing of Stage2.

2.2 Simulation

The testbench has been divided in three parts.

- The first part generates inputs that will feed the multiplier, that is the element under test (DUT). Those inputs will be taken from an input file that has been generated thanks to C prototype, like explained in the previous chapter. The same input feeds both factors and the VIN signal will be kept high until all input values have been used.
- When outputs start to be ready VOUT signal will be raised, and until this signal is high, a text file will be written in order to see which will be the produced results. Then those values will be compared with the golden ones, that have been generated from C prototype script, to check the correctness of the multiplier.
- The last part generates the clock that will feed the input generator, the data sink and the DUT.

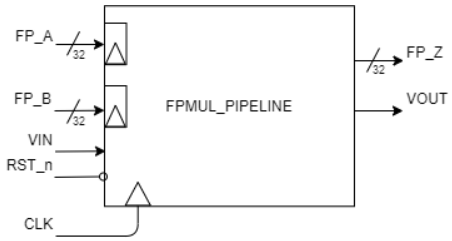


Figure 2.1:

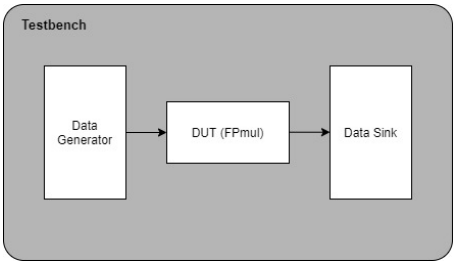


Figure 2.2:

CHAPTER 3

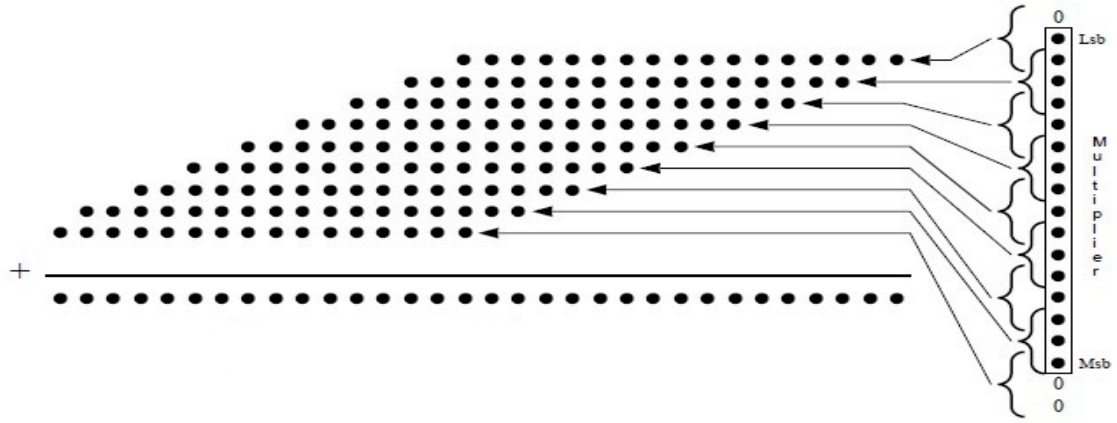
MBE Multiplier

3.1 Introduction

The goal of this section was to design a Modified Booth Encoder multiplier, a radix-4 variant of the standard operation, to be utilized in the provided floating point multiplier. Even though the operation was executed in 32 bits, the last 8 bits of the multiplicands were set to zero. It was then sufficient to implement a 24 bit version of the MBE. With this algorithm, the partial products are obtained by dividing the multiplier in 3-bit slices, where each slice shares 1 bit with the consecutive one; these are used to retrieve the corresponding partial product from the following table.

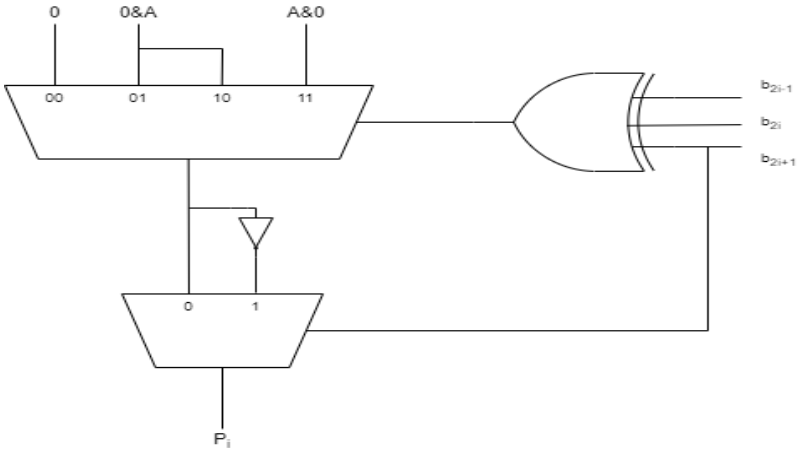
$b_{2i+1}b_{2i}b_{2i-1}$	p_i
000	0
001	a
010	a
011	2a
100	-2a
101	-a
110	-a
111	0

The sum of the partial products will be the result of the multiplication.



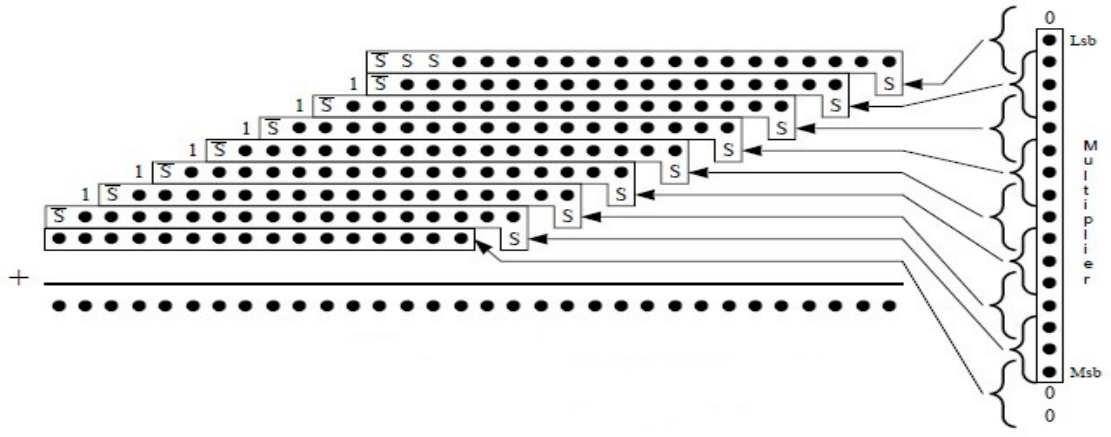
3.2 Partial Products Generation

In order to obtain all the needed partial products, a simple 4-input multiplexer was implemented. It receives as inputs 0, a and 2a, and as selection bits the result of this operation: $(b_{2i}b_{2i-1})xor(b_{2i+1}b_{2i+1})$ as shown in the following image. The b_{2i+1} bit is also used as the sign of the partial product: if the result is negative, the output of the multiplexer is negated in order to compute its 2's complement.



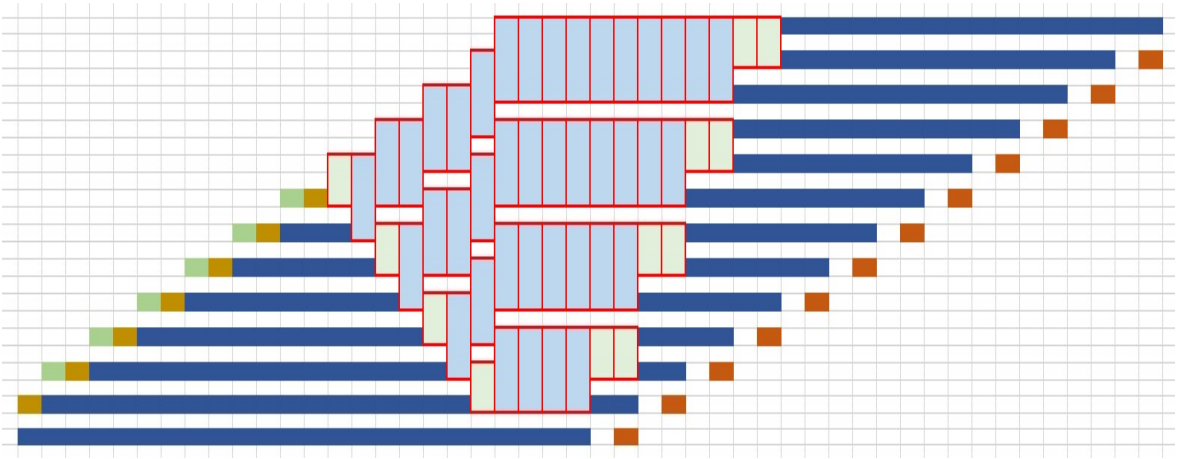
3.3 Sign Extension

The sign extension for unsigned multiplication is a technique that allows to reduce the number of redundant additions when computing the sum of the partial products. In particular, when an operand is negative, the string of $'1'$ in the MSBs causes an unnecessary number of bits to be included in the final sum. The algorithm avoids this problem, and also takes care of the $+1$ needed to compute the 2's complement without the need of an additional operation. The partial products are modified as shown in the following image, where S is the sign of the operand.



3.4 Dadda-Tree

The final step for the computation of the result is to sum the partial products together. To compute this additions a Dadda adder was implemented. This component achieves the solution by adding the operands by columns instead of rows. In particular, the Dadda implementation needs the number of rows to reduce to a specific value each iteration; full adders and half adders are utilized to sum bits of the same column, thus reducing the number of rows. Contrary to the Wallace adder, the Dadda is an "as late as possible" algorithm, meaning that the full and half adders are positioned as close as possible to the end of the operation. In order to visualize the positioning of the adders, schematics representing the operands were needed; the following picture is an example.



CHAPTER 4

Comparison Between Architectures

	Frequency	**Area**
Base Architecture	625.00MHz	4104.1
CSA Multiplier	223.21 MHz	4906.9
PPARCH Multiplier	641.03MHz	4182.8
Optimized with registers	1.08GHz	4656.9
Compiled with compile ultra	714.29 MHz	4478.9
MBE Standard Multiplier	357.14 MHz	4824.2
MBE opt. with registers	1.12 GHz	6318.3
MBE compiled with compile ultra	653.59MHz	4661.6

In general, it can be noticed that the best performance is obtained by means of optimization using registers. Compiling with compile ultra command returns a solution that is a good compromise between performance and total area. For example, compiling the base architecture using compile ultra, it can be achieved a performance growth of 14% at the cost of 9% of additional area, while modifying the base architecture inserting registers, a growth of 73% of performance is achieved, at the cost of about 13% of additional area. Thus, in general, if the area constraints are not so critical, optimizing by means of registers can be strongly efficient in terms of performance.