TU DELFT

ET4171 PROCESSOR DESIGN PROJECT

Report

Authors:
Henrique DANTAS (4172922)
Luca FELTRIN (4270355)

July 25, 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.

The division is not a very common operation, even if in the baseline version is quite slow and can be improved, on the other hand the multiplication is quite frequent. Apart from the calculations for the application software, the multiplier may also be used to calculate addresses for array access, and therefore the benchmarks we will run on the processor, as well as the operating system, can benefit greatly from the improvements.

In section 2 an analysis of the current baseline of the processor is done in order to find the weak points that will be improved. Thereafter in section 3 the improvements to the arithmetic unit, *i.e.* the multiplier and divider, are discussed. In section 4 the results from the simulation, synthesis and the FPGA board are presented and discussed, the section also examines our methodology to compare different designs based on various metrics. In the end some further improvements of the processor are suggested for future work.

2. Baseline Analysis and Working Plan

In order to find out which modifications will augment the processor's performance first an analysis of the baseline version is needed. By looking at the documentation and the results of the synthesis of the baseline version we divided all the potential modifications in three categories:

- Arithmetic improvements
- Clock frequency improvement
- Architectural improvements

As explained before, the SPARC V8 architecture used by the LEON3 processor contemplates several assembly instructions for complex operations such as multiplication and division. The integer unit of the baseline version is equipped with 2 units that can execute hardware multiplications and divisions, but the performance in terms of execution time of these ones are quite poor. Even if they do not occupy too much area with the default settings the multiplication takes 5 clock cycles and the division 36, advanced albeit simple algorithms can be implemented to significantly reduce the execution times. Nonetheless, prior to implementing any improvement it is important to evaluate the corresponding gains with respect to the compound metric we elected as most significant: `P×BS`.

Necessarily the complexity of the resulting circuits will increase and therefore so will the power consumption. However the baseline version's units are not particularly low power. They use carry propagate adders which are not very efficient in terms of power and path delay, thus there is room to improve these aspects and consequently reduce the weight of the power increase on the modified processor.

In addition it is relevant to study how often these instructions are used by an average program. Assuming that the given benchmarks represent the average usage of the processor's resources, it's possible to see that the division is only widely employed in a couple of those: "Division" and "GMPbench, divide". On the remaining the benchmarks the use is sparse.

Consequently the hypothetical gains in benchmarks scores due to division improvements will be modest. Albeit significant enough to justify its redesign. As explained in section 3.2, a simple division algorithm can be implemented to halve the execution time of the baseline version, without having a hefty power growth.

On the contrary, the multiplication, is extensively used by all types of benchmarks. Moreover, the unit can also be used to calculate addresses by the operating system running on the FPGA board. Therefore improvements in this unit are expected to have a greater impact on the benchmark scores.

Inspecting the code for the original multiplier we discovered that there are two possibilities. The first one is to allow the synthesizer to provide the multiplier. This can be defined through the `infer` generic of the `mul32`. Obviously this option should only be used if the synthesis tool is capable of inferring an efficient multiplier implementation. In VHDL code this is simply represented as a $*$.

Alternatively there are precompiled components included in the project tree. On the `mul32` unit they are invoked according to the distinct input sizes 16x16 (`mul_17_17`), 32x8 (`mul_33_9`), 32x16 (`mul_33_17`) and 32x32 (`mul_33_33`).

Therefore the original implementation is opaque to the user and cannot be modified to suit our requirements. For this reason and for total flexibility in the design process we elected to rewrite the multiplier from the ground up.

As explained, reducing the latency of the multiplications can have an important effect on the overall performance of the processor which shall be reflected in the benchmark scores. Moreover since we are targeting the `power`$\times$`benchmark` metric we would like to develop a unit that prioritizes performance and power consumption over other aspects, such as area footprint. Having these trade-offs in mind we can proceed to investigate various multiplier schemes. This analysis will be shown later on, in section 3.1.

After modifying the latency of the multiplier of the original design through `make xconfig` the slowest path of the complete processor, according to the synthesizer, is located in the DRAM controller (from `ddrsp0.ddrc0/ddr32.ddrc/ra.raddr_0` to `ddrsp0.ddrc0/ddr_phy0/ddr_phy0/xc4v.ddr_phy0/casgen`).

Altering the DRAM to reduce the path is expected to be very arduous. First, because the delay in the circuit is affected by the place&routing process, so after the re-design of the arithmetic unit this path could change. Second, the documentation is exiguous on the implementation details, moreover the limited available for this project forced us to focus our work on the changes with the highest impact. Therefore we decided to postpone further analysis until we had a working version with the improved arithmetic core in order to see how the slowest path evolved.

About the architecture, there are several different improvements possible. The baseline version is equipped with a static branch predictor, which can significantly augment performance. Nonetheless even better algorithms exist such as one or two bit branch prediction buffer or correlation prediction. An improvement in this category may be moderate but present in every kind of benchmark: on average, branches constitute 30% of all instruction, so the execution time can be significantly reduced.

The static prediction doesn't contemplate heavy calculations, indeed it's very simple so the power consumption in the baseline version due to the prediction is small. Usually more advanced algorithms are employed. For the branch prediction buffer the algorithm consists of a small state machine, but even a small increase can be very significant as more calculations are executed in every fetched branch. Notwithstanding with respect to the execution time we believe the benefits outweigh the costs.

There are others modification that can be done, for example designing a scalable architecture such as Tomasulo's. Still this kind of changes require a deep alteration of the complete architecture. The code for the baseline version of the integer unit is over 3000 lines of code, and therefore it's very unlikely to have a working scalar integer unit in the small time we have for this project. Consequently we decided to allocate our time on other improvements.

In conclusion, we focused the bulk of our efforts on the arithmetic unit since we consider it to be the most efficient way to invest our resources, given the ratio between the possible improvements and the corresponding work required. Further improvements will take place, if time permits, after completing the multiplication and division units.

3. Improved Arithmetic Cores

This section will explain the design process for the multiplier and divider units. For each the rationale behind the chosen scheme will be detailed, along with its principal advantages and disadvantages. This analysis will necessarily feature a comparison between the different possibilities considered.

Thereafter details of the implementation will be given along with the important handshaking signals that ensure the interface between the units and the processor. Finally, the verification phase will be presented. This includes the testbenches used during the development processed.

As previously stated, in order to improve the performance of the arithmetic unit we redesigned from the ground up both the two units: multiplier and divider.

Alternatively the original multiplier can be configured for 2 or 4 cycles of latency (instead of 5) through the use of `make xconfig`. So although we are writing a new unit it is necessary to configure the processor with the appropriate latency as well, so it 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. A more detailed description of these interface signals is presented later in sections 3.1 and 3.2, respectively for the multiplier and for the divider.

3.1. Multiplier

The simplest multiplier consists of an iterative shift-add routine, which works, as the name implies, adding pairs of bits, accumulating the result and shifting it appropriately. This closely resembles the pen-and-paper algorithm for multiplication. The main hindrance of this technique is the serial nature. It is necessary to wait for one result to advance to the next operation. Therefore a `n-bit`×`m-bit` operating costs $n + m$ cycles. On the other hand is very compact and uses very few components, so the power consumption is low. Nonetheless due to its latency it is not suitable for this project.

Another easy solution would be to define a look-up table multiplier. For this, all possible combination of inputs are tested and the corresponding results are stored for later retrieval. This makes the multiplier very fast but the table size grows exponentially. Therefore for our input sizes this option is not suitable.

In order to improve the performance there are two possibilities:

- Reduce the number of operands (*i.e.* high-radix multipliers)
- Adding the operands faster (*i.e.* tree and array multipliers)

Starting with the first item, high-radix operations have several advantages. They can reduce the size of the operands by "compressing" them in a higher radix.

First, the original operands are converted to the new radix. Then the necessary computations are performed on the converted values. And finally the result is converted back to the original representation.

The translation steps add some overhead to the process and therefore the number of operations on the converted values should be large enough to marginalize the introduced overhead.

Handling "dificult" multiples, such as 3, is also an issue. A possible solution is precomputing these values or rewriting them, *e.g.* $3a = 4a - a$. However these solutions for the corner cases add some complexity and irregularity to the circuit.

Booth recoding is another way to improve the first method. To recode the operand we replace the LSB of a series of 1s by -1 and the first 0 after that series by 1. Thereafter we group each pair and add the result (in radix 4). Example 3.1 from [1] demonstrates this procedure.

10	01	11	01	10	10	11	10	Operand x
-10	10	0-1	10	-11	-11	00	-10	Recoded Version y
-2	2	-1	2	-1	-1	0	-2	Radix-4 Version z

Example 3.1: Example for radix-4 Both recoding (originally from [1]).

The main advantage is halving the number of partial products while keeping the circuit simple (there are no *difficult* multiples). Booth recoding can be combined with Carry Save Adders for fast multiplication. Actually this combination results in a well balanced solution in the *Multiplier Design Spectrum.* It ranges from simple but slow multipliers to very fast but expensive choices (*i.e.* Full tree). However since performance is out first class design constraint it was decided to use a Wallace Tree which is a variation in CSA tree design. The respective details and consequent implementation are featured in section 3.1.1.

3.1.1. 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 well known pen-and-paper algorithm. The resulting tree, using dot notation, is shown in figure 3.1.

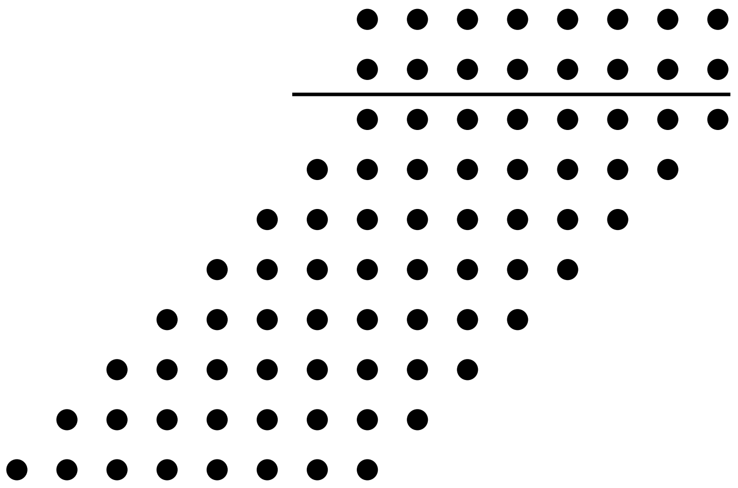


Figure 3.1: Resulting tree after executing step 1 for and 8 bit by 8 bit multiplication.

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.

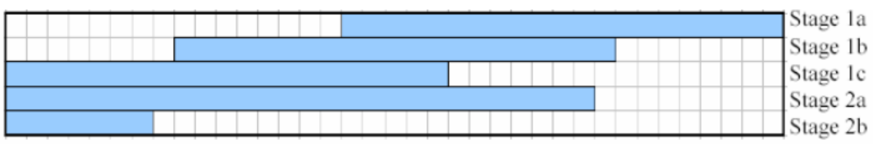
For our particular implementation the aforementioned description was modified to accommodate signed numbers through the use of the modified

Baugh-Wooley algorithm. A simple example is shown in 3.2. This is a simple modification that only adds a level of operands, which is negligible given the height generated by the available inputs.

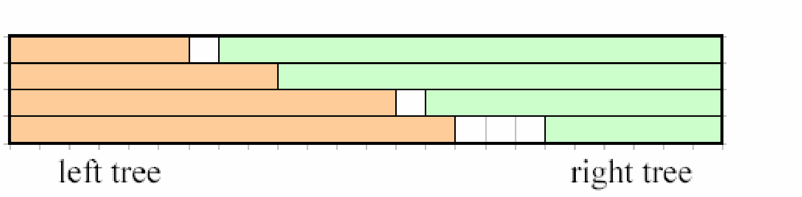
$$
\begin{array}{cccccccccc}
 & & & & & a_4 & a_3 & a_2 & a_1 & a_0 \\
 & & & & & b_4 & b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & & & \overline{a_4x_0} & a_3x_0 & a_2x_0 & a_1x_0 & a_0x_0 \\
 & & & & \overline{a_4x_1} & a_3x_1 & a_2x_1 & a_1x_1 & a_0x_1 & \\
 & & & \overline{a_4x_2} & a_3x_2 & a_2x_2 & a_1x_2 & a_0x_2 & & \\
 & & \overline{a_4x_3} & a_3x_3 & a_2x_3 & a_1x_3 & a_0x_3 & & & \\
 & a_4x_4 & \overline{a_3x_4} & \overline{a_2x_4} & \overline{a_1x_4} & \overline{a_0x_4} & & & & \\
1 & & & & 1 & & & & & \\
\hline
p_9 & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
$$

Example 3.2: Example for Modified Baugh-Wooley algorithm (originally from [1]).

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. [3] details possible techniques that can be implemented to tackle this issue and reduce the area footprint 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 [3]. It is clearly visible a significant percentage of unused area.



(b) Modified Wallace Tree from [3]. 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.

3.1.2. Implementation and Verification

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 and respective components used to accomplish them. The important handshaking signals will also be explained. It will also feature the procedures used to verify the implementation and what results are expected. Finally the conclusion will close the section.

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 [4] and [5], 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 several 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, 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.

The aforementioned ports and generics are summarized in figure 3.3.

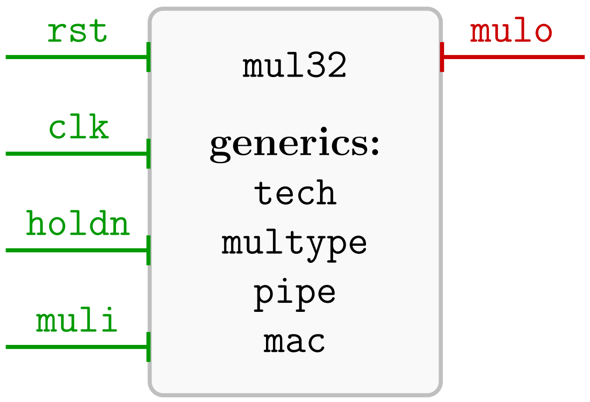


Figure 3.3: Diagram of the `mul32` unit, including all `in` and `out` ports as well as generics.

As mentioned in 3.1.1 the algorithm can be divided in three major parts. However to incorporate the starting signal a "step 0" was added.

Step 0 is a process that simply monitors the reset and start signals to update some internals signals accordingly. This is important to ensure the multiplier respects the handshaking signals and handles correctly the processor's requests.

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. Figure 3.4 summarizes this information. In practice the 3D array was converted to 1D as synthesis tools have better support for the latter, nonetheless the same principles stand. This step is accomplished in VHDL with `FOR-GENERATES` and `IF-GENERATES`. Originally we described this structure with processes, which allowed a more concise and simpler code. However the synthesis tool was not able to complete the synthesization process with the size of the loops we required.

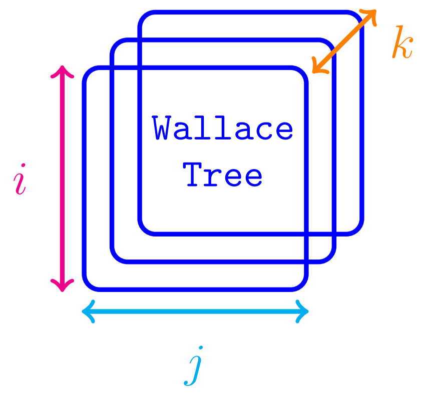


Figure 3.4: Diagram explaining the structure of the `WallaceTree` 3D array, with each dimension explicitly shown.

The tree is reorganized so all columns start from the 'top', as shown in 3.5b. This modification simplifies the implementation of the algorithm for the next steps.

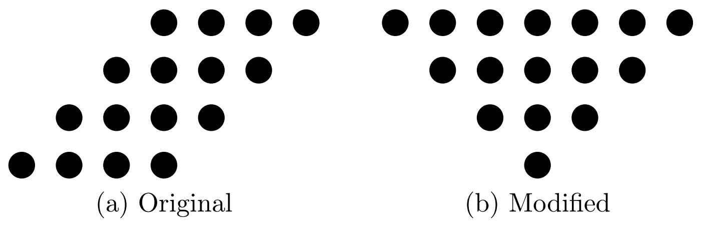
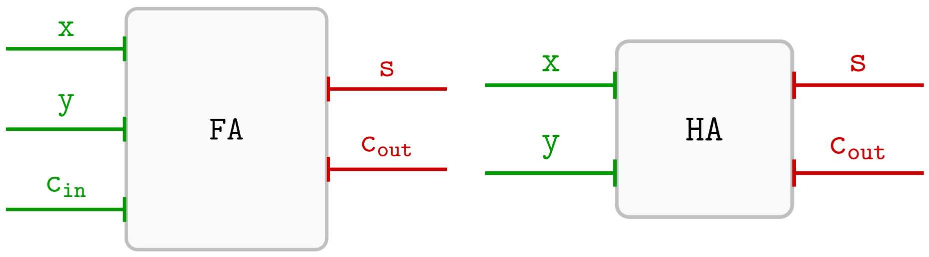


Figure 3.5: Reorganization of the Wallace tree for easier manipulation in further steps.

To be able to multiply signed operands the modified Baugh-Wooley multiplication was used (*cf.* [2] and example 3.2). 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.

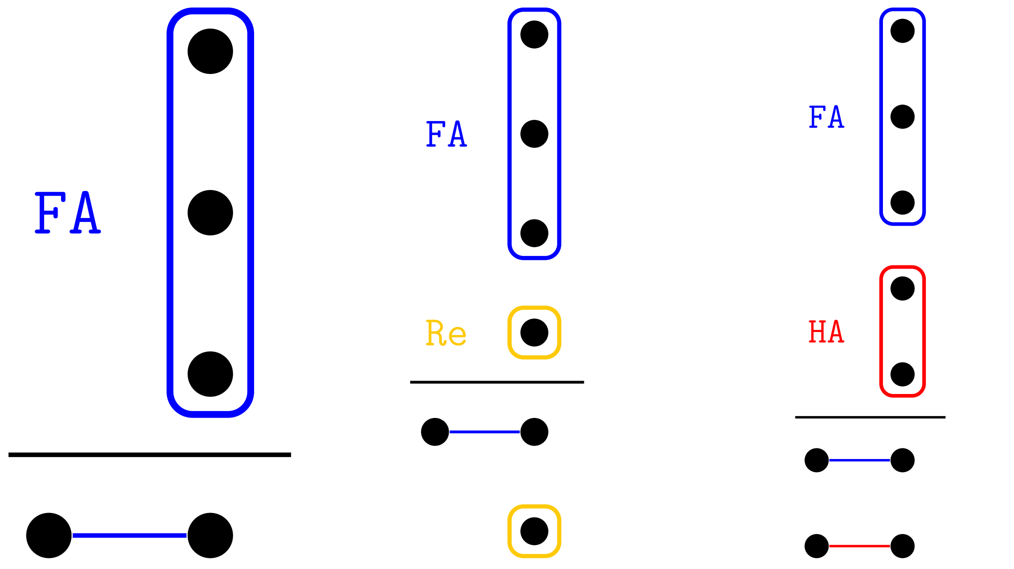


(a) Full Adder (b) Half Adder

Figure 3.6: Diagrams describing the interface ports of the Adders used.

Thereafter each group of 3 or 2 bits must be compressed to 1, using full-adders and half-adders respectively (*cf.* 3.6). If an odd number of bits exist in a certain column the last bit is transferred to the next level. Figure 3.7 assists the reader in understanding the various compressions used. 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.

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



(a) Column with 3 bits. (b) Column with 4 bits. (c) Column with 5 bits.

Figure 3.7: Compressing columns in the WallaceTree with Full Adders and Half Adders. Each example features on column of the tree in level k (before compression) and $k+1$ (after compression).

To help with these operations the exact number of full-adders and half-adders for each column of each stage were precomputed[2]. 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 a process responsible to add with the first two 'lines' of the last stage of the `WallaceTree` signal which yields the final result. However, if the multiplication is signed an extra `STD_LOGIC_VECTOR` is added to these, as explained in [2]. Moreover the condition codes are also computed.

In order to verify the correct behaviour of the modified multiplier it was fundamental to write a simple but comprehensive testbench. Other than all the aforementioned input and output signals the testbench also features some debugging signals that were used during the development phase to observe the behavior of (otherwise) internal signals. The debugging signals were compared to the output of a C program that replicated the desired behavior for the multiplier. This way it was possible to look for differences in the

[2]The precomputation was performed with a C program which simply runs the algorithm sequentially to determine the necessary components on each column for all stages.

VHDL signals and C `printf`'s to easily spot bugs as close as possible to their origin. The advantages of using this method cannot be understated as it allowed us to efficiently debug the VHDL code, notably step 2 of the algorithm.

To test our design several combinations of `holdn` and `start` signals were used in out testbench. In particular changing their frequency with respect to the frequency of the operands. Regarding the latter, the design was tested with operands close to the maximum allowed (*e.g.* for 32x32, $2^{32} \times 2^{32} - 1$, or $-2^{32} \times -2^{32} - 1$) and also around 0. Also multiplying two negative values, or one positive and one negative.

An example describing the evolution of the signals driven by our testbench is shown in 3.8.

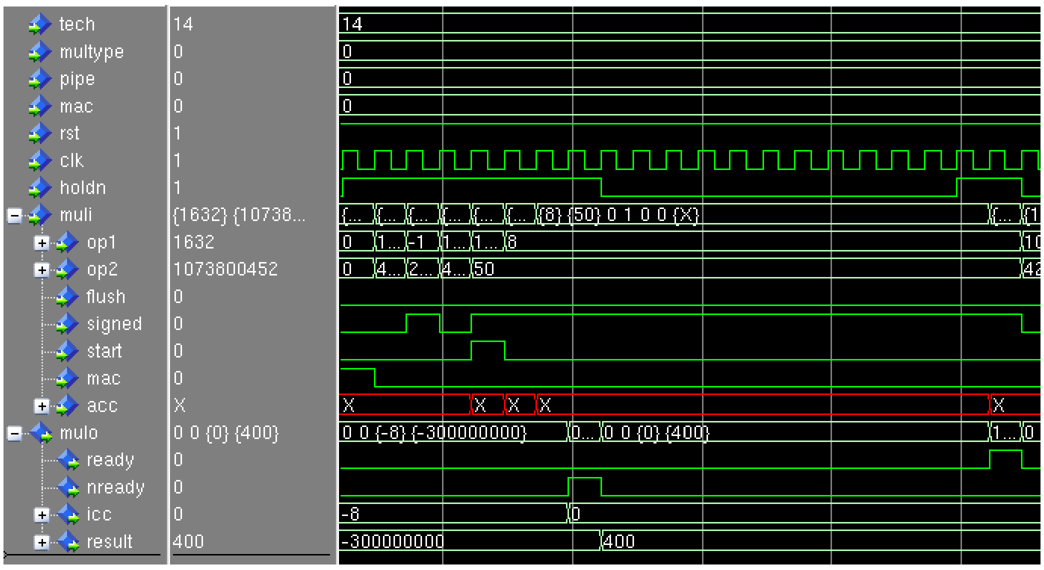
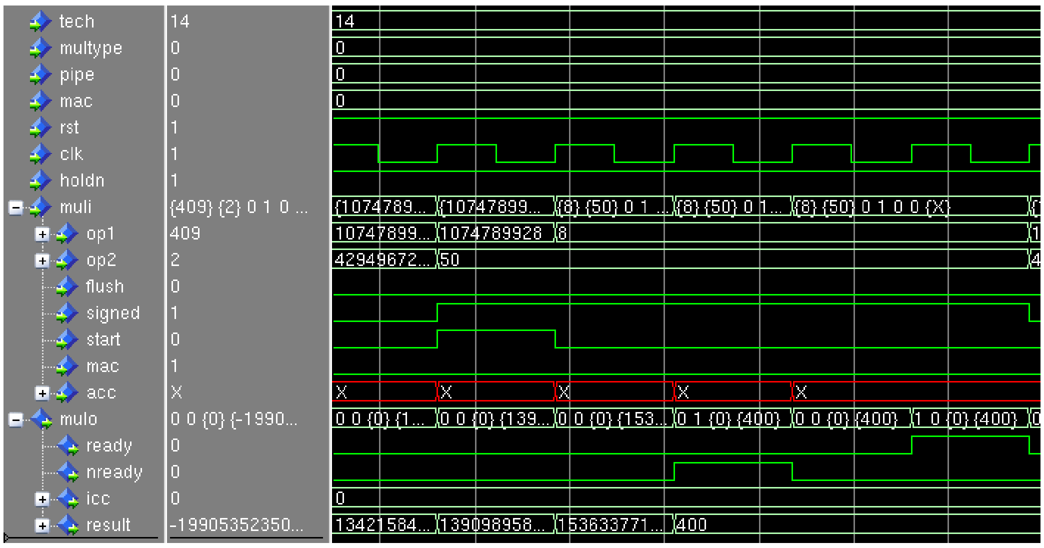


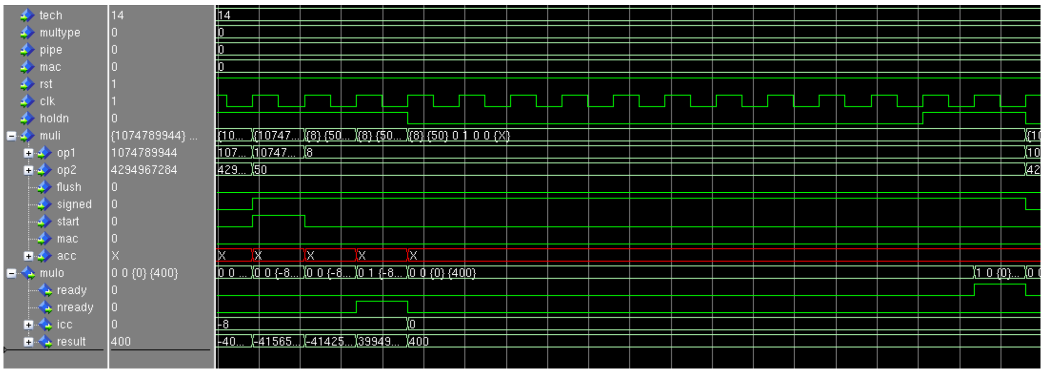
Figure 3.8: Screenshot of the Modelsim's Wave for the multiplier. For this particular simulation a 16x16 operation is shown, along with all the appropriate signals.

The last phase of the verification procedure was to run the Dhrystone testbench in Multisim to determine if the variables match the expected values. When this did not occur we compared the wave results (only the signals relevant to multiplication) of our design with the original's to look for discrepancies. This way it was possible to observe the correct behavior of the handshaking signals, markedly `start`, `holdn`, `ready` and `nready`. This type of "reverse engineering" was needed as the information in documentation does not concur with the implementation. In addition their behavior changes for different multiplier latency configurations (chosen through `make xconfig`).

For the final design we selected 4-cycles of latency[3] consequently we will only focus on the handshaking signals for this setup. Figure 3.9 features two wave signals from Modelsim.



(a) With constant `holdn`



(b) With varying `holdn`

Figure 3.9: Detail of the handshaking signals in the original multiplier for the Dhrystone benchmark.

Relying on the simulations we were able to determine that the operands are read at the falling edge of the muli.start signal. Also the ready signal is raised three cycles after that moment (assuming holdn does not change). The nready signal is set two clock cycle prior to ready. This information is summarized in the multiplier's state machine depicted in figure 3.10.

[3]It was necessary to perform this change otherwise the processor would not be able to handle the shorter latency of the modified multiplier.

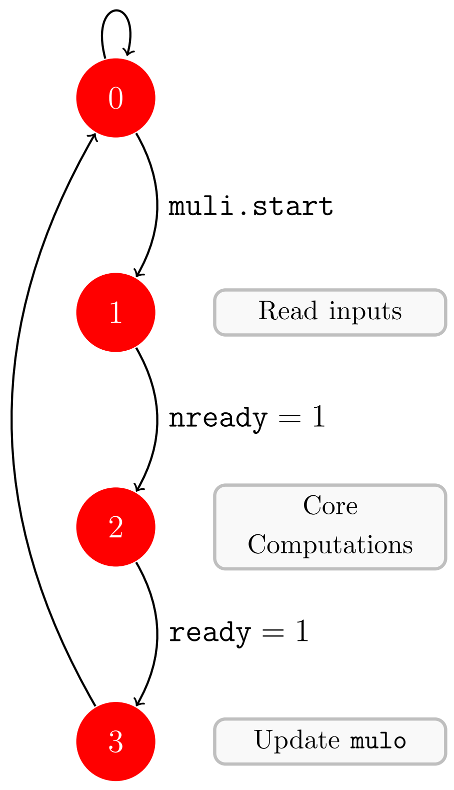


Figure 3.10: Diagram explaining how the various handshaking signals behave.

With this implementation we expect the benchmark results to improvement as the latency of the multiplier was reduced from 5 to 2 cycles. On the other hand, the area and power consumption will likely increase as the Wallace tree makes use of more resources. Some improvements related to better utilization of area are left for future work. In addition other modifications of this design which can have positive impact on metrics would have been interesting to implement and test. Examples include using a Dadda Multiplier (which uses less gates) instead of Wallace's, or using Booth recoding to decrease the size of the operands.

3.2. Divider

The algorithm implemented in the original version of the processor is one of the simplest but the slowest available: a radix-2 non restoring division.

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 because they make use of a whole multiplier to work which in itself is quite a big circuit. Similarly an array divider would have been very fast given we had control of the place&routing process in order to create a regular structure. However this is not possible in FPGAs so we discarded that option.

For these reasons we decided to implement a simple radix-4 division algorithm for simplicity of implementation and of the circuit itself.

Another possibility was to use an higher radix as this could have improved performance but the size of the lookup table required by the algorithm would have increased the area consumption. However the area itself is not a significant concert since, as stated previously, we are focused on power consumption and execution time. Notwithstanding a larger circuit means harder computations and thus more energy consumption.

The divider consist of a state machine (its diagram is shown in figure 3.11) which checks if the inputs will generate an overflow by doing a trial subtraction between the divisor and the MSBs of the dividend. Thereafter it performs a preliminary shift to put the divisor in the appropriate range in order to match the correct range for the look-up table to work properly.

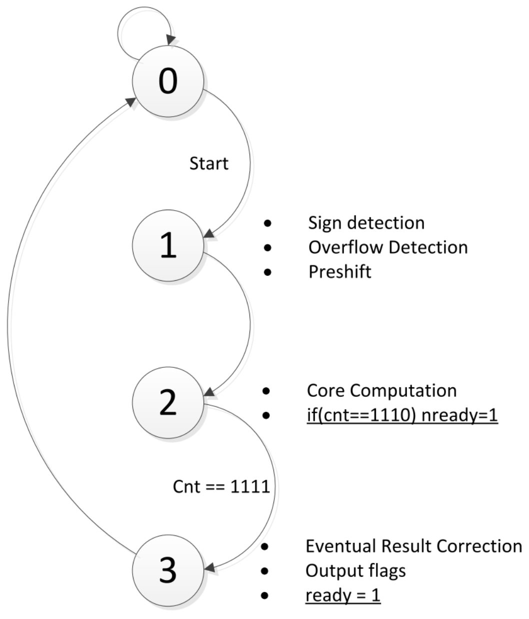


Figure 3.11: Divider State Diagram.

Figure 3.11 also shows when the ready signals are asserted. The documentation [4] explains the ready signal must be set the cycle before the result is stable, thus in the last state of the state machine. The reason for this is that once the machine returns to the idle state the registers are "frozen" and the results is stable. The `nready` signal must be asserted 3 cycles before the results is stable. This cycle is reached when during the core computation the counter's value is 14 ("`1110`"). By observing these signals the integer unit can find out when the operation is finished in order to eventually fetch consequent instructions or to make the pipeline proceed one step further.

After that, the real computation begins and lasts 16 clock cycles. The block diagram of the divider while it is in this state is shown in figure 3.12.

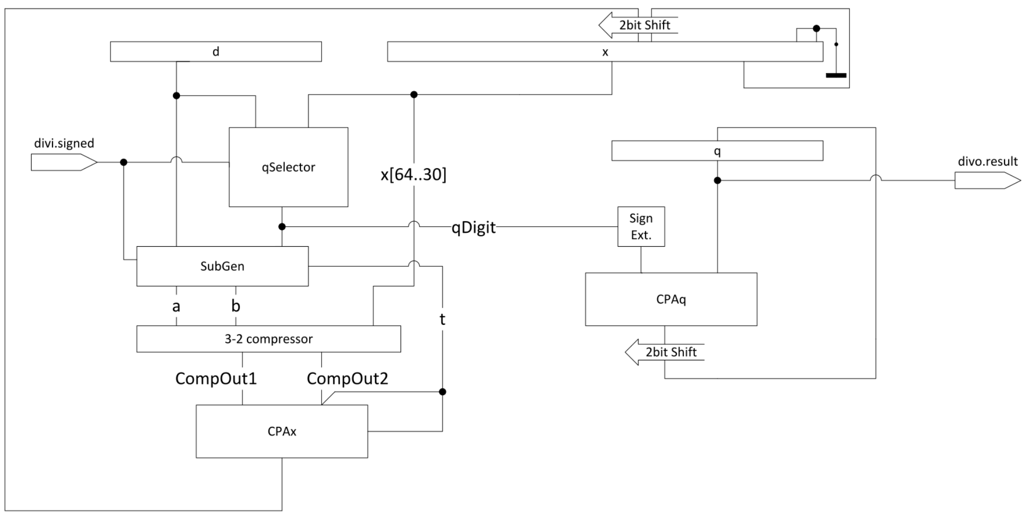


Figure 3.12: Divider Block Diagram (during Core Computation, State = 1).

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 3.13. In case of unsigned division only the right half of the p-d plot is being used.

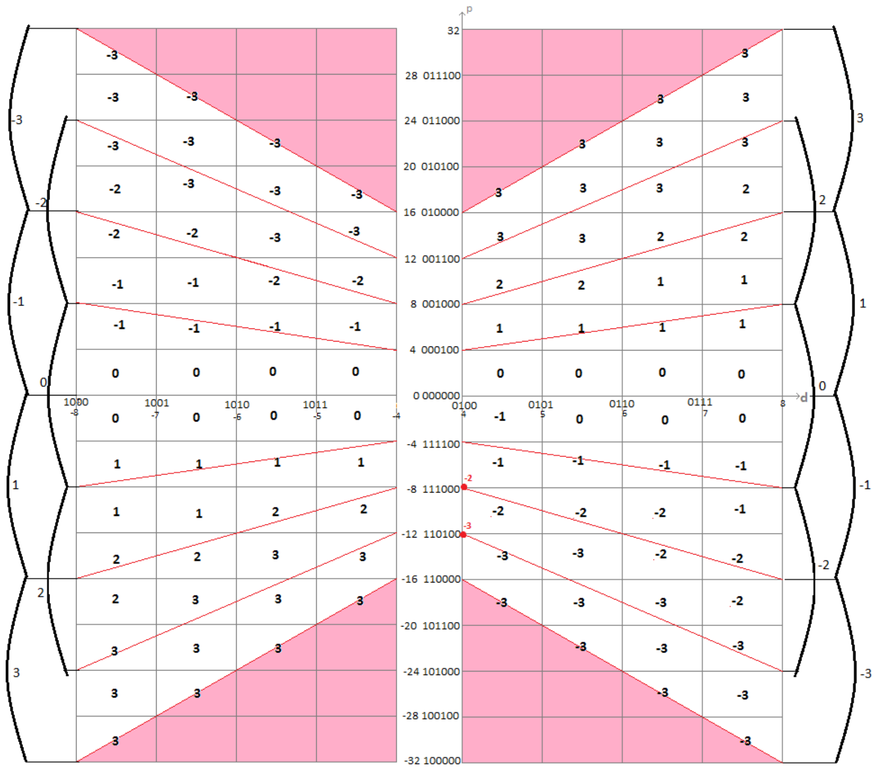


Figure 3.13: Radix-4 P-D Plot (Red Dots are Exceptions).

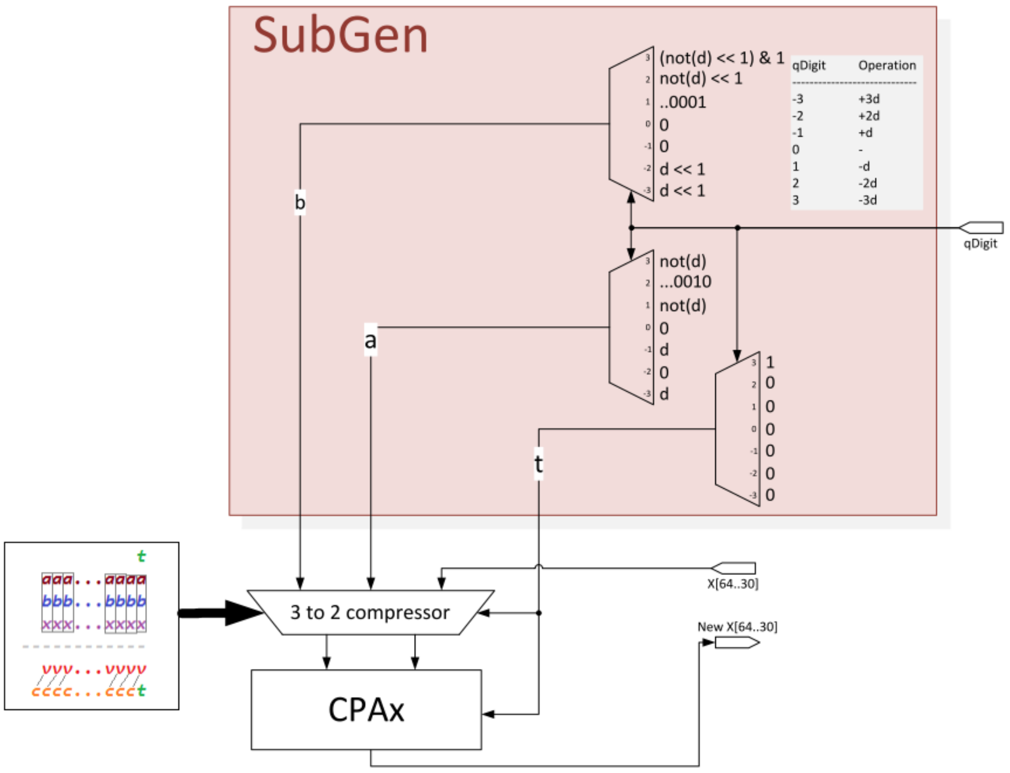
The quotient digits are in a radix-4 redundant format so a conversion to binary format is needed. The conversion is performed gradually every cycle by the 32-bit adder "CPAq" which shifts and sums each generated digit with the already calculated quotient.

An addition/subtraction in Carry-Save format would have been much faster and also easier for both the quotient and the partial reminder, but the selection of the quotient digit would have required the analysis of the most significant bits of both the sum and the carry of the reminder (x). This would make the lookup table several orders of magnitude larger and therefore consume more area.

In our divider "SubGen" generates the multiple of d to sum with the current partial reminder in a carry save format. All these operands are reduced by a 3-2 compressor (1 full adder of delay) and finally the new partial reminder is calculated with a 35-bit adder, "CPAx".

Using this architecture is possible to easily calculate the triples of d allowing us to have a simpler p-d plot and consequently a smaller look-up table. In particular the -3d, which is the most difficult, is calculated as the negation of 2d concatenated with a 1 plus the negation of d. The signal t is set to 1 so it can be summed twice in the compressor and as the carry input of CPAx. In this way its weight is 2 and calculation of the 2s complement of 2d is finished.

The principle behind the calculation of the multiples is explained better in figure 3.14.



qDigit	Operation
-3	+3d
-2	+2d
-1	+d
0	-
1	-d
2	-2d
3	-3d

Figure 3.14: Detail of SubGen (d multiples generator based of the qDigit).

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 resource 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 rest of the system, and its correct functioning, we designed a testbench. In this testbench we placed both the old version and the new version of the divider and the inputs are routed to the two units equally, so we expect to have the same outputs in both but with different latencies. Of course it's prohibitive to test the unit for all possible inputs therefore we determined a set of critical operations to test the design.

There are some categories of operations in this case, simple signed/unsigned division such as 350/100 or -350/100 which can be performed with the signed signal asserted or not, operations involving exceptions in the p-d plot, and operations generating overflows. The first category is not critical but just to be sure and have a complete testbench we implemented some of them.

As shown in figure 3.13, there are a couple of exceptions in the p-d plot. These exceptions are handled by the qSelector apart from all the other cases with dedicated lines of code (simple ifs). For instance one of these exceptions is when the divisor is "0100" and the dividend is "111000" followed by zeros (otherwise it would not be an exception because the point would be inside the uncertanties area and not in the bottom left corner). In this scenario the quotient digit must be -2 instead of -1 so one of these operations could simply be -16/4 for this operation after the pre-shift the input of qSelector would be exactly these ones.

For the operations involving overflows there are several possibilities where the dividend is very high and the divisor very low, enough to make the result greater than 2^{32}. There are also some operations that when considered to be signed don't generate overflows while if unsigned they do. This happen when the most significant bit of the dividend is one, its weight is 2^{63} when unsigned and -2^{64} when signed and so it can change significantly the absolute value of the result.

All of the aforementioned operations have been analyzed in the appropriate testbench, whose simulation are partially shown in figure 3.15. The result is always the same as the baseline version of the divider. The only exception occurs when there is an overflow. Here the result of the radix-4 divider is different from the baseline, sometimes the simulator displays an "X" because in an overflow the qSelector works in some of the not allowed areas. Nonetheless the overflow flag is asserted normally so the processor won't consider the actual result and consequently this difference does not constitute a problem.

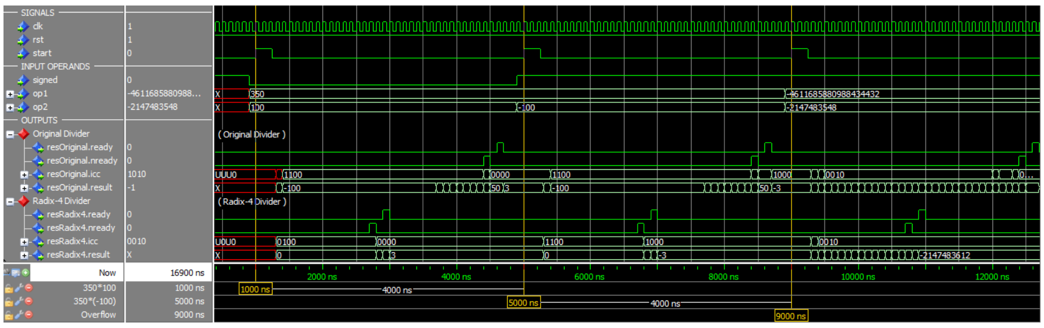


Figure 3.15: Signal Dump Of Radix-4 Divider Vs. Original Divider.

4. Results

4.1. Synthesis

In order to evaluate the performance of our improved processor we need first to play close attention to the starting point: the performance of the baseline version of the processor.

The evaluation of each design will be divided in two main categories

- Resources' Utilization (provided by Xilinx ISE)
- Benchmarks' Results (provided by the FPGA server)

The synthesis tool reported the values shown in table 4.1 for the resources utilization. Results for four different configurations are present which will allow us to better understand the impact of each modification in the area and power consumption. These are the baseline design, the design with the modified multiplier and original divider, modified divider and original divider and finally with both units modified.

One metric that is particularly interesting is "P/f" which reflects the energy efficiency of the processor. Another useful insight is the fact that the power consumption is almost proportional to the clock frequency, therefore we can use this value to estimate the power consumption at different clock frequencies.

In addition, 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_phy0/xc4v.ddr_phy0/casgen`.

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

The cummulative synthesis report with the aforementioned four configurations is summarized in table 4.1:

	Baseline	Multiplier	Divider	Both
Max clk freq. [MHz]	80.522	80.019	80.535	80.205
# of Occupied Slices	9904	10226	10479	10773
Total # of 4-input LUTs	16889	17398	17865	18329
Quiescent power [W]	2.467	2.467	2.468	2.469
Dynamic power [W]	0.721	0.726	0.743	0.756
Total power [W]	3.188	3.193	3.211	3.225
P/f [W/MHz]	0.03959	0.03990	0.03987	0.04021

Table 4.1: Resource Utilization

The frequency does not suffer considerable variations.

As expected the area consumption is moderately higher than before as the units we designed are more complex than the baseline. In particular the lookup table in the divider, and the Wallace tree structure in the multiplier are very space-hungry. This is reflected when comparing the last and second column, with area expansion of around 8%.

Furthermore the power consumption also augments. However the differences are more subtle, varying between 0.1% and 4.9%. The reason is similar, since the algorithms are more complicated more energy is expended for all the calculations. However this trade-off was envisioned at the start of the project, and as we will see later, in section 4.2, the benchmark's performance improved.

Finally the power freqeucy ratio summarizes the points made above. Since the power consumption is higher and the frequency is more or less the same, this metric will increase, although not too significantly.

4.2. Benchmark Scores

Once the processor is synthesized and has been loaded in a FPGA board, Linux is initiated on the processor along with several benchmarks. In table 4.2 the execution times of these benchmarks are reported, for detailed scores please see the accompanying Excel file. Unfortunaly we were not able to obtain results when uploading the multiplier only design (original divider) to the FPGA. This is strange as this multiplier unit worked before, in combination with the rewritten divider. Given this evidence we suspect the problem lays with the FPGA and not with the multiplier unit.

	Baseline	Divider	Both
Stanford [s]	2.30	2.26	2.23
Whetstone [s]	116.2	113.25	111.61
Gmpbench Multiply [Op/ s]	781	801	912
Gmpbench Divide [Op/ s]	15876	16335	19205
Gmpbench RSA [Op/ s]	5123	5284	5353
Division [s]	8.06	7.65	7.31
Mibench JPEG (average) [s]	23.215	22.465	21.815
SSD [s]	10.59	10.21	8.69
Total [s]	219.28	203.30	207.06

Table 4.2: Benchmarks Scores.

The scores obtained with the modified processor are reported in table 4.2.

As one can see introducing the divider has a widespread positive effect on the scores. The same can be said about the multiplier. Analyzing these results with respect to the worst area and power performance will be left for section 4.3.

From these results we can see that almost every benchmark had a healthy improvement, in the end the total execution time improved by 5.6%. It is interesting to observe that both units yield improvements to the benchmark scores. However for the total execution time introducing the multiplier has a detrimental effect. This is counter-intuitive given there were improvements in all the individual benchmark scores. The reason for this behavior is not clear to us.

These scores are not as good as expected but likely the execution of the operating system on the processor causes a non negligible overhead in the execution due to the scheduling. As an example, the divider takes about half the time to execute but the execution time of the division related benchmarks is only about 10% better.

4.3. 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 maximum 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.

For the area we are using the weighted formula to account for the true cost of our implementation instead of just the number of resources used. We make the simplistic assumption that the number of resources available on the device is inversely proportional to their effective cost. Thus we use their reciprocal as a weight to calculate the average. Therefore we obtain a metric which is likely similar to the actual cost of the implementation on a chip.

Moreover some composite metrics can be observed. These metrics consider two or more primitive metrics and are often more interesting than the latter because improvements in one metric are usually accompanied by decrease in other. Thus composite metrics show the overall performance, and give insight on the necessary trade-offs.

Since we want to speed-up the execution of the software while keeping the power consumption low as our target applications are embedded systems, the composite metric we are interested the most is P×BS. It reflects how much the processor is able to compute for the same unit of energy.

Additional composite metrics are A×D, A×BS and P×D. Since we focused on the execution time and power consumption one can notice that the other metrics are worse in our version compared to the baseline.

All the baseline's and modified units' synthesis and benchmark results have been condensed in table 4.3.

	Primitive Metrics			
	A	D	P	BS
Baseline	2.68×10^4	1.24×10^{-2}	3.19	2.19×10^2
Modified (Div)	2.83×10^4	1.24×10^{-2}	3.21	2.13×10^2
Modified (Mul&Div)				
Improvements (Div)	5.8%	-	0.7%	-2.7%
Improvements (Mul&Div)	%	%	%	%

	Composite Metrics			
	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 (Div)	3.52×10^2	6.05×10^6	3.99×10^{-2}	6.85×10^2
Modified (Mul&Div)				
Improvements (Div)	5.8%	2.9%	0.7%	-2.0%
Improvements (Mul&Div)	%	%	%	%

Table 4.3: Final Metrics For Baseline And Improved Versions.

5. Conclusions and further improvements

The improvements made to the arithmetic units have improved the benchmark performance of the processor. Although they are modest, they reflect the chosen target scenario.

Because of lack of time we were unable to perform more changes, but of course there are many things to change in the architecture to improve further the performance.

Different configurations for the multiplier should have been tested to decrease its impact on power and maximum frequency.

The size and structure of the cache memory could be changed to decrease the probability of misses and consequently the benchmarks execution time. Although this could have been done using the configuration tool it would have not been fruit of our own merit. Moreover increasing the cache size probably would have increased also the power consumption, escalating even more this issue.

The LEON3 uses a static branch prediction in the integer unit, which is a good compromise between power consumption, as no difficult computation is needed, yet there are gains in terms of execution time. To improve performance further, a 1 bit or a 2 bit branch prediction buffer algorithm could be implemented. The calculations needed are more intricate and computed more often (30% of the instructions are branches) henceforth the power consumption would probably increase. On the other hand the gains in terms of execution time could make up for the loss.

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 it 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