TU Delft

Processor Design Project

Report

Authors:
Henrique Dantas (4172922)
Luca Feltrin (4270355)

June 17, 2013

Contents

List of Figures

List of Tables

1. Multiplier

1.1. Wallace Tree Multiplier

For this project it was decided the most appropriate multiplier scheme would be the Wallace Tree Multiplier. The most important reason for this choice is due to its great performance although at the cost of gates and area.

The Wallace tree is a regular hardware structure to multiply two operands. It was invented by Chris Wallace, an Australian Computer Scientist in 1964.

The algorithm can be divided in three major steps:

1. The initial `AND` operation between all combinations of bits of each operand. The weights must be adjusted according to the location of the operands, just like in the classical pen-and-paper algorithm. The resulting tree, using dot notation, is shown in figure 1.1.

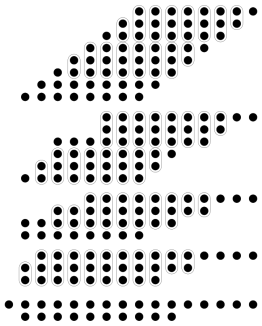


Figure 1.1: Resulting tree after executing step 1 for and 8 bit by 8 bit multiplication.

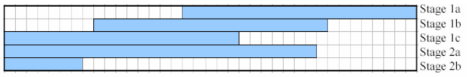
2. Thereafter the tree must be reduced through the use of half adder and full adders. These will convert each two or three "dots", respectively, into one and a carry out for the following column. This step shall be iterated sufficient times until only only two numbers remain.

3. Finally, the two remaining numbers can be summed with a conventional adder. The width of the result should be equal to the sum of the widths of the original operators. For example for a 32 bit times 32 bit operation, the result shall be 64 bits wide.

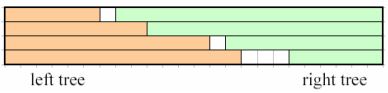
For our particular implementation the aforementioned description was modified to accommodate signed numbers through the use of the modified Baugh-Wooley algorithm. The details of this alteration will be explained in section 1.3.

1.2. Advantages and Disadvantages

The main advantages of this scheme is the speed obtained and regular mapping in hardware. The structure of the tree is consistent throughout the several steps. On the other this design is costly in the amount of gates used and there is some waste of area. This is particular noticeable on the extremes of each line of the tree, as shown in figure 1.2a. [2] details possible improvements that can be accomplished in this regard to improve the area cost of the Wallace multiplier. The main idea is to split the tree into two overlapping trees, hence saving area. Thereafter the additions take place in opposite directions. However due to lack of time it was not possible to implement the proposed ideas.



(a) Detail of Wallace Tree from [2]. It is clearly visible a significant percentage of unused area.



(b) Modified Wallace Tree from [2]. The image reflects the better area utilization based on the algorithm present in the paper.

Figure 1.2

1.3. Implementation and Simulation Results

This section will be used to introduce the reader to the implementation requirements for the multiplier. Followed by a simple, high-level description of the architecture used to accomplish them.

The implementation tries to closely mimic the three algorithm steps. However other signals necessary for the correct interface with the processor were added. According to the documentation [3], there are 4 input signals, `RST`, `CLK` and `MULI` and 1 output signal `MULO`. Moreover, there are 4 generics: `infer`, `multype`, `pipe` and `mac`. `MULI` includes the 32 bit operands (along with an extra signal bit), and severals flag to request flushing of the current operation, indicate signed multiplication, to initiate, and to start multiply and accumulate. On the other hand `MULO` includes a self explanatory ready signal, a nready signal (not used), condition codes that reflect if the result is zero or negative and finally the 64 bit result.

The most significant generic is multype that configures the multiplier to different operand's sizes. These are 16x16, 32x8, 32x16 and 32x32. It is important to note that a discrepancy between the documentation and

the actual code exists regarding the multiplier. The infer generic has been replace by tech (related to the target architecture) in the original VHDL file: `mul32.vhd`. Our custom multiplier replicates this configuration.

As mentioned in 1.1 the algorithm can be divided in three major parts.

On part 1 the operands are `AND`ed together and their weights are adjusted. To reflect this a three dimensional `STD_LOGIC_VECTOR` was created and named `WallaceTree`. The first dimension reflects the number of stages (or levels) necessary to finish the operation. This values are computed offline for the four multype possibilities. The second dimension relates to the number of 'lines' of the matrix, or its height. This values is always equal to the width of the first operand. Finally, the third accounts for the number of columns, or its width, which is necessarily equal to the sum of the width's of each operand. In practice the 3D array was converted to 1D as synthesis tools have better support for the latter, nonetheless the same principles stand.

To be able to multiply signed operands the modified Baugh-Wooley multiplication was used (*cf.* [4]). Therefore some values are complemented if the input is signed. Moreover on the final step a constant value is added to the two operands. This completes the first step.

Thereafter each group of 3 or 2 bits must be compressed to 1, using full-adders and half-adders respectively. If a odd number of bits exists in a certain column the last bit is transfered to the next level. Repeating this process sufficient times[1], the height of the matrix is reduced to 2. These can be added with a conventional adder to determine the final result.

To help with these operations the exact number of full-adders and half-adders for each column of each stage were precomputed. Having this information the appropriate inputs (x, y and carry in for the full adder) and outputs (s and carry out) are mapped to a full (or half) adder instance to determine the result.

Step 3 is simply an addition with the first two 'lines' of the last stage of the `WallaceTree` signal. However, if the multiplication is signed an extra `STD_LOGIC_VECTOR` is added to these, as explained in [4].

The extra signals indicating that the result is ready and the condition codes are updated to reflect the result of the operation. The evolution of the signals driven by our testbench is shown in 1.3.

[1]The exact number of iterations required can be inferred from the integer constant `levels` previously mentioned.

Figure 1.3: Screenshot of the Modelsim's Wave for the multiplier. For this particular simulation a 16x16 operation is shown, along with all the appropriate signals.

References

[1] CA book

[2] betterwallace

[3] doc

[4] part3