TU Delft

ET4171 Processor Design Project

Report

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

June 19, 2013

Contents

List of Figures

List of Tables

1. Introduction

For the Processor Design Project course we have been asked to improve the performance of the LEON3, a 32-bit SPARC V8 processor designed for embedded applications. Our main target is to decrease the computation time for certain benchmarks keeping the power consumption as low as possible, thus for us the most relevant compound metric is the `power`×`benchmark` score (`P`×`BS`).

The SPARC V8 architecture contemplates the use of instruction and special hardware for integer multiplications and divisions, but with the original configuration the multiplier takes 5 clock cycles to calculate the result and the divider 36, so one of the first things we decided to do is to improve these arithmetic cores. Simple algorithms can be implemented to obtain significant improvements.

In addition other changes are described at the end of this document.

2. Improved Arithmetic Cores

In order to improve the performance of the arithmetic unit we redesigned from the ground up both the two multiplier and divider units.

In order to make them compatible with the rest of the processor we studied in a detailed way all the handshaking signals.

The original multiplier can be configured for "2 cycles" of latency (instead of 5) through the use of `make xconfig`. It is important to note that for this particular setup the ready signal is not used. So although we are writing a new multiplier it is necessary to configure the processor with a 2-cycle multiplier as well, so the processor can handle the handshaking signals generated by our multiplier in the correct way.

For the divider there is not a previous configuration, so the processor knows that an operation is completed by inspecting the `ready` and `nready` signals which have been reproduced following the specifications.

The part of the core that handles the other signals such as `start`, `flush` or `holdn`, has been designed to mimic the original version, thereafter all the handshaking signals are handled and generated following the specifications to enable unit compatibility with the processor.

3. Multiplier

3.1. 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 3.1.

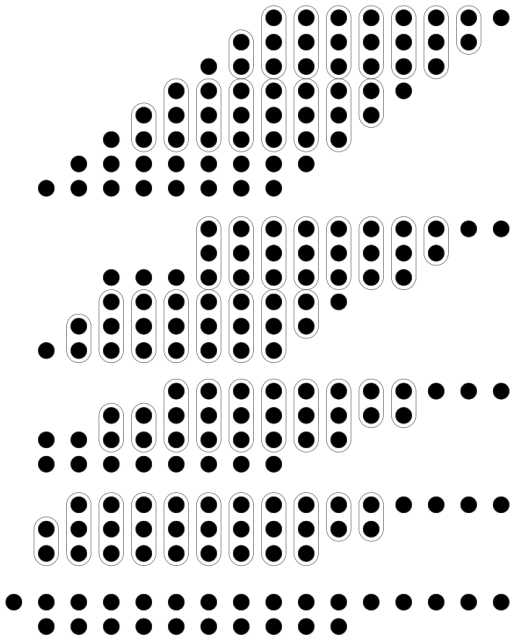


Figure 3.1: Resulting tree after executing step 1 for and 8 bit by 8 bit multiplication.

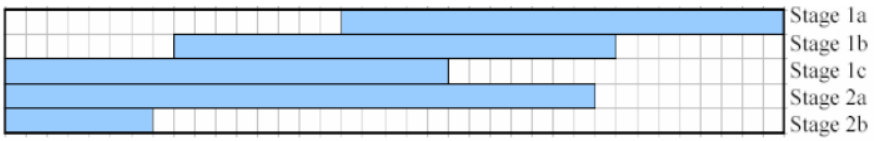
2. Thereafter the tree must be reduced through the use of half adders and full adders. These will convert each two or three "dots", respectively, into one and a carry out for the following column. This step shall be iterated sufficient times until only two numbers emerge.

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.

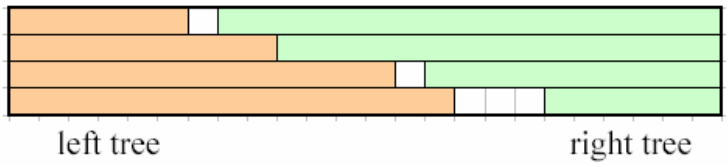
For our particular implementation the aforementioned description was modified to accommodate signed numbers through the use of the modified Baugh-Wooley algorithm. The details of this alteration will be explained in section 3.3.

3.2. Advantages and Disadvantages

The main advantages of this scheme are the speed obtained and regular mapping in hardware. The structure of the tree is consistent throughout the several steps. On the other hand, this design is costly in the amount of gates necessary and total area. The latter can be improved by reorganizing the tree for efficiency. The 'waste' in the traditional structure is particularly noticeable on the extremes of each line of the tree, as shown in figure 3.2a. [2] details possible techniques that can be implemented to tackle this issue and reduce 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 on the multiplier.



(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 3.2: Wallace tree structure before and after the reorganization.

3.3. 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 additional signals are necessary for the correct interface with the processor. According to the documentation [3] and [**?**], there are 3 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 3.1 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. These values are computed offline for the four multype possibilities. The second dimension relates to the number of 'lines' of the matrix, or its height. The range is always equal to the width of the first operand. Finally, the third dimension 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 action 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 an odd number of bits exist 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. Moreover two other constants arrays exist that indicate the number of carry ins each column receives and if it has a remainder bit (odd numbered of lines). Having this information the location of 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 indicate 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 3.3.

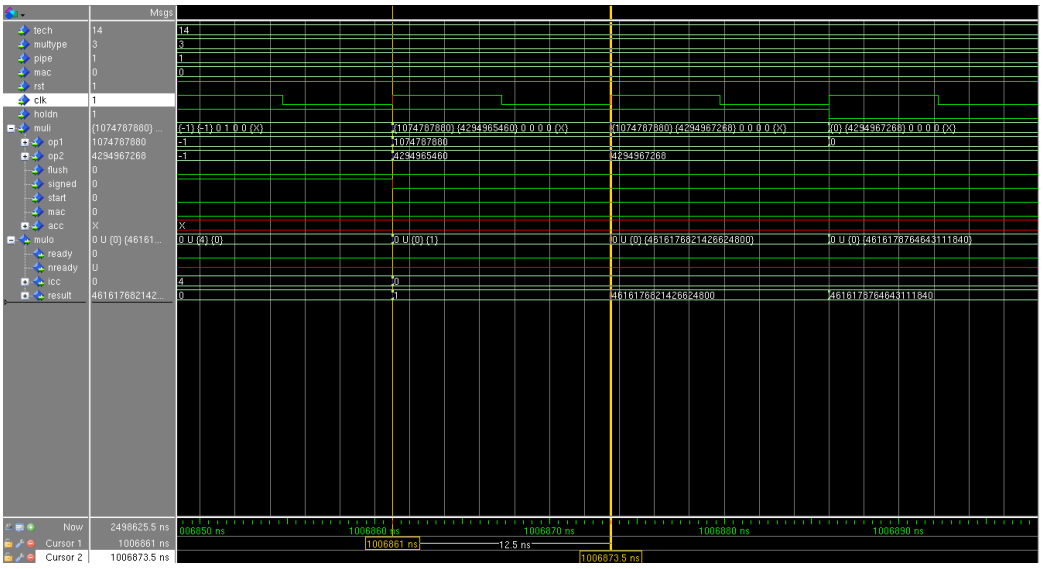


Figure 3.3: Screenshot of the Modelsim's Wave for the multiplier. For this particular simulation a 32x32 operation is shown, along with all the appropriate signals.

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

4. Divider

The algorithm implemented in the original version of the processor is one of the simplest but the slowest available.

Several other algorithms can compute the division faster but all of them present disadvantages that must be taken into account according to the target application.

Algorithms like repeated multiplication or reciprocation are fast but require a significant amount of area, similarly an array divider would have been very fast only If we had control on the place&routing process in order to create a regular structure. In the end we decided to implement a simple radix-4 division algorithm for simplicity of implementation and of the circuit itself.

Using an higher radix could have improved performance but the size of the lookup table required by the algorithm would have increased again the area consumption.

The divider consist in a state machine (its diagram is shown in figure 4.1) which checks if the inputs will generate an overflow and performs a preliminary shift to put the divisor in the appropriate range to be computed correctly.

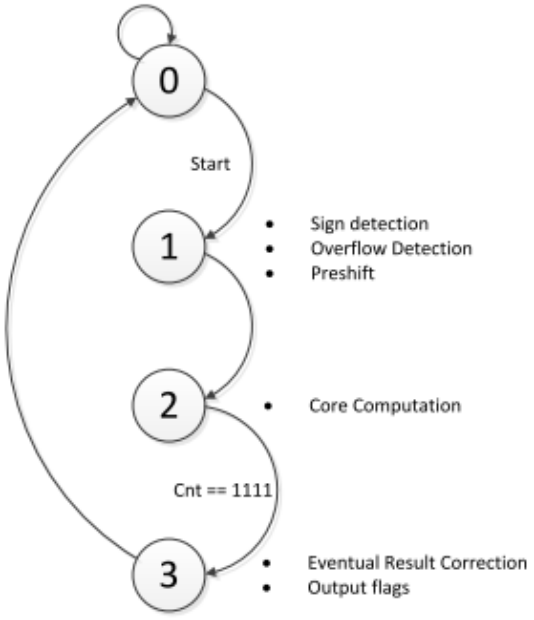


Figure 4.1: Divider State Diagram.

After that, the real computation begins and lasts 16 clock cycles. The block diagram of the divider while it's in this state is shown in figure 4.2.

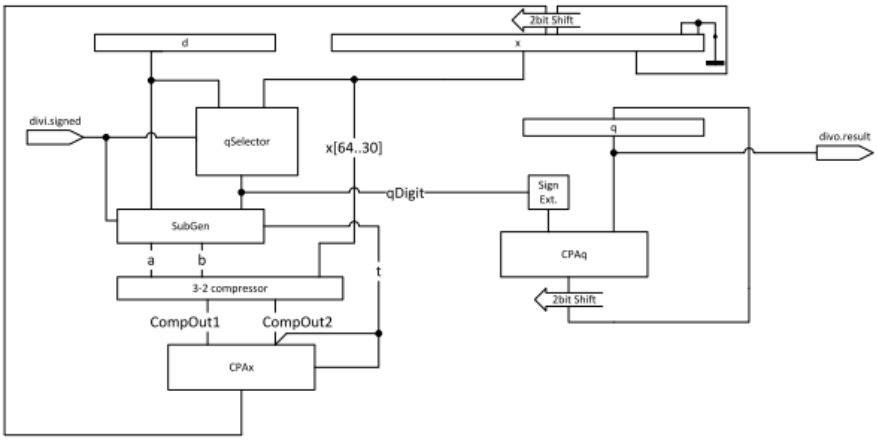


Figure 4.2: Divider Block Diagram (during Core Computation, State = 1).

The algorithm is very similar to the original radix-2 version but in this case the partial reminder (x) is shifted by 2 bits every cycle and the circuit has to guess the quotient digit from the range [-3,3]. "qSelector" is the lookup table which performs the quotient digit guessing and it's based on the p- d plot of the radix-4 SRT division shown in figure 4.3. In case of unsigned division only the right half of the p-d plot is being used.

The quotient digits are in a radix-4 redundant format so a conversion to binary format is needed. The conversion is performed gradually every cycle by the 32-bit adder "CPAq" which shift and sum each generated digit with the already calculated quotient.

One could think that it would be better keep the quotient in a radix-4 redundant format and avoid the addition in order not to slow down the execution every cycle so that the clock frequency could be higher, but also the original divider executes a 32-bit addition every cycle so from this point of view our divider is not worse than the original one, moreover a conversion from radix-4 redundant format to binary is quite complicated, doing this it consists in a simple addition.

The same concept has been used also for the computation of the partial reminder.

An addition/subtraction in Carry-Save format would have been much faster and also easier, but the selection of the quotient digit would have required the analysis of the most significant bits of both the sum and the carry making the lookup table several orders of magnitude bigger. In our divider "SubGen" generates the multiple of d to sum with the current partial reminder in a carry save format, all this operands are been compressed by a 3-2 compressor (1 full adder of delay) and finally the new partial reminder is

calculated with a 35-bit adder, “CPAx”

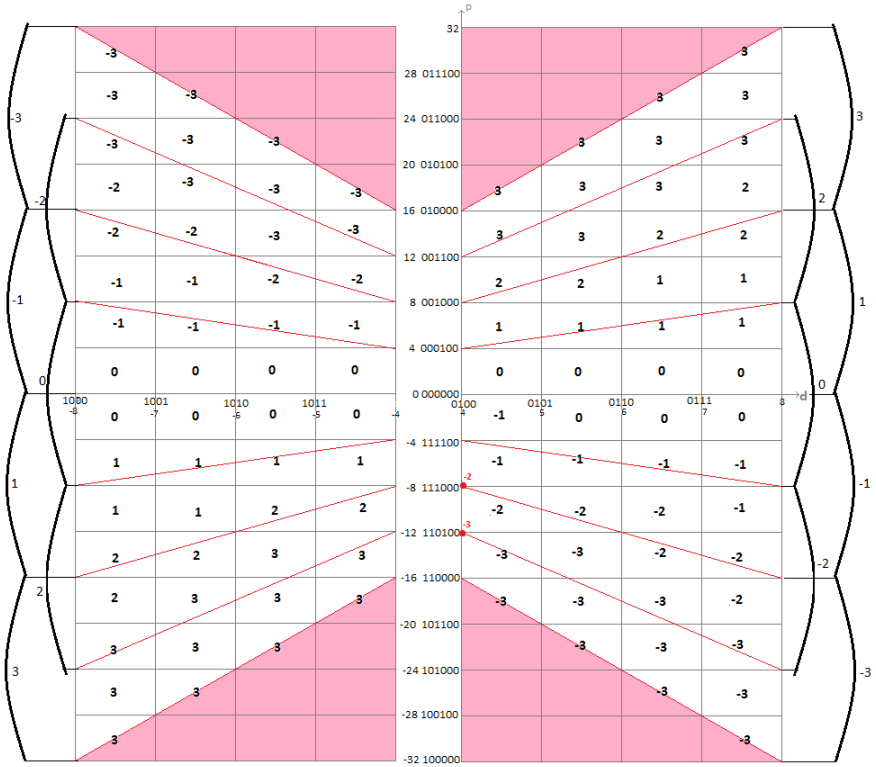


Figure 4.3: Radix-4 P-D Plot (Red Dots are Exceptions).

Other solutions have been analyzed, such as having a lookup table only for unsigned division, half of the size of the final version, and handle the sign separately but the synthesis has shown that the resources utilization would have not changed significantly while one more cycle would have been needed. Hence, we decided to keep the current divider.

At last, in order to check the compatibility of the radix-4 divider with the original one we performed a simulation of the two versions with the same output, the only difference is in the case of overflows where the flag is set up correctly but the result is different or in some cases undefined, but since there is an overflow the result has no meaning so this is acceptable.

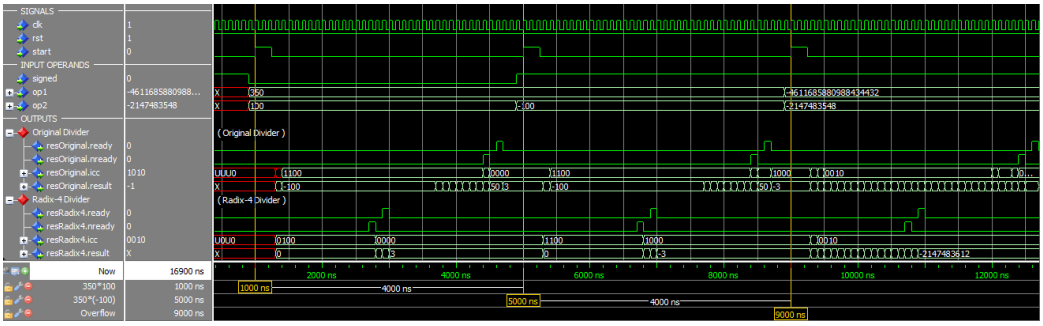


Figure 4.4: Signal Dump Of Radix-4 Divider Vs. Original Divider.

5. Results

5.1. Synthesis

In order to evaluate the performance of our improved processor we need first to evaluate the performance of the baseline version of the processor.

The synthesis tool reported the values shown in table 5.1 for the resources utilization.

Timing Summary (max clock freq. [MHz])	80.522
# of Occupied Slices	9904
Total # of 4-input LUTs	16889
Quiescent power [W]	2.467
Dynamic power [W]	0.721
Total power [W]	3.188
P/f [W/MHz]	0.03959

Table 5.1: Resource Utilization Baseline.

Notice that the value "P/f" indicate the energetic efficiency of the processor, since the power consumption is almost proportional to the clock frequency we can use this value to estimate the power consumption at different clock frequencies.

From the synthesis report we can also see the slowest path which determines the max clock frequency.

This path is from "ddrsp0.ddrc0/ddr32.ddrc/ra.raddr_0" to "ddrsp0.ddrc0/ddr_phy0/ddr_ph

Those components belong to the SDRAM controller and the path is located between the controller and the physical interface with the memory.

The synthesis of our modified version gave us the results showed in table 5.2:

Timing Summary (max clock freq. [MHz])	40.197
# of Occupied Slices	11886
Total # of 4-input LUTs	20469
Quiescent power [W]	2.511
Dynamic power [W]	0.832
Total power [W]	3.343
P/f [W/MHz]	0.08317

Table 5.2: Resource Utilization Modified Version.

As we expected the area consumption is quite worse because the unit we designed is way more complex than the baseline, in particular the lookup table in the divider consumes lots of area.

Also the power consumption increased for the same reason, the algorithm is more complicated so more energy is consumed to do all the calculations, but fortunately the disadvantage is negligible compared to the advantage as it will be shown later.

The only parameter which is better is the maximum clock frequency achievable but probably this is only due to the synthesis tool that rearranged the processor's component in a different way that is better for the clock performance.

5.2. Benchmark Scores

Once the processor is synthesized has been loaded in a FPGA board, then Linux has been launched on the processor as much as several benchmarks. In table 5.3 the execution times of these benchmarks are reported, for detailed scores see the excel file attached.

Stanford [s]	2.30
Whetstone [s]	116.2
Gmpbench Multiply [Op/ s]	781
Gmpbench Divide [Op/ s]	15876
Gmpbench RSA [Op/ s]	5123
Division [s]	8.06
Mibench JPEG (average) [s]	23.215
SSD [s]	10.59
Total [s]	**219.28**

Table 5.3: Benchmarks Scores Baseline.

The scores obtained with the modified processor are reported in table 5.4.

Stanford [s]	2.21
Whetstone [s]	112.08
Gmpbench Multiply [Op/ s]	914
Gmpbench Divide [Op/ s]	19205
Gmpbench RSA [Op/ s]	5353
Division [s]	7.31
Mibench JPEG (average) [s]	21.76
SSD [s]	8.60
Total [s]	**206.92**

Table 5.4: Benchmarks Scores Modified Version.

From these results we can see that almost every benchmark had a slight improvement, in the end the total execution time improved by 5.6%.

These scores are not as good as we expected but probably the execution of an operative system on the processor cause a not negligible overhead in the execution due to the scheduling (the divider takes about half the time to execute but the execution time of the division related benchmarks is only about 10% better).

5.3. Metrics Comparison

In order to get a fair comparison between the baseline and the improved processor some standard metrics have to be calculated and studied.

Usually these metrics are `A`, the area consumption here calculated as the weighted sum of the number of occupied slices and the number of 4-input LUTs used where the weight is the reciprocal of the number of available resources, `D` is the delay or the reciprocal of the max clock frequency and indicate the delay of the slowest path, `P` is the power and it's simply calculated as the total power consumed by the Dhrystone benchmark used for the simulation and `BS` is the benchmark score which indicate how fast a program can be executed, it's calculated as the total execution time of the benchmarks on the FPGA board.

Moreover some composite metrics can be observed: these metrics consider two or more primitive metrics and often are more interesting than these ones because when a modification is done on the processor usually we obtain two opposite effects, one primitive metrics increases while another one decreases, but what we want is that all in all we have an improvement. Composite metrics show the overall performance.

Since we want to speed-up the execution of the software while keeping the power consumption low because this is a processor designed for embedded

applications, the composite metric we are interested the most is P×BS which show how much the processor is able to execute the software fast with the same amount of energy.

Other composite metrics are A×D, A×BS and P×D. Since we focused on the improvement of the execution time and power consumption one can notice that these metrics are worse in our version compared to the baseline, because the area consumption has increased notably and the delay hadn't had any significant changes.

Both the baseline version's and our improved version's synthesis and benchmark results have be condensed in this metrics and their values are in the table 5.5.

	Primitive Metrics			
Version	A	D	P	BS
Baseline	2.68×10^4	1.24×10^{-2}	3.19	2.19×10^2
Modified	3.24×10^4	2.49×10^{-2}	3.34	2.07×10^2
Improvements	21%	100%	5%	-6%
	Composite Metrics			
Version	A×D	A×BS	P×D	P×BS
Baseline	3.33×10^2	5.88×10^6	3.96×10^{-2}	6.99×10^2
Modified	8.05×10^2	6.69×10^6	8.32×10^{-2}	6.92×10^2
Improvements	142%	14%	110%	-1%

Table 5.5: Final Metrics For Baseline And Improved Versions.

6. Conclusions and further improvements

The improvements have not been as good as we expected but now our modified processor is more 'formated' from an energetic point of view making it more suitable for embedded applications.

Because of lack of time we didn't do any other changes, but of course there are many things to change in the architecture to improve further the performance.

The size and structure of the cache memory can be changed to decrease the probability of misses and so the benchmarks execution time, but this can be done using the configuration tool and so it would have not been an our real achievement, moreover increasing the cache size probably would have increased also the power consumption making things worse.

The LEON3 uses a static branch prediction in the integer unit, which is a good compromise between power consumption, because no difficult computation is needed, and gain in terms of execution time. To improve performance further a 1 bit or a 2 bit branch prediction buffer algorithm. The calculation needed is more complicated and it has to be done very often (30% of the instructions are branches) so the power consumption probably would increase, but the gain in terms of execution time could be worth it.

In the end another heavy modification that could have been done is making the integer unit super-scalar and implementing Out of Order execution. Hypothetically it could yield significant performance improvements in terms of execution time. Notwithstanding a complete re-design of the integer unit would have been needed, which deemed impossible to complete in the (small) time budget for this project.

References

[1] Parhami, Behrooz. Computer arithmetic: algorithms and hardware designs. Oxford University Press, Inc., 2009.

[2] Parhami, Behrooz. Computer arithmetic: Part III, Multiplication. Available at `http://www.ece.ucsb.edu/~parhami/text_comp_arit.htm#slides`.

[3] Bohsali, Mounir, and Michael Doan. "Rectangular Styled Wallace Tree Multipliers."

[4] GRLIB IP Core User's Manual Version 1.1.0 - B4104, November 2010

[5] GRLIB IP Library User's Manual Version 1.1.0 - B4100, October 1, 2010