ALU Architecture with Dynamic Precision Support

Getao Liang, JunKyu Lee, Gregory D. Peterson
Department of Electrical Engineering and Computer Science
University of Tennessee
Knoxville, TN, USA
[gliang, jlee57, gdp]@utk.edu

***Abstract*—Exploiting computational precision can improve performance significantly without losing accuracy in many applications. To enable this, we propose an innovative arithmetic logic unit (ALU) architecture that supports true dynamic precision operations on the fly. The proposed architecture targets both fixed-point and floating-point ALUs, but in this paper we focus mainly on the precision-controlling mechanism and the corresponding implementations for fixed-point adders and multipliers. We implemented the architecture on Xilinx Virtex-5 XC5VLX110T FPGAs, and the results show that the area and latency overheads are *1% ~ 24%* depending on the structure and configuration. This implies the overhead can be minimized if the ALU structure and configuration are chosen carefully for specific applications. As a case study, we apply this architecture to binary cascade iterative refinement (BCIR). *4X* speedup is observed in this case study.**

Keywords-dynamic precision, ALUs, FPGAs, high-performance computing, iterative refinement

I. INTRODUCTION

In computational science and engineering, users continually seek to solve ever more challenging problems: faster computers with bigger memory capacity enable the analysis of larger systems, finer resolution, and/or the inclusion of additional physics. For the past several decades, standardized floating-point representations have enabled more predictable numeric behavior and portability, but at the expense of making it impractical for users to exploit customized precision with good performance. The introduction of reconfigurable computing platforms provides scientists with an affordable accelerating solution that not only has the computational power of dedicated hardware processors (ASICs and DSPs), but also the flexibility of software due to the fabric and circuit configurability [13]. Serving as either standalone platforms or co-processors, field-programmable gate arrays (FPGAs) have shown the potential of significant performance improvement over microprocessors for certain applications [14]. With the development of FPGA hardware and CAD tools, floating-point arithmetic functions are no longer impractical for FPGAs designs [15]. Instead, FPGA accelerators can out-perform general processors or GPUs in very-high-precision floating-point operations (higher than double-precision) due to the native hardware support. Other advantages FPGAs have over other computing solutions include their fine-grained parallelism, fault-tolerant designs, and flexible precision configurations.

Exploiting precision can gain performance improvement in many applications [1-5]. Lower precision renders Arithmetic Logic Units (ALUs) smaller implying higher performance and better energy efficiency [4]. The exploitable parallelism can be increased since the number of ALUs in fixed area is increased using smaller ALUs. Fewer transistors are employed to build smaller ALUs implying lower power consumption. Most current computing platforms employing statically defined arithmetic units such as multi-cores and GPUs face limitations on exploiting precision since they employ only single and double precision ALUs in hardware. However, FPGAs are able to support arbitrary precision ALUs as long as sufficient hardware resources are available.

Previous work shows the advantages of arbitrary precision ALUs on FPGAs in some applications [4-6], but the impact on performance of utilizing arbitrary precision can be maximized if ALUs on FPGAs can be switched to desirable precision ALUs *on the fly*. Exploiting multiple precisions in some applications is explored in [1, 7]. To employ multiple precision ALUs, it would be necessary to download bit-stream files on FPGAs whenever the precision requirement is changed, causing degraded performance. Partial reconfiguration (PR) can be applied to reduce the re-programing time with smaller partial bit-streams [33]. However, the performance improvement might be limited for designs with relatively small and frequently switching PR modules as ALUs. The small size of the ALU makes the cost of embedded/external processor and interface

CFP1225P-ART/12 $26.00 © 2012 IEEE
DOI 10.1109/SAAHPC.2012.29

for PR control too expensive; the frequent reconfiguration makes it harder to control and to guarantee the accuracy of operation; and the device-specific PR makes it difficult to port the design to other vendors or impossible to other technologies.

It is desired to remove the requirement for updating the configuration bit-stream. Hence, we propose a new ALU architecture performing arbitrary precision arithmetic operations according to the user's preference. This new paradigm for ALUs may enable users to prevent unnecessary loss of performance and energy from applying unnecessarily high precision on computation (e.g., programmers often employ double precision for scientific computations even when not needed). In small-scale applications, people often do not care as much about the performance loss from overly high precision. However, as scientific applications become larger, the potential impact of this new ALU architecture on achievable accuracy and performance becomes greater.

What distinguishes this dynamic precision ALU research from previous multiple precision work is the dynamic support of arbitrary precisions with arbitrary position arrangement. Although the proposed architecture is designed for floating-point ALUs, the main focus of this paper will be put on the fixed-point datapath, which is the fundamental arithmetic engine of a floating-point ALU.

Previous work on ALU structures and applications requiring multiple precisions are discussed in section II. Section III is dedicated to the approach and implementation for the proposed architecture. The implementations results are analyzed and a case study is conducted to justify the potentials of the proposed innovative architecture in section IV, followed by conclusions in section V.

II. Previous Work

A. ALUs Designs

Computer engineers have never stopped trying to improve system performance by optimizing arithmetic units. In 1951, Booth proposed a signed binary recoding scheme [16] for multipliers to reduce the number of partial products, and this scheme was later improved by Wallace in [17]. Besides integer operations, floating-point arithmetic is also a hot topic for many researchers [15, 18-22].

With the emergence of reconfigurable computing, engineers started to look for practical solutions for multiple-precision computations. Constantinides, and his colleagues had broad explorations [23-25] of bit-width assignment and optimization for static multiple-precision applications. Wang and Leeser spent years to develop and refine a complete statically-defined, variable-precision fixed- and floating-point ALU library for reconfigurable hardware [26].

Since re-synthesizing, re-downloading, and re-configuring are required for static multiple-precision ALUs whenever precision is changed, this solution is not practical for applications that require frequently changing precision. Thus, multi-mode ALUs become more attractive. In [27], Tan proposed a 64-bit multiply accumulator (MAC) that can compute one 64x64, two 32x32, four 16x16, or eight 8x8 unsigned/signed multiply-accumulations using shared segmentation. On the other hands, Akkas presented architectures for dual-mode adders and multipliers in floating-point [28, 29], and Isseven presented a dual-mode floating-point divider [30] that supports two parallel double-precision divisions or one quadruple-precision division. In [31], Huang present a three-mode (32-, 64- and 128-bit mode) floating-point fused multiply-add (FMA) unit with SIMD support. It is clear that all the above multi-mode multiple-precision structures can only support a few pre-defined precisions. To the best of our knowledge, our proposed architecture is the first true dynamic precision architecture targeting both fixed-point and floating-point ALUs.

B. Applications

Dynamic precision computations were investigated around 20 or 30 years ago, since at that time the computational scientists required extremely accurate numeric solutions compared to the contemporaneous computing technology [1, 7-9]. At that time, computations with high precision arithmetic generally used software routines.

In [1], Muller's method is implemented using dynamic precision computations. The implementation monitors the magnitude of some function values in order to recognize the solution is getting closer to the solution. For example, when the function value is small, they change the precision from low to high, recognizing the solution is getting closer to the exact solution. They claimed that the performance gain was from *1.4* to *4* compared to static precision computation. They utilized two types of precision on the MasPar MP-1 and Cray Y-MP.

Binary Cascade Iterative Refinement (BCIR) was proposed in 1981 [9]. BCIR seeks optimized performance to solve linear systems by utilizing multiple precisions. BCIR faces limitations for practical use since the condition number of the matrix should be given before computation to decide an initial precision (i.e., lowest precision among multiple precisions). Another paper proposes BCIR employing doubled precision arithmetic per iteration given an initial precision [8]. For numeric proofs and the BCIR algorithm, refer to [8, 9]. We perform a case study for the performance gain of BCIR if BCIR employs our ALU design in section IV.

III. Proposed Approach

Arithmetic logic units are the fundamental components within a CPU to process input data and produce output using certain standardized number systems. In addition to the arithmetic and logic operations, ALUs also need to detect exceptions and generate corresponding error flags for the status register. Given the huge design space, we simplify the ALU design presented in this paper without affecting generality by establishing some design specifications.

In order to be compatible with most current systems, a two's complement number system is selected as the interfacing data format between the ALU and outside logic. This system is also used as the internal representation within the functional unit only when signed values are required to be represented (e.g., Booth multiplier).

In this paper, only unsigned fixed-point operations are considered for designing the arithmetic units. The reasons for such a decision are as follows. First, fixed-point arithmetic is the core datapath that handles the computations on the mantissa bits and the exponent (mostly addition/subtraction) for a floating-point operation. Most of the circuit area for the dynamic precision ALU are dedicated to these portions of a floating-point unit. Second, it is trivial to transform an unsigned integer ALU into a signed one by simply adding extra control logic, without significant modification to the circuit. In most current floating-point representations, the mantissa can be only used to represent positive numbers or zero (e.g., enabling an integer unit compare to work for floating point values), so all the operand inputs for the fixed-point datapath can be represented by unsigned integers. Since this work is a gateway research for the dynamic precision floating-point ALUs, limiting the focus on unsigned operations will be sufficient to cover the design requirements.

No exception handling other than carry-in/carry-out will be taken into account in this paper.

A. Dynamic precision support

The block diagram for both the traditional design and the proposed ALU architecture are presented in Fig. 1, which obviously shows the similarity shared between these two designs. The only difference shown in the diagrams is that the bit-width for the control (*ctrl*) and carry-out (*carry*) signals for the design with dynamic precision (DP) support are no longer single-bit when the standard design is extended to support more than one precision-mode. The extra bits are used to deliver precision and overflow information respectively.

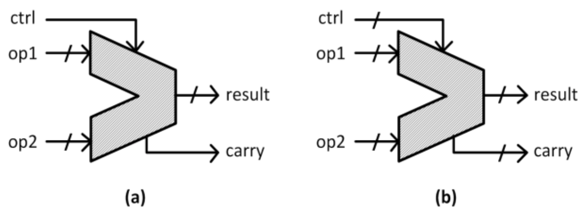

Figure 1. ALU Designs. (a) Traditional ALU. (b) ALU with multiple precision support

In contrast to multiple precision-mode designs, where only operations of certain precisions that match the supported modes can be performed in parallel (e.g., two single precision or one double precision operation), ALU with true dynamic precision support allows operations of any combination of different precisions simultaneously, as long as the total bit-width for operands is not exceeded.

To support true dynamic precision, an *n*-bit operand is partitioned into *k* sub-blocks with block size of *n/k*-bits. Depending on the design requirement on precision granularity, the number of blocks, *k,* can be any value between 1 and *n*. This size arrangement can be even more aggressive, allowing optimized a non-uniform, asymmetric distribution among different blocks for certain ALU structures. Adjacent sub-blocks can be combined together dynamically to form a super-block according to the currently required precision, so that independent operations can be performed on each super-block pair (one form each operands) without affecting others. This grouping process is controlled by a *(k-1)*-bit control signal (*ctrl*). The concept of the proposed operand segmentation and precision-controlling mechanism is illustrated in Fig. 2, where an *n*-bit operand is divided into two *n/4*-bit numbers and one *n/2*-bit number. More precision modes can be obtained by changing the *ctrl* signal. It is obvious that

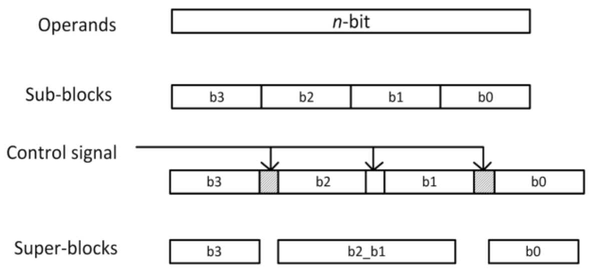

Figure 2. Operand segmentation and precision arrangement

the proposed mechanism guarantees that support for arbitrary precision-combinations is only limited by the sub-block segmentation.

Note that for addition (and subtraction), one extra bit *ctrl* signal is required to supply the carry-in shared by all the super-blocks, and a different one-bit *carry* will be generated for every super-block.

B. *Adder Design with DP support*

The critical path within an adder design is the carry chain that links separated full-adders together and produces the carries from the lowest position to the output. It is also the key factor to look into, when modifying a traditional adder to support multiple precision modes. Though implementation varies, the generation processes of carry signals can be classified into two categories: (1) carry propagated from previous bits and (2) carry generated using operands [32]. Accordingly, two different methods can be applied to handle dynamic precision support based on the actual implementations.

1) *Carry extraction and insertion*

Pure carry propagation is mainly used for slow adder realizations that require less area. Serial adders and ripple-carry adders are two common adder structures that fall into this category. In such adders, the carry-out for the current bit position cannot be evaluated until the carry from the previous position is available. Thus, in order to perform independent calculations on the re-grouped super-blocks, the carry propagations in between two super-block neighbors need to be specially handled.

First, it is necessary to terminate the carry propagation, so that the computation of one super-block will not be affected by the result from previous one. Then, carry extraction and insertion, as described in Eq. (1), can be performed to compute the carry-out (overflow) and carry-in values for the *i*-th and *(i-1)*-th sub-blocks. As mentioned above, a carry-in, denoted as c_0, for all the super-blocks should be identical for the support of SIMD-like operations.

$$\begin{aligned} carry\_out_{i-1} &= ctrl_{i-1} \cdot c_{i-1} \\ carry\_in_i &= ctrl_{i-1} \cdot c_0 + \overline{ctrl_{i-1}} \cdot c_{i-1} \end{aligned} \tag{1}$$

The hardware implementation for these special functions is shown in Fig. 3. One can see that in both Eq. (1) and Fig. 3 the corresponding *ctrl* signal bit is used to generate the correct carry-in and carry-out signal for that respective sub-block. In fact, this circuit also suggests the construction of super-blocks

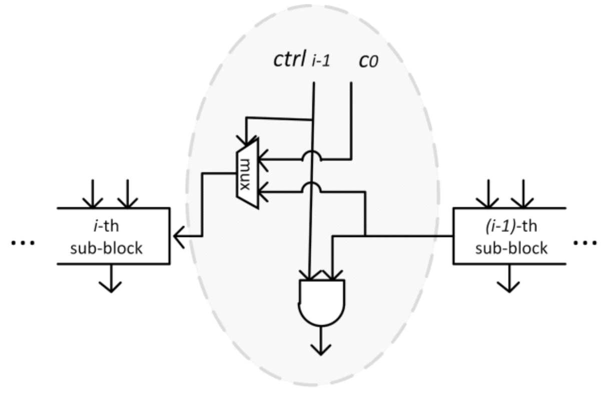

Figure 3. Hardware for carry extraction and insertion

implicitly. Thus no extra hardware resources have to be allocated for a dedicated segmentation circuit. Note also that for every sub-block, there are 4 extra gates, resulting in two extra gate-levels of delay. The latency from every extra block accumulates due to the fact that these circuits are cascaded on the critical path. Therefore, the total area and delay overhead introduced by adding dynamic precision support to a ripple-carry adder can be calculated with Eq. (2). Assuming a full adder consists of 4 gates with a latency of 3 gate-levels, the area overhead is *k/n* of the area of a static precision design, and the latency overhead is *2k/3n*.

$$\begin{aligned} area_{overhead} &= 4k \\ delay_{overhead} &= 2k \end{aligned} \tag{2}$$

2) *Carry manipulation*

In contrast to the adders with carry propagation whose worst-case delays increase linearly with the bit-width of operands, most modern CPU adders employ a scheme called carry-lookahead for carry generation from the operands directly or indirectly [32]. This scheme can achieve logarithmic time delay at a cost of extra hardware area. Instead of working on the carry signals directly as in ripple-carry adders, carry manipulation is applied for fast adders. This scheme manipulates the carry process at the edge between two continuous super-blocks by engineering the MSB's generate (g_i) and propagate (p_i) signals from every sub-block. The logic of this method is shown in Eq. (3).

$$\begin{aligned} p_i' &= \overline{ctrl_i} \cdot p_i \\ g_i' &= ctrl_i \cdot c_0 + \overline{ctrl_i} \cdot g_i \end{aligned} \tag{3}$$

By comparing Eq. (3) with Eq. (1), one can see that they are arithmetically similar to each other. Thus, the hardware logic for carry manipulation can also be implemented with one two-input MUX and one AND gate. However, with this method, the calculation of all generate and propagate signals can be performed in parallel without having to wait for the completion of carry generation from previous positions. Since the carry generation process is modified, the calculation of the final carry-out (overflow) for each super-block requires one extra gate-level delay. Therefore, the total latency overhead for adding dynamic precision support is only *(2+1)* gate-levels. For better comparison with the ripple-carry adder, the absolute total area and delay overhead introduced by dynamic precision is listed in Eq. (4). As there are various carry-lookahead adder designs [32], it is difficult to represent the extra area and delay in terms of percentages to those in static precision designs. But it is safe to say that the percentage will be small, given that most carry-lookahead adders are larger than ripple-carry adders.

$$\begin{aligned} area_{overhead} &= 4k \\ delay_{overhead} &= 3 \end{aligned} \tag{4}$$

C. Multiplier Design with DP support

The block diagram, as illustrated in Fig. 4, for a tree multiplier with dynamic precision support consists of three major design blocks: a partial products generator, a partial products reduction tree, and a final carry-propagate adder. Extra circuitry is designed in the first two blocks in order for the multiplier to support dynamic precision modes.

1) Partial products generator

To support dynamic precision operations, extra care is taken when generating the partial products. Instead of copying the whole multiplicand as the partial product for every '1' in the multiplier, the generator only selects the multiplicand bits from the corresponding super-block based on the multiplier bit position, and sets all others as '0'. This can be realized using AND gates with multiplicand bits and a block-select signal (*block_sel*) as inputs. For every sub-block pair, a *k*-bit *block_sel* is generated by a series of OR trees using *ctrl* signals. Fig. 5 shows the hardware for the selection process of sub-blocks and generation of the un-shifted partial product. As mentioned above, the multiplier design presented in this paper supports only unsigned operands, so no hardware is needed for negative weighted bits as in signed numbers. The additional area and delay introduced by the DP circuits is listed in Eq. (5).

$$\begin{aligned} area_{overhead} &= k(k-1) + \sum_{i=2}^{k-1}\left(\sum_{j=1}^{i-1}(i-j) + \sum_{j=i+1}^{k}(j-i)\right) \\ &= \frac{1}{3}k^3 - \frac{1}{3}k \\ delay_{overhead} &= \log_2(k) \end{aligned} \tag{5}$$

If a high-radix Booth recoding is used for the generation of partial products, additional circuit blocks are necessary to handle the following problems:

- Generation of sign extension
- Zero padding for LSB/MSB of each super-block
- Overlapped recoding cases for LSB of each super-block
- Extra recoding cases for MSB of each super-block

These hardware blocks include a sign encoder, a block-boundary detector, a boundary-bit detector, and other supporting modules.

2) Reduction tree

If a bit-wise partial products generator is employed, the carry will never be generated or propagated outside of the super-block (2X the size of the operands) due to the nature of unsigned multiplication. Therefore, no additional circuit is required in the reduction tree.

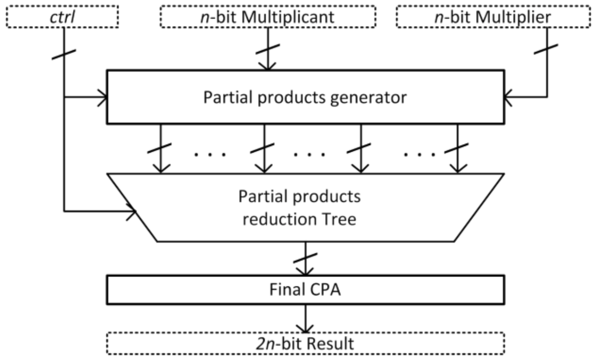

Figure 4. Multiplier design with dynamic precision support

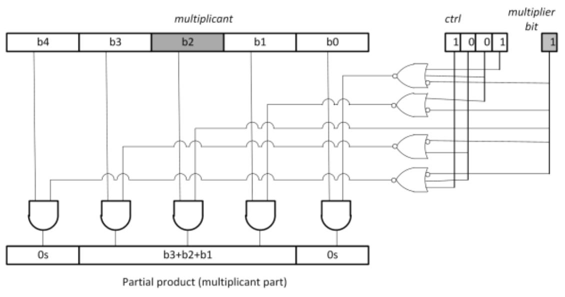

Figure 5. Hardware for partial products generation

However, when Booth recoding is used, some of the partial products might become negative numbers represented as two's complement number. Summation of two's complement numbers can cause overflow, resulting in unwanted carries propagating through super-block boundaries. The *ctrl* signal is used to filter any unwanted carries at the sub-block boundaries. The fundamental arithmetic for such filtering is the same as in Eq. (1) for the hardware implementation. At the sub-block boundaries, one extra AND gate is added to the critical path for every level of the reduction tree. In fact, this can be further optimized by reducing the number of level that might cause unwanted carries, or integrating the termination circuit with the reduction elements.

IV. IMPREMENTATION RESULTS AND A CASE STUDY

The proposed precision-controlling mechanism requires additional hardware resources and introduces extra delay in the critical path of ALU designs. To examine the latency and area impact on different implementations of adders or multipliers, two versions of implementations are developed for several popular adder/multiplier structures. For fair comparison, a standard version is created manually first, and then modifications are performed to enable dynamic precision support, using the same level of development and optimization efforts.

All designs are implemented in VHDL with parameterized bit-width for operand and sub-block, and are extensively simulated with randomly generated operands to verify their functionality. Xilinx ISE 13.4 is used to synthesize the designs for Xilinx Virtex-5 XC5VLX110T FPGAs.

Table I shows the area and latency information for four different adder implementations: (1) adder generated from Xilinx proprietary IP (fabric only); (2) ripple-carry adder with small footprint and linear delay; (3) multi-level carry-lookahead adder; (4) hybrid prefix adder with a moderate latency-area tradeoff. Since the Xilinx adder is deeply optimized for Xilinx's own FPGAs, it gives the best results for both area and latency. The other three manually designed adders show some degree of degradation in comparison, which is expected considering the characteristics of these implementations and the lesser amount of effort in design optimization.

Normalized area results for the three adder implementations with dynamic precision support are provided in Table II. To emphasize the overhead posed by the modification, the area data listed are normalized with those from respective adders without DP support. The hardware size growth with the increasing of either

TABLE I. AREA AND LATENCY FOR STATIC DESIGNS

Data Width						Area: LUT
	4	8	16	32	64	128
Xilinx IP	4	8	16	32	64	128
RCA	8	16	32	64	128	256
CLA	20	48	107	227	470	958
Prefix	24	56	128	288	640	1408

Data Width						Latency: ns
	4	8	16	32	64	128
Xilinx IP	1.92	2.03	2.23	2.65	3.48	5.15
RCA	3.02	5.33	9.97	19.23	37.76	74.81
CLA	3.22	5.04	5.63	7.52	8.20	10.10
Prefix	3.69	4.94	5.72	6.40	7.08	7.76

TABLE II. NORMALIZED AREA FOR DYNAMIC PRECISION ADDERS

	Data Width				Normalized Area
RCA	8	16	32	64	128
Block Size 4	1.13	1.19	1.22	1.23	1.24
8		1.06	1.09	1.11	1.12
16			1.03	1.05	1.05
32				1.02	1.02
64					1.01
CLA	8	16	32	64	128
Block Size 4	1.08	1.11	1.12	1.13	1.13
8		1.04	1.05	1.06	1.06
16			1.02	1.03	1.03
32				1.01	1.01
64					1.00
Prefix	8	16	32	64	128
Block Size 4	1.07	1.09	1.10	1.09	1.09
8		1.03	1.04	1.04	1.04
16			1.01	1.02	1.02
32				1.01	1.01
64					1.00

TABLE III. NORMALIZED LATENCY FOR DYNAMIC PRECISION ADDERS

	Data Width				Normalized Latency
RCA	8	16	32	64	128
Block Size 4	1.11	1.17	1.21	1.23	1.24
8		1.06	1.09	1.11	1.12
16			1.03	1.05	1.05
32				1.02	1.02
64					1.01
CLA	8	16	32	64	128
Block Size 4	1.08	1.17	1.14	1.13	1.10
8		1.07	1.11	1.10	1.08
16			1.00	1.06	1.06
32				1.00	1.06
64					1.00
Prefix	8	16	32	64	128
Block Size 4	1.07	1.15	1.15	1.14	1.13
8		1.03	1.12	1.12	1.11
16			1.03	1.10	1.10
32				1.01	1.08
64					1.00

data width or number of blocks is consistent with the analysis in section III. It is also clear that prefix adders suffer less, in terms of area, from the additional precision-control circuitry than the others architectures.

Likewise, normalized latency results are provided in Table III. In general, the latency impacts from the additional modules share the same pattern shown in Table II. However, carry-lookahead adders have the best performance in terms of absorbing latency overhead.

To evaluate the impact of dynamic precision on real applications, we emulate the DP ALUs and analyze the impact on performance when utilizing multiple precisions dynamically. We choose Binary Cascade Iterative Refinement (BCIR) in [8, 9] for the application. A set of arbitrary precision arithmetic operators is developed in C, and then used to implement the BCIR algorithm employing Gaussian Elimination with Partial Pivoting (GEPP). We compare the total run-times obtained from BCIR and the double precision direct method. The direct method solves the problem with matrix decomposition and back-substitution [10]. The iteration in BCIR is terminated if the solution accuracy is higher than that from the double precision direct method. 100 64 x 64 Gaussian Random matrices are tested, and the run-time for each matrix is calculated based on the run-time impact on precision described in [4]. Finally, achievable speedups are calculated as T_{DIR}/T_{BCIR}, where T_{DIR} is the run-time for the direct method and T_{BCIR} is the run-time for the BCIR.

Fig. 6 shows the required number of iterations for BCIR according to the condition numbers. The X-axis represents the log_2 based condition numbers and the Y-axis represents the required number of iterations for BCIR. Notice that employing our ALUs does not require downloading bit-stream files while employing static arbitrary precision ALUs with FPGAs requires downloading bit-stream files as often as the required number of iterations due to the changing precision used for each iteration. Fig. 7 shows the estimated speedups when utilizing precisions to solve linear systems with

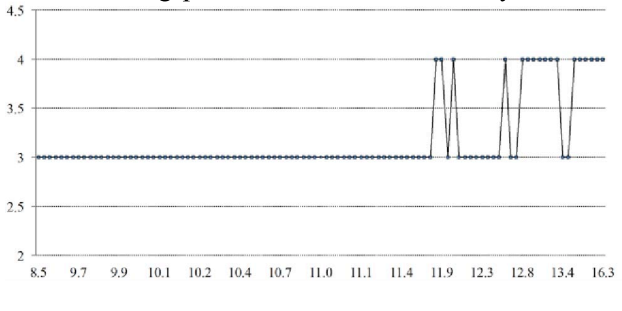

Figure 6. Required Number of BCIR Iterations as a Function of the Logarithm of the Condition Number

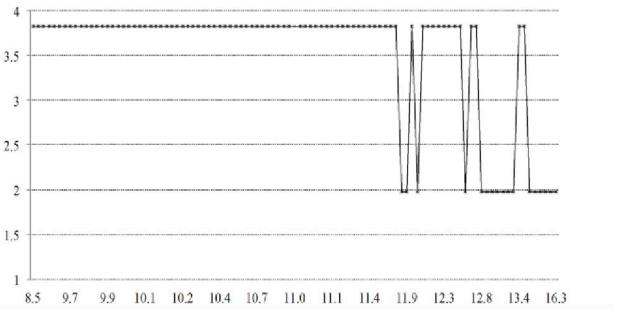

Figure 7. Performance Gain as a Function of the Logarithm of the Condition Number

BCIR employing our proposed dynamic precision ALUs (i.e., this run-time does not consider the overhead due to multiple iterations). In Fig. 7, the X-axis represents the log_2 based condition numbers and Y-axis represents the achievable speedups using BCIR employing our ALUs.

In this case study, nearly *4X* speedup can be achieved by utilizing dynamic precisions in BCIR employing our DP ALUs when the systems are not ill conditioned, as compared to the given initial precision. The speedup can be greater when matrix size becomes larger or accuracy requirements become higher [11, 12]. Hence, we expect our ALUs to have even more benefit with large-scale high-accuracy applications.

V. Conclusion

Reconfigurable computing with FPGAs has shown impressive performance improvements on a variety of applications for computational sciences Exploiting precision at both the application-level and ALU-level can gain even more from these powerful yet flexible hardware accelerators. In this paper, we proposed an innovative ALU architecture that supports dynamic precision operations on the fly. Our architecture can improve the computational throughput for low-precision operations by increasing the parallelism, without losing the ability to perform high-performance operations for high-precision applications. We implemented the proposed architecture for some fixed-point adders and multipliers with parameterized VHDL, and their functionalities are tested and verified. Implementation results from 3 adder structures of different precision configurations were presented to analyze the area and delay overheads, which range from *1%* to *24%*. A case study was also conducted to evaluate the impact of the approach, with speedups approaching 4X for a linear system solver application of small size. When system size becomes larger or accuracy requirement becomes higher, the speedup can be greater.

ACKNOWLEDGMENT

This work was partially supported by the National Science Foundation, grant NSF CHE-0625598.

REFERENCES

[1] D. A. Kramer and I. D. Scherson, "Dynamic Precision Iterative Algorithms," in The IEEE 1992 Symposium on the Frontiers of Massively Parallel Computation, 1992.

[2] J. Kurzak and J. Dongarra, "Implementation of mixed precision in solving systems of linear equations on the Cell processor: Research Articles," Concurr. Comput. : Pract. Exper., vol. 19, pp. 1371-1385, 2007.

[3] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra, "Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems)," presented at the Proceedings of the 2006 ACM/IEEE conference on Supercomputing, Tampa, Florida, 2006.

[4] J. Lee and G. D. Peterson, "Iterative Refinement on FPGAs," in Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on, 2011, pp. 8-13.

[5] A. R. Lopes, A. Shahzad, G. A. Constantinides, and E. C. Kerrigan, "More flops or more precision? Accuracy parameterizable linear equation solvers for model predictive control," in IEEE Symposium on Field Programmable Custom Computing Machines, Napa, California, 2009.

[6] J. Sun, G. D. Peterson, and O. O. Storaasli, "High-Performance Mixed-Precision Linear Solver for FPGAs," IEEE Trans. Comput., vol. 57, pp. 1614-1623, 2008.

[7] M. Zubair, S. N. Gupta, and C. E. Grosch, "A variable precision approach to speedup iterative schemes on fine grained parallel machines," Parallel Computing, vol. 18, pp. 1223-1231, 1992.

[8] A. Smoktunowicz and J. Sokolnicka, "Binary cascade iterative refinement in doubled-mantissa arithmetics," BIT, vol. 24, pp. 123-127, 1984.

[9] A. Kielbasinski, "Iterative refinement for linear systems in variable-precision arithmetic," BIT, vol. 21, pp. 97-103, 1981.

[10] L. N. Trefethen, Numerical Linear Algebra: SIAM, 1998.

[11] D. H. Bailey, "High-Precision Floating-Point Arithmetic in Scientific Computation," Computing in Science and Engg., vol. 7, pp. 54-61, 2005.

[12] A. Edelman, "Large Dense Numerical Linear Algebra in 1993: The Parallel Computing Influence," The International Journal of Supercomputer Applications, vol. 7, pp. 113-128, 1993.

[13] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," ACM Computing Surveys, vol. 34, no. 2, pp. 171-210, Jun. 2002.

[14] S. Craven and P. Athanas, "Examining the Viability of FPGA Supercomputing," EURASIP Journal on Embedded Systems, vol. 2007, pp. 1-8, 2007.

[15] K. S. Hemmert and K. D. Underwood, "Fast , Efficient Floating-Point Adders and Multipliers for FPGAs," Technology, vol. 3, no. 3, 2010.

[16] A. Booth, "A signed binary multiplication technique," The Quarterly Journal of Mechanics and Applied, vol. 4, no. 2, pp. 236-240, 1951.

[17] C. S. Wallace, "A suggestion for a fast multiplier," Electronic Computers, IEEE Transactions on, vol. EC–13, no. 1, pp. 14–17, 1964.

[18] S. Anderson and J. Earle, "The IBM system/360 model 91: floating-point execution unit," IBM Journal of, no. January, 1967.

[19] P. Seidel and G. Even, "Delay-optimized implementation of IEEE floating-point addition," IEEE Transactions on Computers, vol. 53, no. 2, pp. 97-113, Feb. 2004.

[20] R. M. Jessani and M. Putrino, "Comparison of single- and dual-pass multiply-add fused floating-point units," IEEE Transactions on Computers, vol. 47, no. 9, pp. 927-937, 1998.

[21] M. J. Schulte, D. Tan, and C. E. Lemonds, "Floating-point division algorithms for an x86 microprocessor with a rectangular multiplier," in 2007 25th International Conference on Computer Design, 2007, pp. 304-310.

[22] G. Even and P.-M. Seidel, "A comparison of three rounding algorithms for IEEE floating-point multiplication," IEEE Transactions on Computers, vol. 49, no. 7, pp. 638-650, Jul. 2000.

[23] G. A. Constantinides, P. Y. K. Cheung, and W. Luk, "Wordlength optimization for linear digital signal processing," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 22, no. 10, pp. 1432-1442, Oct. 2003.

[24] G. a. Constantinides, P. Y. K. Cheung, and W. Luk, "Optimum and heuristic synthesis of multiple word-length architectures," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 13, no. 1, pp. 39-57, Jan. 2005.

[25] D.-U. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides, "Accuracy-Guaranteed Bit-Width Optimization," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 25, no. 10, pp. 1990-2000, Oct. 2006.

[26] X. Wang, "VFloat: A Variable Precision Fixed-and Floating-Point Library for Reconfigurable Hardware," ACM Transactions on Reconfigurable Technology, vol. 3, no. 3, pp. 1-34, 2010.

[27] D. Tan, A. Danysh, and M. Liebelt, "Multiple-precision fixed-point vector multiply-accumulator using shared segmentation," in 16th IEEE Symposium on Computer Arithmetic, 2003. Proceedings., 2003, vol. 00, no. C, pp. 12-19.

[28] A. Akkas, "Dual-mode floating-point adder architectures," Journal of Systems Architecture, vol. 54, no. 12, pp. 1129-1142, Dec. 2008.

[29] A. Akkas and M. Schulte, "Dual-mode floating-point multiplier architectures with parallel operations," Journal of Systems Architecture, vol. 52, no. 10, pp. 549-562, Oct. 2006.

[30] A. Isseven and A. Akkas, "A Dual-Mode Quadruple Precision Floating-Point Divider," in Signals, Systems and Computers, 2006. ACSSC'06. Fortieth Asilomar Conference on, 2006, pp. 1697–1701.

[31] L. Huang, S. Ma, L. Shen, Z. Wang, and N. Xiao, "Low Cost Binary128 Floating-Point FMA Unit Design with SIMD Support," IEEE Transactions on Computers, vol. PP, no. 99, pp. 1-8, 2011.

[32] B. Parhami, Computer Arithmetic: Algorithms and Hardware Designs, 2nd edition, Oxford University Press, New York, 2010.

[33] C. Kao, "Benefits of partial reconfiguration," Xcell journal, pp. 65-67, 2005.