Parallel Numerics '05, 193-206 M. Vajteršic, R. Trobec, P. Zinterhof, A. Uhl (Eds.)
Chapter 7: Systems and Simulation ISBN 961-6303-67-8

A High-Performance Data-Dependent Hardware Divider

Rainer Trummer[1,*], Peter Zinterhof[1], Roman Trobec[2]

[1] University of Salzburg, Department of Scientific Computing
Jacob Haringer-Str. 2, A-5020 Salzburg, Austria

[2] Jožef Stefan Institute
Jamova 39, SI-1000 Ljubljana, Slovenia

Hardware dividers are needed in many areas of applications like computer floating-point units, communication systems, cryptography, signal processing, etc. The performance requirements of these applications differ regarding data and architectural issues. In this paper, the basic principles used in hardware integer dividers are shown. A hybrid data-dependent divider is proposed based on several improvements that speedup division on average for 600%. Data-dependent performance was simulated on a parallel computer cluster in order to search a larger number of cases. Different principles for the generation of random numbers were used to emphasize potential advantages or drawbacks of the proposed dividers.

1 Introduction

Division is the most complex of the four basic arithmetic operations and, in general, does not produce an exact answer, since the dividend is not necessarily a multiple of the divisor. Therefore, the correct quotient and remainder are usually obtained through performing a sequence of iterations until the desired precision is reached. This procedure is called *sequential division* and serves as the basic principle for many practical implementations [1, 3, 4].

Based on sequential division, the most prevalent representatives are the so-called *radix-r* dividers, where r denotes the radix, typically chosen to be a power of 2. In order to compute the answer, these dividers perform a

*Corresponding author. E-mail: rtrummer@cosy.sbg.ac.at

constant number of iterations, depending on the used radix. Several concepts have been developed to speedup the exhaustive sequential process. The most efficient method is called *SRT division*, named after their inventors Sweeney, Robertson, and Tocher, who had the same idea about the same time [3, 4, 7, 10]. The main principle is to interpret the divisor D and all partial remainders R_i as normalized fractions that are assumed to satisfy the relations $1/2 < |D| < 1$ and $1/2 < r|R_i| < 1$, where r is the used radix. Depending on the underlying *redundant quotient digit set*, which can be generalized as $Q_i \in \{-m, \ldots, -1, 0, 1, \ldots, m\}$ with $(r-1)/2 \leq m \leq r-1$, a certain number of significant digits of the divisor and partial remainder is used to select an appropriate quotient digit from a look-up table. This selected redundant quotient digit can be converted to the corresponding binary digits *on the fly*, which avoids the need of additional storage place.

The main drawback of SRT division is the need of large look-up tables that grow quadratically with increasing radix. Depending on the radix used and quotient digit set, look-up tables can consist of many thousand to millions of entries. Moreover, generating all of the required divisor multiples for radix 8 and higher is difficult and raises the cost of additional hardware. Due to this, practical table implementations are restricted to radix 2 and radix 4 [7, 10]. Another drawback is that during the development process of a large look-up table a few bits can easily get lost, a consequence that has become very popular as the *Pentium Flaw* [13].

However, the search for more efficient solutions is greatly encouraged by the rapidly increasing demand for small and fast integer dividers, needed in many areas of applications like communication, cryptography, signal processing, etc. [8, 9, 14]. One of them are *data-dependent* dividers that execute in variable time. The basic principle is to skip all redundant operations and carry out only shifts as long as there are leading zeros of the remainder or divisor, depending on the algorithm used [5]. This method is called *shifting over zeros* and also used in SRT division for normalizing the remainder after each iteration [2, 3, 4, 7]. Data-dependent dividers do not require any look-up tables and, in general, achieve a much higher throughput than standard radix-2 dividers. Therefore, they are primarily incorporated in small architectures like signal processors. Unfortunately, the speed of data-dependent dividers is currently not adequate to be also competitive concerning fully pipelined architectures.

Compared to tons of papers that have been published about SRT division and related methods [9, 11, 12], very little is available about data-dependent divider architectures. This was our main motivation to contribute some work in this field. Referring to the insufficient speed of data-dependent dividers

based on the shifting-over-zeros method, our main contribution is a more sophisticated hardware divider that achieves an average speedup of 600%.

Section 2 introduces two basic divider concepts, which we implemented for reliable comparisons with two improved dividers presented in Section 3. The environment and methods used to analyze the performance are described in Section 4. The obtained results are discussed in Section 5, followed by conclusions and a brief outline of future work.

2 Division Arithmetic

The most simple representative of sequential division is the so-called *Radix-2 divider* (R2), which can be interpreted as the binary version of the classical *paper-and-pencil method*. It is also referred to as performing *long division*, since a $2n$-bit shift register (SR) is used to hold an n-bit dividend and form an n-bit quotient. The dividend is placed initially in the n low-order bits of the register and shifted left 1 bit per iteration. The free least significant bit (LSB) is then used to store the new quotient bit. Upon completion, the SR contains the remainder in its high-order word (HW) and the quotient in its low-order word (LW). The iterative part consists of three basic steps: (1) shift left the register for 1 position, (2) subtract the divisor from the remainder, (3) store the result as HW and store the new quotient bit as the LSB.

A demonstration of dividing $7 = 0111_2$ by $3 = 0011_2$, producing quotient $2 = 0010_2$ and remainder $1 = 0001_2$, is given in Table 1. The described division method implemented by the R2 is generally known as *restoring division*, since the remainder must be restored if the subtraction yields a negative result. The restoring step can be avoided in two ways, either by a method known as *non-restoring division*, which does not offer any significant advantage, or by an elaborated implementation illustrated schematically in Figure 1. The multiplexor subsequent to the adder chooses between the result representing the new partial remainder or the preceding one, depending on the adder's carry-out signal (CO), which is also used to set the new quotient bit. Note that subtraction is implemented as 2's complement addition with the carry-in signal (CI) set to 1, and therefore, referred to as *addition* throughout the rest of the paper. If subtraction yields a negative result, 2's complement addition underflows, indicated by CO=0, and restoring of the remainder is achieved by rewriting the previous one. In Figure 1 right, a demo calculation for $0111_2/0011_2$ is shown in binary presentation. Columns show iteration number, operation type (I initialization, + addition, 0 or 1 CO, W write, and $\leftarrow$ shift-left), HW and LW of the SR, and some comments denoted by //, respectively.

Step	HW	LW	Explanation
initialize	0000	0111	set HW to 0, set LW to dividend
shift-left	0000	111x	
subtract	−0011		HW − divisor = −0011
	−0011	111**0**	result is < 0, set LSB of LW to 0, restore HW
restore	0000	111**0**	HW + divisor = 0000
shift-left	0001	11**0**x	
subtract	−0011		HW − divisor = −0010
	−0010	11**00**	result is < 0, set LSB of LW to 0, restore HW
restore	0001	11**00**	HW + divisor = 0001
shift-left	0011	1**00**x	
subtract	−0011		HW − divisor = 0000
	0000	1**001**	result is ≥ 0, set LSB of LW to 1, no restoring
shift-left	0001	**001**x	
subtract	−0011		HW − divisor = −0010
	−0010	**0010**	result is < 0, set LSB of LW to 0, restore HW
restore	**0001**	**0010**	HW + divisor = 0001

Table 1: Demo of restoring division for $0111_2/0011_2$ in binary representation.

At the first glance, the additional multiplexor appears to stretch the critical path and hence to increase the execution time. Actually, the opposite is the case. The SR basically consists of flip-flops and 2-to-1 multiplexors that implement write or shift operation. In case of a 1-bit shift-left, the actual shift is hard-wired by feeding back the $n-1$ high-order bits combined with a constant zero-LSB. Referring to the divider, shown in Figure 1, two serial 2-to-1 multiplexors are combined in a 4-to-1 multiplexor that implements in a single step either initialize, shift, write, or write-shift operation. Due to this technique, the iterative part of the divider reduces to an addition followed by a write operation. Initialization places the dividend in bits $n, \ldots, 1$ rather than $n-1, \ldots, 0$, saving an initial shift-left, and consequently 1 complete iteration. The number of remaining iterations is determined by a counter, which is a part of the control logic.

If no exceptions occur, the divider always performs n iterations independently of the given operands. The great advantage is the constant execution time, thus making such dividers very attractive for fully pipelined architectures. On the other hand, dividers based on this concept are obviously very inefficient if operands are relatively small, because then there are many initial useless iterations. To increase the efficiency of the R2 without changing the basic architecture, we applied a small modification for initial normalization of the dividend. Precisely, the LSB of the HW forces skipping of additions until the most significant bit (MSB) of the dividend, stored initially in the LW,

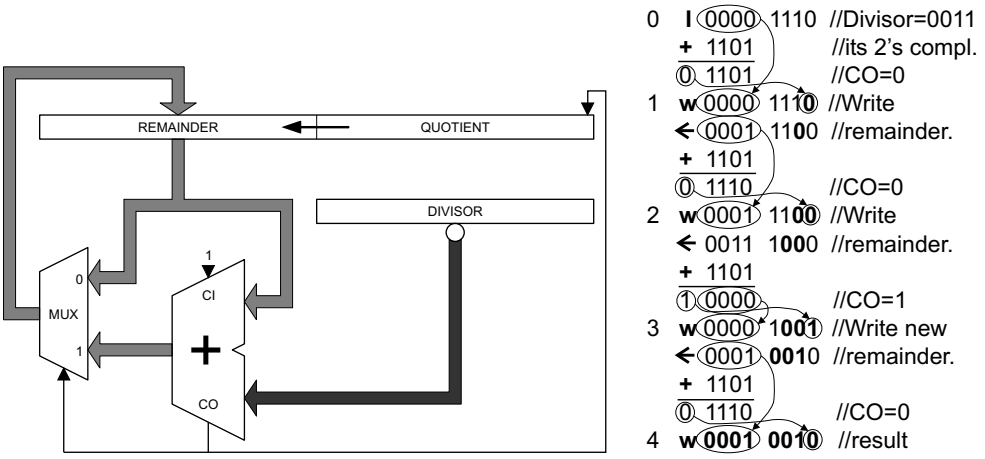

Figure 1: Schematic of Radix-2 divider (left) and demo division $0111_2/0011_2$ in binary (right).

reaches the HW. We named this modified version *Radix-Two divider* (RT) to distinguish it from the original R2.

An alternative to achieve better performance are data-dependent dividers. Figure 2 shows the schematic of the *Self-Aligning divider* (SA) [5]. Our demo example is given again with the same notation as in Figure 1. Columns show now the iteration number, operation type, the SR with HW and LW, and dividend, placed initially in the remainder storage, respectively. The basic architecture of the SA is similar to the RT, except that the SR is bidirectional and some additional simple control logic is introduced to disable over-shifting and indicate the last iteration. The main difference compared to the previous concept is that the divisor is shifted instead of the remainder. If the dividend occupies more bits than the divisor, the divisor is *aligned* first by left shifting to meet the dividend's data length. The shifted divisor is compared with the dividend, using the existing adder and its CO to test if the result is negative and alignment is finished. Due to this alignment, a quantity representing the divisor multiplied by 2^k, where k is the number of skipped iterations, is subtracted from the remainder in a single step. Now, the shift direction is reversed to begin the same add-write-shift process as described by the RT.

The bidirectional SR is implemented by a 3-to-1 multiplexor that realizes initialize, shift-left, or shift-right operations that can be done in parallel with writing the remainder. Regarding the critical path, the iterative part is similar to the RT. The new quotient bit is stored as the MSB of the LW, which implies that the quotient grows from left to right. Upon completion, the LW contains

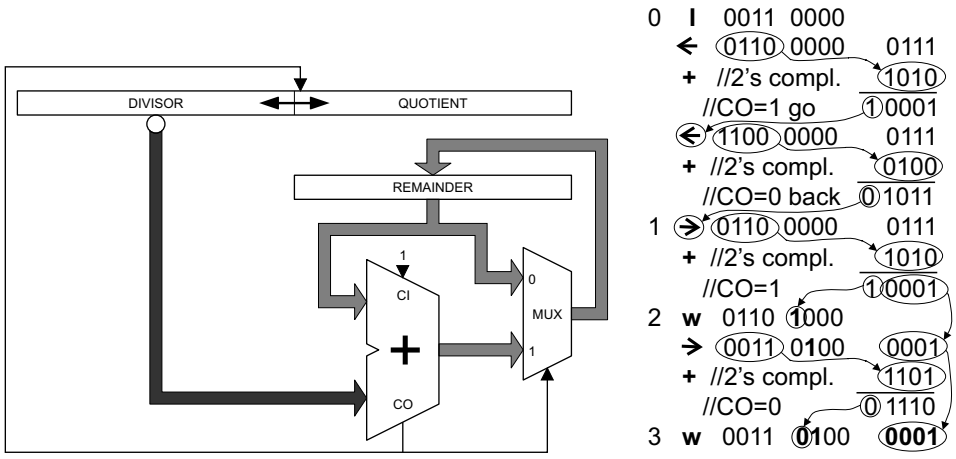

Figure 2: Schematic of Self-Aligning divider (left) and demo division $0111_2/0011_2$ in binary (right). Notation is the same as in Figure 1.

the quotient in *reversed* order.

The total number of performed iterations, and hence the execution time, depends exclusively on the input data. Although the SA requires more cycles in the worst case (a tiny divisor will be shifted a long way left first and then back all the way) than the RT, the average throughput of the SA is generally much higher.

3 Improved Division

The main drawback of the previous SA is that it only shifts 1 bit per iteration, independently of the current shift direction. Thus, if the operands differ largely in their data lengths, many iterations are required for aligning the divisor. Due to this, the basic idea was to provide a correctly aligned divisor at each iteration. In fact, this can be done efficiently by using two logical shifters modified for this special purpose. A logical shifter that can shift an n-bit input up to $n-1$ positions basically consists of n n-to-1 multiplexors and a common k-to-n decoder, where $k = \log_2 n$. The decoder is implemented by an n-bit *priority encoder* that converts the k-bit shift count to an n-bit 1-hot selection signal, i.e., all trailing bits after the leading non-zero bit are set to zero. This selection signal is supplied to the multiplexors that implement appropriate logic wiring and consequently k-bit shifting. The desired behavior of aligning the divisor correctly at each iteration is achieved by connecting two such modified logical shifters, one for each direction. Such a combinational

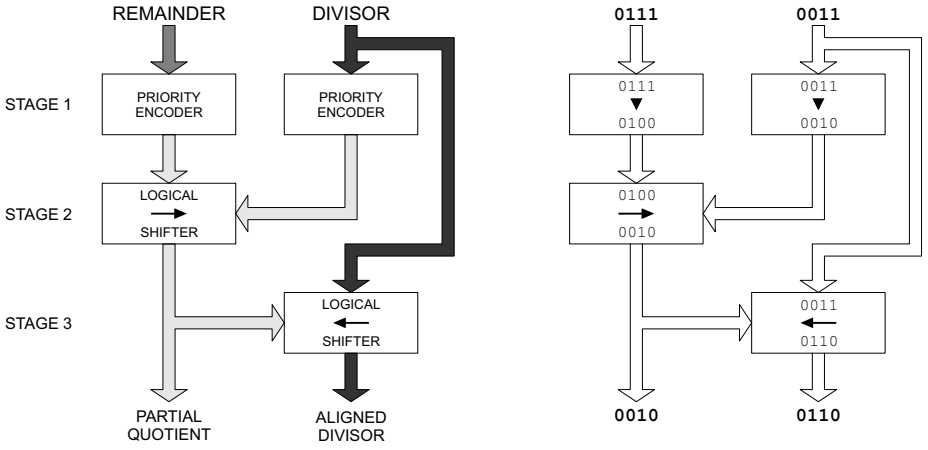

Figure 3: Schematic of Aligner (left) and demo with 4-bit operands 0111_2 and 0011_2 in binary (right).

unit, called *Aligner*, provides both, the aligned divisor (AD) and the corresponding partial quotient (PQ), as demonstrated in Figure 3. The Aligner's operational sequence can be separated into three stages: (1) convert the input data to 1-hot signals, (2) generate PQ, and (3) generate AD.

Referring to the right part of Figure 3, again with 4-bit demo operands, in stage 2, the remainder's 1-hot signal is shifted right 1 bit according to the divisor's 1-hot signal and producing immediately the partial quotient. In stage 3, the divisor is shifted left 1 bit according to the partial quotient's 1-hot signal, producing the aligned divisor. The generated divisor is always aligned correctly to the remainder regarding data lengths, however, both operands might contain any possible combination of trailing bits after the leading non-zero bit. This implies that the AD can be either less, equal, or greater than the current remainder, which causes an addition overflow in the latter case. Fortunately, an addition with an AD shifted right by 1 bit can never overflow. The described concept is integrated in the *Direct-Aligning divider* (DA) that uses two adders in parallel, as can be seen in Figure 4. Adder 1 is supplied with the full n-bit inverted output as second operand, whereas Adder 2 is supplied with a constant non-zero MSB combined with the $n-1$ high-order bits of the inverted output (hard-wired shift-right). The CO signal of Adder 1 is used to control the two multiplexors. In case the primary addition underflows, Mux 1 selects the result of Adder 2 and Mux 2 selects the shifted right PQ, which is then or-ed with the current quotient. In the present version of the DA, an additional comparator is used for determining the last iteration, since this is the most convenient way and it also keeps the controller simple. Nevertheless,

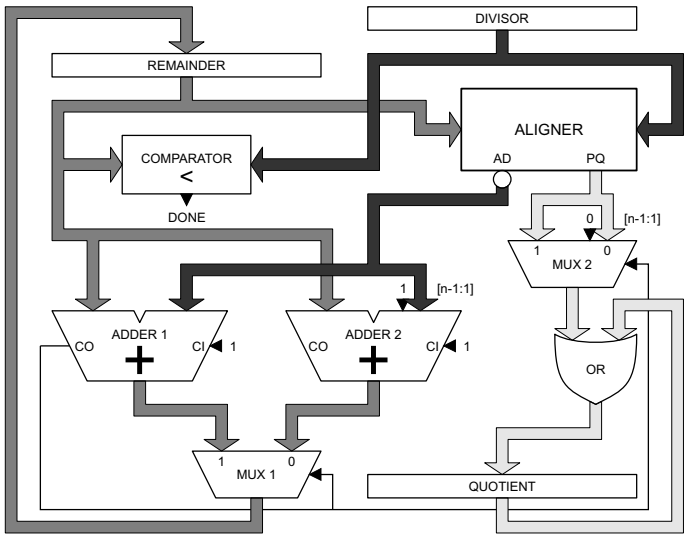

Figure 4: Schematic of Direct-Aligning divider.

it can be replaced by some more advanced control logic that evaluates the stopping-specific criteria.

Checking our previous demo, the aligner generates the PQ and AD directly, as shown in Figure 3. The sum of the AD's 2's complement and the remainder, initialized with the dividend, results in 0001_2 and CO=1, which implies that the new remainder is 0001_2. Because the remainder is less than divisor, the division is finished. From Figure 4 it also follows that the final quotient is equal to the PQ, which evaluates to 0010_2.

The architecture of the DA looks somewhat strange, and in fact, unusual compared to other dividers. However, in the theoretical sense, the DA represents an optimal solution concerning data-dependent divider architectures. It reduces the process of sequential division to the *minimum* number of subtractions required for breaking down a given dividend. Although the average performance could be improved greatly compared to the previous SA, the overall performance is still insufficient. The main drawback of the DA is that the adders are connected in series to the Aligner, which largely extends the critical path delay.

However, in case of the DA we could solve this problem at the cost of one additional multiplexor and two temporary registers (TR). The resulting divider, named *Hybrid-Aligning divider* (HA), is shown in Figure 5. The underlying concept was to combine the strength of the DA with a distinct worst-case behavior. Precisely, whenever the operands differ largely in their

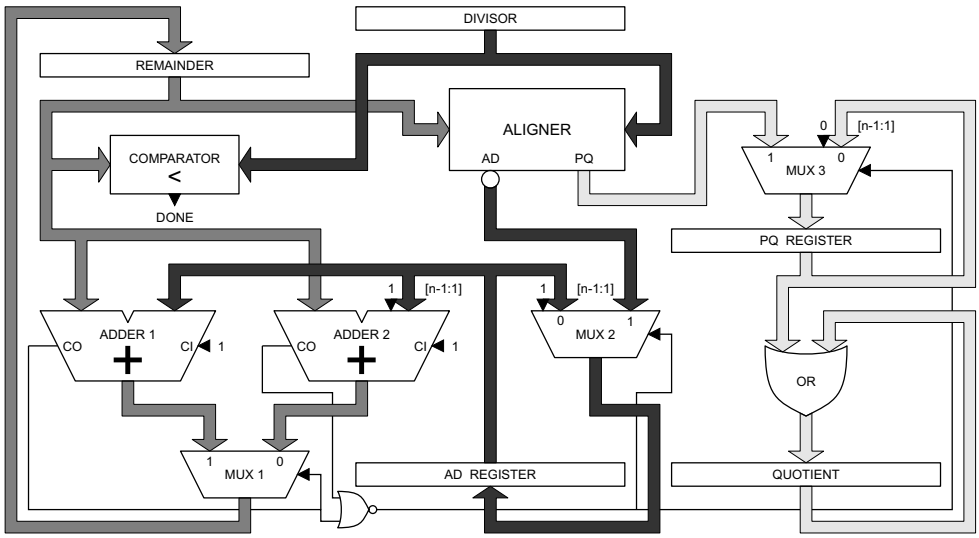

Figure 5: Schematic of Hybrid-Aligning divider.

data lengths, it is definitely more efficient to involve the adders in a short iterative path, similar to the RT or SA, rather than involving them in the extended Aligner path. For this purpose, we added two TRs that can perform a hard-wired shift-right through a multiplexor. These TRs are loaded with the first AD and PQ as soon as they are available. From this point, the divider uses the contents of the TRs, which are written, and hence shifted, after each addition. The CO signal of Adder 1 is used to control Mux 1, similar to the DA. If the primary addition overflows (CO=1), the controller forces the quotient to be written before the PQ, hence the current content of the PQ-TR is or-ed with the quotient. In case the primary addition underflows (CO=0), the PQ is written first, which implies a shift-right of the PQ-TR before its content is or-ed with the quotient. The CO signals of both adders are nor-ed to control Mux 2 and Mux 3 that are used for writing and shifting the two TRs. The adders are involved in the short critical path using the content of the AD-TR until the secondary addition also underflows. Whenever this happens, the TRs are reloaded with the operands provided by the Aligner. This causes an iteration without obtaining a partial remainder, however, a gap that will be filled immediately during the next iteration.

Testing our demo again, the calculation is exactly the same as for the DA in this simple case. Using longer and more complicated operands containing several zeros would reveal the real advantage of the HD. Due to the extension proposed, the DA efficiency could be improved significantly. The resulting HA

clearly dominates over all analyzed dividers in both worst-case and average performance.

4 Performance Analysis

In order to test the theoretical behavior, we developed synthesizable architectures of the four discussed dividers. For this purpose, Verilog HDL (Hardware Description Language) in combination with the Xilinx Project Navigator 6.2.03i environment configured for the Spartan xc3s50-5pq208 FPGA (Field-Programable Gate Array) [5, 15] was used. To gain full control over each implementation detail and make the synthesis as technology-independent as possible, we created a set of components based exclusively on RTL (Register Transfer Level) descriptions. In other words, each component was described by some structure of basic gates and their connectivity.

The overall performance of the four dividers depends primarily on the underlying algorithms, since they are fully data-dependent. We found out that the encompassed logic delays influence the simulation results only by a linear factor. Thus, to implement just 4 instead of 16 possible implementations of optimized controllers for different logic delays and different word lengths, the 64-bit delays were used throughout all simulations, since most related work is based on the same data. Each individual divider was tested for the architecture sizes 16, 32, 64, and 128 bits.

The best/worst-case analysis was conducted directly with the Xilinx Model Simulator XE II 5.7g, whereas the average-case analysis required a much more elaborate method. Obviously, to obtain the average performance one would merely need to collect all results and calculate the average. Unfortunately, this is currently only possible for small word lengths, because even for 32-bit operands the total number of combinations cannot be tested in a meaningful amount of time. According to this, we developed a C++ parallel program based on MPI (Message-Passing Interface), which gave us the opportunity to involve a large number of processors in the simulation [6]. The behavior of each divider was mapped to a *performance function*, designed to return the exact number of required *clock cycles* for a given pair of operands. During program execution, the four performance functions were called in subsequent order with the same operands, which were generated randomly.

The parallel simulation program is based on a master-slave principle, where the master processor distributes the input parameters (word length, test duration, etc.) among the slave processors and lets each of them run its part of the simulation independently until all partial results are calculated and gathered in the master processor. Thus, communication time is not crucial. The

simulation is based on random numbers generated locally on each processor using different local seeds equal to the unique processor's rank. To obtain results that are not influenced by any exceptional cases, each operand pair was incremented immediately by 1 to avoid zero-values. In case the dividend was less than the divisor, the operands were exchanged.

Every data-dependent divider has its own characteristic behavior. Thus, for evaluating the average performance it is definitely not sufficient to apply *any* series of random operands, since the performance of one divider might be excellent, whereas the performance of another one might be poor, or contrary. To cover a large range of possible operand combinations, we developed a multi-word RNG (Random Number Generator) that allows some variations regarding the distribution. This RNG was designed to use the processor's built-in RNG for generating multi-word random numbers of any desired word length, based on XOR operations, followed by a random shift-right of the produced number. Due to this adjustable random shift-right, the distribution of generated numbers could be varied to enable different simulations. To explore each divider's behavior affected by different distributions, we used the following four RNG random shift-right ranges: (1) $0,\ldots,n/8-1$, (2) $0,\ldots,n/4-1$, (3) $0,\ldots,n/2-1$, and (4) $0,\ldots,n-1$, where n denotes the word length in bits. We expect that multi-word upgrade and dividend-divisor separation have ensured that eventual correlations of pseudo-random sequences cannot affect the performance results.

5 Obtained Results

The minimum and maximum performance results for all four dividers were obtained through simulations with analytically determined best and worst cases. Note that the worst case is not always given by dividing the largest representable value by the smallest one. The average-case results were obtained by running the parallel program on a 16-node AMD OPTERON 244 (1.8GHz) dual processors, connected in a 2-D toroidal 4-mesh by Gb Ethernet, using MPI library and C++ under Linux Fedora2.

Against our expectations, the average values converged very fast, usually within a few million divisions. Nevertheless, a minimum of 35 billion divisions was executed during each simulation to prove stability. A summary of all conducted measurements is given in Table 2. Note that due to the supplied clock frequency of 1 GHz, the number of required *clock cycles* is identical with the execution time in *nano-seconds*. It points out that the RT divider covers the smallest spectrum of all four dividers. This is due to the fact that its execution time is constant in the original R2 version and the applied shifting-over-zeros

method could cause only a slight improvement of the average performance, which is very close to the maximum. The SA stands out from all others with the widest average spectrum, since it is highly sensitive to strongly varying distributions. In other words, a sufficient average performance can only be achieved with operands that do not differ too much in their data lengths. The DA spans the largest spectrum due its very long iterative path. However, its average spectrum is much smaller and lower compared to the SA. The HA stands out from the others with the smallest and lowest average spectrum. In this sense, it is least sensitive concerning strong variations of input data.

16-bit Performance [*clock cycles*]						
Divider	Minimum	RNG 1	RNG 2	RNG 3	RNG 4	Maximum
RT	58	150	145	135	116	163
SA	23	57	64	81	116	270
DA	26	34	37	47	68	311
HA	26	30	33	39	52	161

32-bit Performance [*clock cycles*]						
Divider	Minimum	RNG 1	RNG 2	RNG 3	RNG 4	Maximum
RT	106	305	295	275	238	323
SA	23	64	82	122	205	534
DA	26	37	47	71	118	615
HA	26	33	39	56	89	315

64-bit Performance [*clock cycles*]						
Divider	Minimum	RNG 1	RNG 2	RNG 3	RNG 4	Maximum
RT	202	615	595	558	483	643
SA	23	82	122	209	382	1062
DA	26	47	71	120	219	1223
HA	26	39	56	92	163	635

128-bit Performance [*clock cycles*]						
Divider	Minimum	RNG 1	RNG 2	RNG 3	RNG 4	Maximum
RT	394	1235	1198	1123	973	1283
SA	23	122	209	384	734	2118
DA	26	71	120	221	422	2439
HA	26	56	92	166	311	1275

Table 2: Measured performance of four dividers in number of clock cycles.

Finally, Table 3 summarizes the average speedups resulting from comparisons against the constant execution times of the R2, which are equal to the maxima of the RT. Each entry was determined by dividing the RT maximum by the average of the four obtained simulation values RNG 1 through RNG 4 from Table 2.

Average Speedup				
Divider	16 bit	32 bit	64 bit	128 bit
RT	1.2	1.2	1.1	1.1
SA	2.1	2.7	3.2	3.5
DA	3.5	4.7	5.6	6.2
HA	4.2	6.0	7.3	8.2

Table 3: Average speedup of four dividers compared to R2.

6 Conclusions

We have discussed the basic principles and two fundamental concepts of sequential division. We have further introduced the Direct- and Hybrid-Aligning divider, which represent two possible implementations of the proposed Aligner concept and might serve as inspiration for more sophisticated designs. In our effort to improve the efficiency of data-dependent dividers, we have developed an elaborate method that offers an average speedup of 600%. The underlying architecture is fully scalable, and therefore, might become attractive compared to other divider architectures that can only be scaled to some extend.

Different designs of the Aligner applied to both existing an new divider architectures are expected to be investigated in the future. Another important goal is to determine more accurate and technology-specific performance data based on fully optimized implementations.

References

[1] David A. Patterson and John L. Hennessy, *Computer Organization & Design*, 2nd ed., Morgan Kaufmann, San Francisco, California, 1998.

[2] John L. Hennessy and David A. Patterson, *Computer Architecture—A Quantitative Approach*, 2nd ed., Morgan Kaufmann, San Francisco, California, 1996.

[3] Israel Koren, *Computer Arithmetic Algorithms*, 2nd ed., A K Peters, Natick, Massachusetts, 2002.

[4] Mi Lu, *Arithmetic and Logic in Computer Systems*, Wiley-Interscience, Hoboken, New Jersey, 2004.

[5] Michael D. Ciletti, *Advanced Digital Design with the Verilog HDL*, Prentice Hall, Upper Saddle River, New Jersey, 2003.

[6] Ian T. Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1995.

[7] Michael J. Flynn and Stuart F. Oberman, *Advanced Computer Arithmetic Design*, Wiley-Interscience, New York, New York, 2001.

[8] Alan Daly, William Marnane, Tim Kerins, and Emanuel Popovici, *Fast Modular Division for Application in ECC on Reconfigurable Logic*, Proc. of the 13th International Conference on Field-Programmable Logic and Applications (2003).

[9] A. De Vora, M. Ley, E. Ofner, and H. Grünbacher, *A High-Speed Radix 4 Hardware Divider For ASIC's*, Tagungsband Mikroelektronik (2003).

[10] David L. Harris, Stuart F. Oberman, and Mark A. Horowitz, *SRT Division Architectures and Implementations*, IEEE Symposium on Computer Arithmetic (1997).

[11] David W. Matula and Alex Fit-Florea, *Prescaled Integer Division*, IEEE Symposium on Computer Arithmetic (2003).

[12] James E. Stine and Michael J. Schulte, *A Combined Interval and Floating-Point Divider*, Proceedings of the 32nd Asilomar Conference on Signals, Systems, and Computers (1998).

[13] H. P. Sharangpani and M. L. Barton, *Statistical Analysis of Floating Point Flaw in the Pentium Processor*, Technical Report (1994).

[14] Robert Uang, Christian Bourdé, and Tim Bagwell, *A SiGe 23 GHz Fractional-N Frequency Synthesizer*, Technical Report (2004).

[15] Roman Lysecky and Frank Vahid, *A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning*, Proceedings of the Design, Automation and Test, Volume 1 (2005).