**TU Delft**

13

**ET4171 Processor Design Project**

**LEON3 processor optimization**

**Luca Feltrin, Henrique Dantas**

# 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 application.

Our main target is to decrease the computation time for certain benchmarks keeping the power consumption as low as possible, so the main compound metric we are going to care of 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 this arithmetic cores, some easy algorithms can be implemented to have a real improvement.

Some other changes are described at the end of this document.

# Improved Arithmetic Cores

## Multiplier

## 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. 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 look-up table required by the algorithm would have increased again the area consumption.

The divider consist in a state machine (its diagram is shown in Fig. 1) which check if the inputs will generate an overflow and performs a preliminary shift to put the divisor in the appropriate range to be computed correctly.

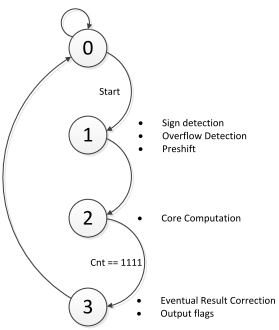


Fig. 1 Divider State Diagram

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 Fig. 2.

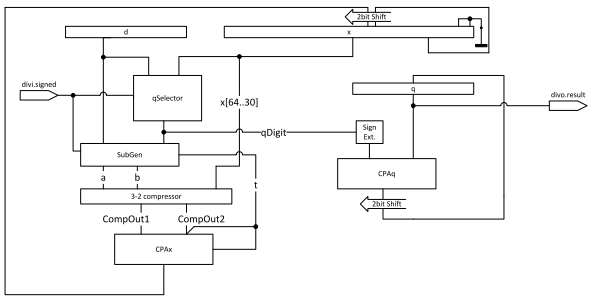


Fig. 2 Divider Block Diagram (during core computation, state=1)

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 look-up table which perform the quotient digit guessing and it’s based on the p-d plot of the radix-4 SRT division shown in Fig. 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 in 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”.

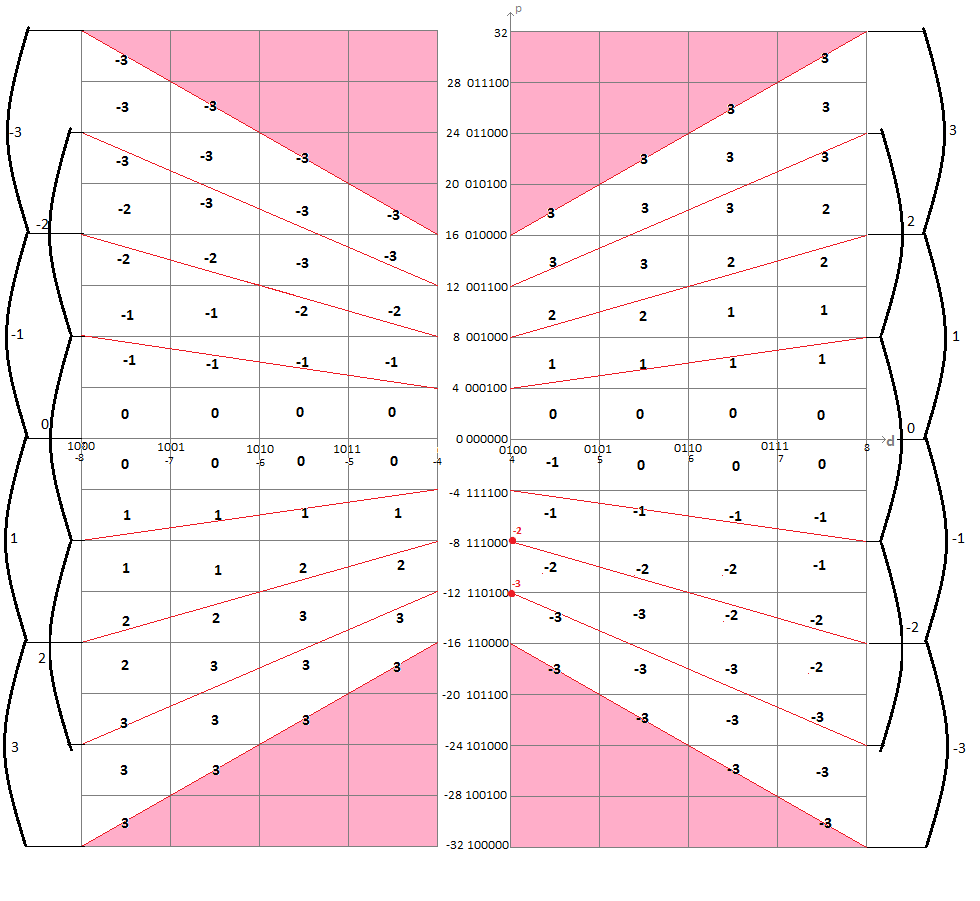


Fig. 3 Radix-4 p-d plot (red dots are exceptions)

Other solutions have been analyzed, such as having a look-up 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 a lot while one more cycle would have been needed so we decided to keep using this divider.

In the end 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.

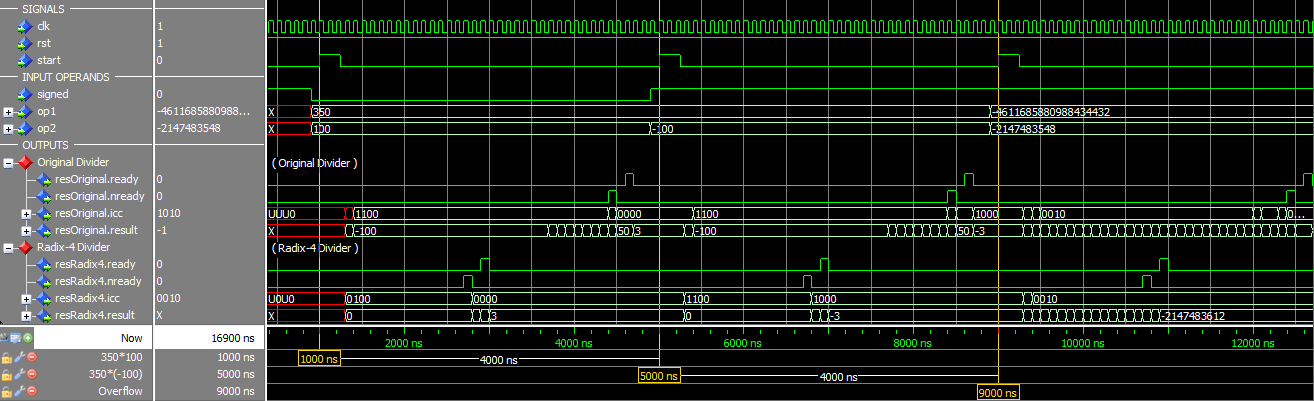


Fig. 4 Signal dump of radix-4 divider vs. original divider

# Results

## Synthesis

## Benchmark Scores

## Metrics confrontation

# Conclusions and further improvements

(FURTHER IMPROVEMENTS)

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.

# Attached documents

## Baseline

## Upgraded version