VLIW Coprocessor for IEEE-754 Quadruple-Precision Elementary Functions

YUANWU LEI, YONG DOU, LEI GUO, JINBO XU, and JIE ZHOU, National University of Defense Technology

YAZHUO DONG, People's Liberation Army, Beijing

HONGJIAN LI, Logistics Scientific Institute, Beijing

In this article, a unified VLIW coprocessor, based on a common group of atomic operation units, for Quad arithmetic and elementary functions (QP_VELP) is presented. The explicitly parallel scheme of VLIW instruction and Estrin's evaluation scheme for polynomials are used to improve the performance. A two-level VLIW instruction RAM scheme is introduced to achieve high scalability and customizability, even for more complex key program kernels. Finally, the Quad arithmetic accelerator (QAA) with the QP_VELP array is implemented on ASIC. Compared with hyper-thread software implementation on an Intel Xeon E5620, QAA with 8 QP_VELP units achieves improvement by a factor of 18X.

Categories and Subject Descriptors: B.2.4 [Arithmetic and logic structures]: High-Speed Arithmetic

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Quadruple precision floating-point, elementary function, QP_VELP, very long instruction word

ACM Reference Format:

Lei, Y., Dou, Y., Guo, L., Xu, J., Zhou, J., Dong, Y., and Li, H. 2013. VLIW coprocessor for IEEE-754 quadruple-precision elementary functions. *ACM Trans. Architect. Code Optim.* 10, 3, Article 12 (September 2013), 22 pages.

DOI: <http://dx.doi.org/10.1145/2512430>

1. INTRODUCTION

Most scientific applications operate on double-precision (64-bit) floating-point arithmetic. However, its accuracy is not reliable for future large-scale computation applications. In ExaScale computing, accuracy errors increase up to a level where only 11 significant bits in the mantissa are left [Dou et al. 2010]. Therefore, with the IEEE-754-2008 revision of the standard for the binary floating-point arithmetic [IEEE 2008], the quadruple-precision floating-point (Quad: 128-bit) format has been introduced, satisfying the demand for numerical precision of scientific applications, such as climate modeling, supernova simulations, and computational geometry [Bailey 2005].

An earlier version of this article appeared in the IEEE 29th International Conference on Computer Design (ICCD) [Lei et al. 2011].

This work is supported by NSFC (61125201, 61202126, and 60903057).

Author's addresses: Y. Lei (corresponding author), L. Guo, J. Xu, J. Zhou, National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Changsha 410073, China; email: yuanwulei@nudt.edu.cn; Y. Dong, Unit 91655, People's Liberation Army, Beijing, China; H. Li, Logistics Scientific Institute, General Logistics Department, Beijing, China.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2013 ACM 1544-3566/2013/09-ART12 \$15.00

DOI: <http://dx.doi.org/10.1145/2512430>

Variations of the Quad arithmetic are typically implemented through software approaches and then integrated into compilers for different companies, such as Intel Fortran Compiler [Intel 2012], gcc on PowerPC, and Sun Studio Compiler on SPARC. The main disadvantage of software approaches is slower performance, which is at least one order of magnitude slower than double-precision arithmetic. Thus, hardware designs for Quad arithmetic are required to overcome the speed limitation of software solutions. However, little hardware instruction at present directly supports Quad arithmetic in general-purpose processors. To the best of our knowledge, the IBM S/390 G5 processor [Schwarz et al. 1999] is the only commercial processor that supports Quad basic arithmetic in hardware with 64-bit internal datapaths. Computation units for Quad basic operations have been presented. These units include the multimode Quad FMA unit proposed by Mustafa [Gok and Ozbilen 2008] and Huang Huang et al. [2012], and the dual-mode Quad adder [Akkas 2008], multiplier [Akkas and Schulte 2006], and divider proposed by Akkas et al. However, few works have focused on the hardware design for Quad elementary functions or on the design for Quad coprocessor to accelerate scientific applications with Quad arithmetic.

Elementary functions and the combination of these functions are important ingredients of scientific applications. They have served as performance bottlenecks for some compute-bound applications. For example, Stuart F. Oberman found that floating-point division accounts for 40% of the latency in only 3% of the codes from the SPECfp92 benchmark suite [Oberman and Flynn 1995]. In the Simplified General Perturbation v4 / Simplified Deep-space Perturbation v4 (SGP4/SDP4) application [Vallado et al. 2006], approximately 70% of the latency is used for the evaluation of elementary functions.

Whether or not elementary functions should be implemented in hardware has been discussed in the beginning of the PC era [Paul and Wilson 1976]. It goes through from hardware design to software emulation, then back to hardware implementation. In the 1970s, researchers focused on the hardware implementation of elementary functions [Wrathall and Chen 1978; Ercegovac 1973]. However, the chip area was small and the hardware cost for elementary functions occupied a large part of the chip area at that time. Thus, the general-purpose processors were not equipped with the hardware for these functions. The processor used microcode or software library to evaluate elementary functions, which led to considerably lower performance. With the advent of reconfigurable devices, such as Field-Programmable Gate Arrays (FPGAs), the devices have facilitated the implementation of elementary functions as coprocessors to achieve higher performance [Detrey et al. 2007; Lei et al. 2011]. With the further increase of chip area, it is possible to implement the more complex elementary function units on one chip. As such, it is necessary to consider reintegrating the elementary function units into the processor.

Recently, heterogeneous multicore processors have become the one of main current architectures for future processors [Brodtkorb et al. 2010]. Different from traditional CPUs, heterogeneous architectures offer an alternative to provide high peak performance, with traditional multicore architectures in combination with accelerator cores such as GPU and reconfigurable devices. For a given application, accelerator cores are designed to maximize performance with fewer transistors and lower frequencies, leading to higher energy and cost efficiency compared with the traditional CPUs. Therefore, it is feasible to integrate a Quad arithmetic coprocessor, which evaluates the Quad basic operations and Quad elementary functions, into the processor to improve the performance and efficiency for high-precision applications.

However, some challenges remain in the hardware implementation of Quad arithmetic accelerators, particularly a Quad elementary function. First, the hardware cost for a Quad elementary function is extremely large. The realization algorithm for

elementary functions is more complex than the basic floating-point operation. The computation complexity of common elementary functions with the Taylor method is given by $O(n^{0.5} \cdot M(n))$, where $M(n)$ refers to the complexity of the multiplication operation and is $O(n^2)$ in the general schoolbook scheme, and n refers to the precision of the result. Therefore, the consumption of hardware resources quickly increases with the computation width.

Second, it is unreasonable and impractical to use large number of silicon to implement all Quad elementary functions in a processor individually. This is because a variety of Quad basic operations and elementary functions appear in the same application. For instance, the SGP4/SDP4 application includes five types of elementary functions, that is, division, square root, reciprocal cube root, sine, and cosine. Likewise, the Chirp Scaling algorithm (CS) [Raney et al. 1994], a SAR radar image processing algorithm, includes five types of elementary functions, namely division, square root, logarithm, sine, and cosine. The utilization of hardware becomes lower due to the low-usage frequency of these functions. Moreover, it is difficult to extend in order to evaluate other elementary functions with the one-by-one way.

Third, the latency of Quad elementary functions is long because higher internal working precision and more iteration times are required to obtain an accurate result. Therefore, a hardware algorithm with high convergence is chosen to improve the performance, thereby increasing the hardware cost.

Finally, in some application domains, various complex elementary functions, called “coarser-grain functions” in the current article, are constructed with several common elementary functions and basic operations. These functions, such as $\log_2(1 + 2^x)$ in a logarithm number system, are the basic functions and limit the performance of the applications. Thus, hardware implementation can help improve the performance of applications in such domains. However, it is difficult to satisfy the variety and versatility of coarser-grain functions with the traditional implementation method.

Based on the analysis of hardware algorithm for Quad elementary functions, we summarize a common group of atomic operations that implement various complex elementary functions. Subsequently, we present a unified Very Long Instruction Word (VLIW) coprocessor for IEEE-754 quadruple-precision elementary functions (QP_VELP), which is built with multiple atomic operation units. The coprocessor integrates the VLIW architecture’s properties, such as low hardware cost, high performance, and high scalability and customizability, thus satisfying the variety and versatility of complex elementary functions for different applications. In the ASIC implementation, several schemes, such as 128-bit truncated multiplier, two-level VLIW instruction RAM scheme, and Estrin’s scheme for the evaluation of a polynomial, reduce hardware cost and improve scalability and performance. Finally, the Quad arithmetic accelerator chip, equipped with the QP_VELP array, is developed to accelerate Quad elementary functions and Quad applications. The experimental results show that the accelerator chip, running at 1.39 GHz, outperforms the serial and parallel software implementation on an Intel Xeon E5620 CPU with 2.4 GHz by a maximum factor of 95X and 18X, respectively. Moreover, this chip demonstrates significantly lower power consumption.

This article is based on our previous works in Lei et al. [2011] and the following improvements are presented.

- The instruction set architecture of QP_VELP is presented, which is composed of a group of atomic operations for Quad arithmetic.
- The two-level VLIW instruction RAM scheme is proposed to reduce the memory requirement for VLIW instructions and achieve high scalability and customizability.
- Estrin’s scheme is proposed to reduce the execution cycle for the polynomial evaluation and improve the performance of QP_VELP.

Method	Precision	Hardware Cost	Convergence	Algorithms	Elementary Function
Lookup Table	Low	Very High	Very High	-	All
Digit-Recurrence	Medium	Low	Low	SRT, CORDIC, Non-Restoring	Common functions
Polynomial Approximation	Medium/High	High	Medium	Taylor	Almost all
Function-Recurrence	Medium/High	Medium	High	Newton, Goldschmidt	Square/Cube root, Division

Fig. 1. Comparison of hardware algorithms.

- Quad arithmetic accelerator, equipped with a QP_VELP array, is implemented in an FPGA prototype and ASIC platform.
- Quad basic operations, elementary functions, coarse-grain functions, key program kernels, and CS SAR application are implemented to explore the performance and scalability of QP_VELP.

The rest of the article is organized as follows. Section 3 presents the summary of the common atomic operations for Quad elementary. Sections 4 and 5 present the implementation details of the QP_VELP and the optimization scheme on QP_VELP, respectively. Section 6 illustrates the accelerator chip of the QP_VELP array. Section 7 presents the comparison of the performance with software implementation. Section 8 concludes and proposes ideas for future work.

2. BACKGROUND

2.1. IEEE-754 Quadruple-Precision Format

The IEEE-754 quadruple precision floating-point format (Quad), also called binary128, is a 128-bit base-2 format with a 1-bit sign (Sx), a 15-bit bias exponent (BEx), and a 113-bit significand (Fx). The significand is written with an implicit lead bit for normalized value, that is, $1 \leq Fx < 2$. In this article, we support the normal number and rounding off the result to the nearest even number. The value of normal Quad number x is given by $x = (-1)^{Sx} \times Fx \times 2^{BEx-16383}$.

2.2. Definition of Operation Types

The operation types are defined as the following.

- Atomic operation.* The basic operation units, such as 128-bit fixed computation operations and condition operations, constitute the Instruction Set Architecture (ISA) of QP_VELP.
- Quad basic operation.* The basic quadruple-precision arithmetic operations are defined, such as Quad addition, subtraction, multiplication, and division.
- Quad elementary function.* The complex quadruple-precision arithmetic operations, such as Quad square root, exponential, logarithm, and trigonometric functions are defined.
- Quad coarser-grain function.* The complex elementary functions, such as $\ln(-\ln(x))$ and $\sqrt{x^2 + y^2 + z^2}$, are constructed with several common Quad elementary functions and Quad basic operations.

2.3. Hardware Algorithms for Elementary Function

The main methods used for evaluating elementary functions in hardware are classified into two: noniterative and iterative. The first method includes direct lookup table and polynomial approximations, and the second includes digit and function recurrence. Figure 1 summarizes the features of each method.

The direct lookup table method is suitable for low-precision calculations and can obtain the result in one or two cycles. However, the memory requirement increases exponentially with precision. Even for single-precision arithmetic, tables with the size of $2^{32} \times 32 = 128G$ bits are required. The lookup table method is generally combined with other methods to balance the speed and hardware requirements of high-precision arithmetic [Muller 2006].

Digit-recurrence methods such as SRT and CORDIC algorithm are implemented with simple addition and shift operations. However, it is linearly convergent and requires long latency to obtain a high-precision result.

Newton's algorithm and polynomial approximation are multiplicative approaches used as major tools for high-precision arithmetic. Newton's algorithm [Brent and Zimmermann 2010] computes functional inverse functions, such as reciprocal, division, and square root. Typically, this method has quadratic convergence, resulting in low latency for high-precision calculation. Polynomial approximation is a general technique used in evaluating numerical functions of one variable, such as exponential, logarithm, and trigonometric functions. The degree of the polynomial is proportional to the precision of the result. Numerous multiplication and addition operations must be performed for high-precision calculation. Therefore, a lookup table is always followed by multiplicative approaches in order to reduce computation complexity. The compromise between lookup table size and Newton's iterations number or polynomial degree should be taken into consideration.

3. ATOMIC OPERATIONS IN ELEMENTARY FUNCTIONS

In this section, we first introduce the computation steps for elementary functions and summarize the common group of atomic operations for such functions. Based on these atomic operation units, a unified VLIW architecture for computing various Quad elementary functions and coarser-grain functions is built. The unified architecture integrates the high-performance, scalability, and customizability properties of the VLIW technique.

3.1. Computation Steps for Elementary Functions

The range reduction scheme is used to improve computational efficiency. The evaluation of the function f at x is divided into three steps: range reduction, evaluation, and reconstruction.

(A) Range Reduction

The first step is to transform x from a Quad number to fixed-point format (i.e., Sx , BEx , and Fx). Subsequently, a two-level range reduction scheme is used to reduce the argument range. In the first level, the value of Fx is transformed into a small value $Fx'[Fx' \in [0, 2^i]]$ based on the properties of f , such as periodicity of the trigonometric function. In the second level, the interval $[0, 2^i)$ is split into 2^k subintervals, namely $Fx' = y + z = 2^{i-k} \times y' + z$, where y' is the leading k bits of the binary representation of Fx' , and $z \in [0, 2^{i-k})$. Thus, the evaluation of $f(x)$ is divided into two parts: $f(y)$ and $f(z)$. Then, the value of $f(y)$ is obtained from a lookup table indexed by y' . The computation operations, such as multiplication, addition, and shift, are included in the first level. The lookup table operation is the major operation in the second level.

(B) Evaluation

Generally, the value of $f(z)$ is evaluated using the CORDIC algorithm, Newton's method, or polynomial approximation. For the CORDIC algorithm, only the addition and shift are used. Owing to the linear convergence of this algorithm, the number of iterations (N_C) should be greater than 113 to obtain a Quad result. Newton's method is implemented using the iteration approach. The number of iterations (N_N) satisfies

the expression given by

$$N_N = \log_2 \left(\frac{Prec_r}{Prec_i} \right) \geq \log_2 \left(\frac{128}{k-i} \right).$$

where $Prec_i$ and $Prec_r$ denote the precision of the initial value and result, respectively. For polynomial approximation $f(z) = \sum_{i=0}^{N_P} a_i \times z^i$, the degree (N_P) satisfies

$$N_P \geq \frac{128 - \log_2(a_{N_P})}{(k-i)}.$$

The two equations indicate that the latency of Newton's method or polynomial approximation is inversely proportional to k . However, the size of the lookup table for $f(y)$ is approximately $128 * 2^k$. Thus, a suitable value of k should be selected to balance the computational complexity of $f(z)$ and the storage requirement of $f(y)$. Both Newton and polynomial approximation can be divided into a series of multiplications and additions.

(C) Reconstruction

The value of $f(x)$ is reconstructed from $f(y)$ and $f(z)$, using a simple identity transform, consisting of basic fixed-point operations. The normalized result is finally obtained by counting the leading zeros and rounding the rightmost bit of $f(x)$ to the nearest even number.

3.2. Common Atomic Operations

At first, atomic operations are defined next.

- Computation Operation.* $\otimes(R, A, B, sel)$ executes the 128-bit fixed operation $R = A \otimes B$, where $\otimes \in \{+, -, *, <, >, >>\}$. For $\pm$ operation, $sel = 0$ indicates that A/S is an addition operation; otherwise it is a subtraction operation. For shift operation, “L”, “RL”, and “AR” refer to logic left shifter, logic right shifter, and arithmetic right shifter, respectively.
- Condition Operation.* $IF(c_1, c_2, \oplus, R, a, b)$, where $\oplus = 0, 1, 2$ refer to the conditions $c_1 < c_2$, $c_1 = c_2$, and $c_1 > c_2$. If $c_1 \oplus c_2$ is satisfied, then $R = a$; otherwise $R = b$.
- Jump Operation.* This atomic operation realizes the loop or branch of other atomic operation sequences.
- Lookup table Operation.* $LOOKUP_T(R, A, func)$ returns R from the lookup table, which is indexed by A and $func$.
- Count Leading Zero Operation.* $Count_LZ(R, A)$ returns R , which is the number of the leading zero of A .
- Normalize Operation.* $NORMAL(R, func, F, E)$ normalizes the mantissa F and exponent E into Quad format result R according to the function type $func$.

The Appendix describes the atomic operation sequences. Table I summarizes the types of atomic operations for the Quad basic operations and Quad elementary functions, that employ the combination of polynomial approximation (or Newton method) and lookup table. Thus, for each elementary function, a $LOOKUP_T$ operation reduces the iteration of Newton's method or the degree of polynomial, thereby improving the computational efficiency. In the evaluation of the Quad addition, logarithm, and sine/cosine functions, the subtraction of two nearly equal numbers may occur. This process leads to the uncertain number of leading zeros. Thus, the $Count_LZ$ and $SHIFT$ operations are executed before normalizing the mantissa into the Quad format.

Based on the preceding analysis, we can decompose the Quad basic operations and elementary functions with a common group of atomic operations. For example, in the atomic operation sequence for Quad multiplication (as shown in the Appendix),

Table I. Atomic Operation in Quad Elementary Functions

Function	Add	Mult	Div	Sqrt	Exp	Ln	S/C
A/S	Yes	Yes	Yes	Yes	Yes	Yes	Yes
MULT		Yes	Yes	Yes	Yes	Yes	Yes
SHIFT	Yes		Yes	Yes	Yes	Yes	Yes
Lookup.T			Yes	Yes	Yes	Yes	Yes
Count.LZ	Yes					Yes	Yes
Condition	Yes			Yes	Yes		Yes
Jump			Yes	Yes	Yes	Yes	Yes
Normalize	Yes	Yes	Yes	Yes	Yes	Yes	Yes

“Yes” indicates the inclusion of the atomic operation in the corresponding Quad elementary functions.

multiplication is used to compute the product of mantissas, while an addition is used to compute the exponent. Finally, a normalization operation normalizes the product and exponent into the Quad format. Several types of computation and condition operations are employed in the two-level range reduction performed for the Quad exponential function, as shown in Figure 3(a). Only the computation operations are included in polynomial evaluation, whereas the lookup table and normalization operations are used in reconstruction. This is the same for other Quad elementary functions. Thus, it is now possible to implement the Quad arithmetic in a unified hardware framework which consists of the multiple atomic operation units.

3.3. Advantages of the VLIW Architecture in Supporting Atomic Operations

The atomic operation sequences and data dependencies between these atomic operations vary for different elementary functions. Thus, organizing and utilizing these atomic units effectively is the key technique in designing the hardware structure of a Quad arithmetic. Moreover, realizing the property of the scalability for new functions is another important consideration in ensuring the variety and versatility of complex elementary functions for different applications.

The VLIW technique [Fisher 1983] is effective in exploiting the Instruction-Level Parallelism (ILP). The technique simultaneously executes numerous operations on multiple basic units based on a fixed schedule sequence. The advantages of the VLIW technique listed next are oriented well to the unified coprocessor for Quad arithmetic based on atomic operation sequences.

- VLIW architecture offers significant computational power with less hardware resources. Since the operation scheduling and conflict resolution are handled by the compiler, the VLIW processor does not need the complicated scheduling hardware. Thus, more logic can be used to build the high-precision arithmetic units that are used to implement the atomic operations.
- The static scheduling and optimization of the VLIW architecture efficiently exploit the ILP for the fixed hardware algorithms of Quad basic operations and Quad elementary functions. It is possible to schedule the atomic operations in the design of these functions, such that the data dependencies between atomic operations are maintained under the control of custom VLIW instruction. The resulting explicitly parallel technique of VLIW instruction and pipeline parallel scheme can lead to improved performance.
- The VLIW processor can achieve high scalability and customizability. Under the control of specific VLIW instructions, the atomic operation units are combined into special-purpose hardware for Quad elementary functions. Moreover, this custom VLIW processor can be easily extended to implement Quad coarser-grain functions

for some domains. Moreover, extending the processor is more efficient than a general-purpose processor implementation.

4. IMPLEMENTATION OF THE VLIW COPROCESSOR

This section presents the organization of the VLIW coprocessor for the IEEE-754 Quad elementary functions (QP_VELP), the idea of which is similar to the concept of horizontal microcode in Wilkes [1951]. The coprocessor uses the unified hardware to evaluate various Quad basic operations and Quad elementary functions.

4.1. The QP_VELP Structure

Figure 2(a) shows that the QP_VELP mainly consists of a parallel computation module, a control state machine module, a two-level VLIW instruction RAM module, and an update VLIW instruction module. The control state machine module controls the running of QP_VELP via the change of two program counters (PC1 and PC2). The update VLIW instruction module controls the initialization of VLIW instruction RAM and allows it to support new elementary functions and key program kernels.

The evaluations of Quad basic operations and Quad elementary functions are accomplished by the parallel computation module through the explicitly parallel technique of the custom VLIW instruction. This module consists of four on-chip memory modules, control logic, and multiple basic 128-bit fixed-point arithmetic units.

The four on-chip memory modules comprise the lookup table, ROM-C, and two RAM-M. The Lookup-Table, which is a 512×10 -bits ROM with a single port, stores the initial approximation for Newton's method based on the lookup table method, such as the initial reciprocal value for the division operation and the approximation of reciprocal square root for the square root operation. ROM-C, a 512×128 -bits ROM with two read ports, stores constants such as the coefficients of polynomial and the initial value for the polynomial approximation method such as exponent and logarithm. Each RAM-M is a 32×128 -bits RAM with two read/write ports. So, two RAM-Ms provide four 128-bit read or write ports and serve the role of register file as in a general-purpose processor to buffer the middle data. These on-chip memory modules are the major data sources for the atomic operation units.

The atomic operation units are divided into the functional and computational groups. The first group, which includes the lookup table operation, the count leading zero operation, and the normalize operation, appears once for a given elementary function. Thus, only a single unit is instanced in the QP_VELP unit. In Figure 2(a), the atomic operations in the computational group, which includes the 128-bit multiplier (n_M), adder/subtractor ($n_{A/S}$), and shifter (n_S), are executed frequently. The 128×128 pipelined truncated multiplication unit is the largest hardware module in QP_VELP and occupies more than 40% of the area. However, Estrin's scheme for the polynomial, described in Section 5.2, fully utilizes the pipeline parallel of the 128-bit multiplier. Thus, $n_M = 1$ achieves good area-performance balance. The utilization of this multiplier is greater than 50%, as shown in Figure 4(e). The slightly lower hardware costs for the adder/subtractor and shifter, two adder/subtractors ($n_{A/S} = 2$), and two shifters ($n_S = 2$) are employed to improve the performance. The latency of the 128-bit multiplier is 6 cycle and that of the other atomic operation units are all one cycle.

A fully associative bypass network, as the data bus shown in Figure 2(a), is designed to connect these on-chip memories and atomic operation units. This network is composed of ten 128-bit 8:1 selectors to provide operands for atomic operation units, and occupies about 9% of the total area in QP_VELP. However, this network can provide 1280-bit internal bandwidth, so these operation units can be executed in parallel to improve the performance of QP_VELP.

Fig. 2. The QP_VELP structure.

4.2. 128-Bit Truncated Multiplier

The QP_VELP uses a 128-bit internal interface and computation precision. It is required that the error should be less 2^{-128} that for all basic operation units. Therefore, a 128-bit truncated multiplier is presented to reduce resources, delay, or power consumption. The idea is to save the computation of some of the less significant columns in the multiplication array, under the condition where the error is smaller than 2^{-128} .

Next, we determine the maximum number of calculated columns, denoted by $(p + k)$ in Figure 2(b). The error of truncated multiplier comes from two sources: approximation error (ε_A) and rounding error (ε_R). Here, ε_A is the sum of the truncated columns that

satisfy

$$0 \leq \varepsilon_A \leq 2^{-256} \sum_{i=1}^d i * 2^{i-1} = (d-1)2^{d-256} + 2^{-256}.$$

And ε_R is the error of rounding the $(p+k)$ -bits intermediate result to p bits, with the rounding to nearest mode that satisfies $\varepsilon_R \leq 2^{-129}$. Thus, $\varepsilon_A \leq 2^{-129}$ and $d \leq 120$.

Figure 2(b) shows that the 128-bit truncated multiplier uses eight guard bits. It consists of a 64×64 multiplier, two 32×32 multipliers, four 16×16 multipliers, twenty-four 8×8 multipliers, and some logic. The area of this multiplier is approximately 56% of the full 128-bit multiplier.

4.3. Two-Level VLIW Instruction RAM Scheme

The VLIW instruction RAM stores the custom VLIW instruction sequence for each elementary function. It is an important part that ensures the good performance, flexibility, and hardware cost of the QP_VELP. In the current study, a two-level VLIW instruction RAM scheme is presented, as shown in Figure 2(a).

The first level of VLIW instruction RAM, called the compressed VLIW instruction module, stores the custom VLIW instruction sequences for Quad basic operations (Quad addition and Quad multiplication), common elementary functions (Quad division, square root, exponential, logarithm, sine, and cosine), and parts of coarser-grain functions. Since Quad basic operations and common Quad elementary functions are the basic components for Quad applications, the VLIW instruction sequences for these Quad operations are solidified into QP_VELP with ROM. Thus, QP_VELP can accomplish basic Quad arithmetic without updating VLIW instructions.

In order to enhance the scalability of QP_VELP, part of the compressed VLIW instructions RAM is implemented with RAM. Special VLIW instruction sequences can be custom for some Quad coarser-grain functions, which are the basic functions in one domain. The Update VLIW instruction module loads these sequences into VLIW instruction RAM. Therefore, the QP_VELP can implement these coarser-grain functions with high efficiency.

The second level of VLIW instruction RAM is designed for the extension of some coarser-grain functions and key program kernels on QP_VELP. For example, the coarser-grain function $e^x - x$ can be divided into two continuous Quad operations (Quad exponential and Quad addition), so the VLIW instruction sequences for Quad exponential and addition are called ordinally to evaluate $e^x - x$. The second level is built with an address sequence RAM, which stores the initial address of scheduled functions in compressed VLIW instructions RAM. Thus, each word in this level includes a control field and an address field, as shown in Figure 2(c).

The two-level VLIW instruction RAM is reloaded by the update VLIW instruction module to implement coarser-grain functions and key program kernels for some applications. The width of this configure path is 64 bits.

A. Compression scheme for VLIW instruction

The code density is the primary disadvantage of the VLIW architecture. In each cycle, the one-to-one correspondence among functional units should be kept in the custom VLIW instructions. This process introduces the “horizontal” no-ops for unused units, which are the main sources of inefficiency for code density. Therefore, in the QP_VELP design, a code compression scheme based on the flag is presented in the first level of VLIW instruction. We built one flag, called *Sel_Fu*, for each functional unit in the *Use_Flag* RAM. The *Sel_Fu* was set to 1 if the corresponding functional unit was used in this cycle. Otherwise, it was set to 0. A control signal RAM was built for each unit (Fu_i), which merely stored the used control signal, indicating that the corresponding flag was valid ($Fu_i = 1$). The address generator logic was then built to access the

corresponding signal RAM. The control field for each unit is defined in Figure 2(c), where *sel_x* is used to select the data source for the computation unit, *addr_x* is the address field for on-chip memory, *A/S* is the selecting signal for addition or subtraction, and *en_a* is the written enable signal.

There is a need to add a flag and address generator logic for each functional unit in the proposed code compression scheme. However, the distributed control signal RAM for each functional unit is advantageous to the ASIC placement and layout. Moreover, this scheme only stores the control signal for a used functional unit; thus, it can avoid wasting the VLIW instruction RAM fully. The compression ratio is relative to the frequency of use (P_{Fu_i}) and the width of control signal ($Width_{Fu_i}$) for each functional unit. Both satisfy the following relationship.

$$Compression_Ratio = \frac{\sum_{i=1}^t (1 + P_{Fu_i} \cdot Width_{Fu_i}) + Width_{control2}}{\sum_{i=1}^t Width_{Fu_i} + Width_{control2}}$$

B. The QP_VELP Work Procedure

The run of QP_VELP is controlled by the control state machine module, which changes the value of PC1 and PC2 to access different VLIW instructions from the two-level VLIW instruction RAM for the corresponding Quad program kernels (Quad basic operations and elementary functions are regarded as special program kernels). The QP_VELP works through the following stages.

Stage 1. Initiation. We loaded the custom VLIW instruction sequences of coarser-grain functions and key program kernels in some application domains into the two-level VLIW instruction RAM.

Stage 2. Start-up. Upon receiving the *Start* signal, we fixed the initial address (*PC2*) of the second level in VLIW instruction RAM according to the program kernels (*Func*). Meanwhile, *PC2* indexes the address of the first level (*PC1*) for common elementary functions meant for subsequent calculation.

Stage 3. Calculation. Initial data (*Indata*) are stored into RAM-M. The corresponding address (*PC1*) then reads the control signal for each functional unit and control state machine. The execution of the parallel computation module is controlled, and the *PC1* value is changed according to the control field in the VLIW instructions.

Stage 4. Switch. Upon completion of the calculation of this elementary function, we added the *PC2* value by 1 to begin the calculation of the next Quad elementary function. The value of *PC1* and *PC2* is controlled by the control state machine module.

Stage 5. Finish. Stage 3 and stage 4 are repeated until all elementary functions for the given program kernel are completed. Then, the *Over* signal is generated and the results are written into the result FIFO.

4.4. Implementation of Elementary Functions

We built the atomic operation sequence to ensure the scalability and customizability of QP_VELP. Each elementary function was considered to be in isolation. Then, we customized the corresponding VLIW instruction sequence, which was integrated into QP_VELP, to yield the desired result and achieve high performance. In the subsequent discussion, we took the Quad exponential function as an example to illustrate the details of customizability for elementary functions.

Step 1. Building an atomic operation sequence for elementary function. We choose a suitable hardware algorithm, where the error analysis and the trade-off between area and performance are made. Subsequently, we implement the corresponding elementary

Fig. 3. Implementation of the Quad exponential function.

function with a series of atomic operations that can be executed by basic arithmetic units.

For example, the Quad exponential algorithm ($y = e^x$) is based on the lookup table and the Taylor series. As shown in Figure 3(a), the atomic operation sequence consists of three steps. The two-level range reduction scheme was carefully designed to guarantee the precision of results. The trade-off is between the size of lookup table and the evaluation latency of the Taylor series, which is used in Estrin's scheme. The precision of an initial value of 6 indicates better balance, thus, the lookup table is organized into a 64×128 ROM and the degree of polynomial is set at 15.

Step 2, Mapping the atomic operation sequence to the VLIW instructions. In this step, we mapped the atomic operation sequence into the custom VLIW instructions in accordance with the data dependence among them. Through static scheduling and optimization, we used as few VLIW instructions as possible to obtain the lowest latency. As in the exponential example in Figure 3(b), and the schedule of polynomial approximation with Estrin's scheme, the pipeline of the multiplier and the explicitly parallel atomic operation units are used to improve performance.

Step 3, Packing VLIW instructions into the hardware functions. We used two methods to pack the custom VLIW instructions into the QP_VELP. For common Quad elementary functions, the corresponding VLIW instructions are solidified into ROM in the second level of VLIW instruction RAM. For the extension of coarser-grain functions and key program kernels, the VLIW instructions are written into the first (or second) level of VLIW instruction RAM with the help of the update VLIW instruction module.

5. OPTIMIZATION SCHEMES ON QP_VELP

In this section, we improved the performance and scalability of the QP_VELP by introducing Estrin's scheme for polynomials and by using the extension scheme of coarser-grain functions, respectively.

5.1. Estrin's Scheme for Polynomial

As described in Section 2, polynomial approximation is a significant method for the evaluation of elementary functions. In QP_VELP, two adders with one stage pipeline

Fig. 4. Polynomial evaluation on QP_VELP.

and a 128-bit pipelined multiplier with six stages evaluate the polynomial $p(x) = a_0 + a_1x + \dots + a_nx^n$, where n is polynomial degree. Polynomials can be evaluated using numerous schemes, such as *Direct*, *Horner*, and *Estrin's* schemes.

The Horner scheme is written as

$$p(x) = a_0 + x * (a_1 + x * (a_2 \dots + x * (a_{n-2} + x * a_n) \dots))$$

and is implemented using fused multiply and add blocks. It requires the least computation cost (n additions and n multiplications). However, the Horner scheme is a sequential structure that necessitates in serial the addition, shift, and multiplication operations. Thus, $7n$ cycles are required for the *Horner* scheme on QP_VELP.

For the Direct scheme, $2n-1$ multiplications are required to evaluate the components (x^i and a_ix^i , $1 \leq i \leq n$). However, more internal parallelism can be exploited with a pipelined multiplier. The evaluation order as illustrated in Figure 3 in Lei et al. [2011] can be used to schedule as many operations as possible between two data dependence operations to reduce data hazard stalls and keep the pipeline full. As shown in Figure 4(c), when the degree of a polynomial (n) is greater than 8, $2n + 12$ cycles are required to evaluate this polynomial.

Estrin's scheme attempts to overcome the serialization of the *Horner* scheme while still being reasonably close to the optimal in terms of the number of multiplications. This scheme can be written as [Muller 2006]

$$p(x) = (a_0 + a_1x) + (a_2 + a_3x)x^2 + ((a_4 + a_5x) + (a_6 + a_7x)x^2)x^4 + \dots$$

where the number of multiplication operations is $n + \lceil \log_2(n) \rceil$ as shown in Figure 4(d), and the multiplication and addition can be performed in parallel or in a pipelined fashion. Similar to the Direct scheme, to keep the pipeline full, parallelism among the multiplications and additions must be exploited by finding sequences that can be overlapped in the pipeline. First, we build the dataflow graph for Eq. (1), where the nodes refer to operations in Eq. (1), and the weight of the edge is the latency of the corresponding operation. The graph of a degree-7 polynomial is taken as an example in Figure 4(d). Then, the maximum latency of each operation to the final result is evaluated through the modified DAG-shortest-paths algorithm [Cormen et al.

2001]. Finally, we schedule the operations according to the maximum latency and the data dependence among them. Thus, the operations in the critical path are executed first to obtain a lower bound on the latency.

Figure 4(c) shows the polynomial execution cycle using the three schemes. Note that Estrin's scheme executes fewer multiplication operations; thus, it is better than the Direct scheme (14 versus 21) and is employed in the current study.

5.2. Extension Scheme for Coarser-Grain Functions

We can easily customize the VLIW instruction sequences in the QP_VELP to evaluate coarser-grain functions for several special application domains. Customization can be achieved in two ways.

For some coarser-grain functions, it is composed of multiple Quad basic operations and common elementary functions, such as $\log_2(1 + 2^x) = \ln(1 + e^{x \cdot \ln 2}) / \ln(2)$ in the logarithm number system and $\ln(-\ln(x))$ in the Monte Carlo algorithm. The first method is to call the multiple VLIW instruction sequences of common elementary functions, and initializes the call sequences in the second level of the VLIW instruction RAM. Sequential execution is required for the multiple common elementary functions. For example, to evaluate $y = \ln(1 + e^{x \cdot \ln 2}) / \ln(2)$, we can execute the VLIW instruction sequences of Quad multiplication ($t1 = x \cdot \ln 2$), Quad exponential function ($t2 = e^{t1}$), Quad addition ($t3 = 1 + t2$), Quad logarithm function ($t4 = \ln(t3)$), and Quad multiplication ($y = t4 * (1/\ln(2))$) in order. Through the integration of multiple basic Quad functions into more complex ones, this method improves the performance of coarser-grain functions by reducing the communication overhead.

The second method is similar to the implementation of the Quad elementary function. The corresponding VLIW instruction sequence requires customization and initialization into the first level of VLIW instruction RAM, such as $\frac{\log(x+1/2)}{x-1/2}$, $\frac{x}{\sqrt{x^2+y^2+z^2}}$, and $\sqrt{x^2 + y^2 + z^2}$, which is used to evaluate the distance or velocity in three-dimensional simulation. Compared with the first method, this approach achieves higher performance through one-time reduction of the normalization operation and the scheduling of more ILP in the design of the VLIW instruction sequence.

Moreover, the QP_VELP provides a scalability mechanism to accelerate the Quad key program kernels. The implementation method is the same as the first method for Quad coarser-grain functions. In our experiment in Section 7, we use the QP_VELP to evaluate key program kernels in the SAR radar application.

6. ACCELERATOR CHIP OF THE QP_VELP ARRAY

This section presents the structure and ASIC implementation of the accelerator chip with a QP_VELP array.

6.1. Structure of Accelerator Chip

The performance of Quad arithmetic is improved using a Quad arithmetic accelerator chip (QAA) equipped with a QP_VELP array; this is used to evaluate batch Quad functions. Figure 5 shows that this accelerator mainly comprises the following modules: task allocation, result recycle, PE array, large result buffer, DMA controller, and other control logic.

The task allocation module assigns computation tasks to each PE in a round-robin mode. This is meant to balance the tasks and guarantee that results are evaluated in order. At the same time, the task allocation module writes the number of executed PE into Result_Order_fifo. The result recycle module receives the results from the result FIFO, specified by Result_Order_fifo, and writes them into the result buffer, which is a 8192×128 RAM with one port.

Fig. 5. Structure of the accelerator with a QP_VELP array.

The PE processor array consists of $4n$ FIFOs (n initial data FIFOs, n result FIFOs, and $2n$ function FIFOs) and n QP_VELP units. At the start, each QP_VELP initiates the same functions in VLIW instruction RAM. Therefore, these units can independently complete Quad elementary and coarser-grain functions, as well as key program kernels.

The active DMA mode implements data communication between the Quad arithmetic accelerator chip and the host. The DMA controller module controls the accelerator and the data transfer between memory and the accelerator chip without occupying CPU time. The implementation procedure is described next.

Stage 1. Start up DMA. The host activates the DMA controller module with the function type, the evaluated number of functions, as well as the initial addresses of the original data and results.

Stage 2. Reading original data and computing function. The DMA controller begins reading the original data from memory in order, and sends them to the task allocation module. Thus, reading operations can overlap with the accelerator calculation. Results are then stored in the result buffer module.

Stage 3. Writing the results. After the valuation of the QP_VELP arrays, the DMA controller writes the results in the memory.

In summary, the running time of the Quad arithmetic accelerator (T_{ACC}) includes time for the startup DMA ($T_{Startup_DMA}$), computation (T_{Cal}) reading the original data from the memory to the accelerator (T_{Read}), and writing the results in the memory (T_{Write}). Generally, the start-up DMA time is the latency between the CPU and accelerator chip and is smaller than other parts. With the DMA controller, reading operations can overlap with the accelerator calculation, so that the total running time is

$$T_{Acc} = T_{Startup_DMA} + \max\{T_{Read}, T_{Cal}\} + T_{Write}.$$

For the evaluation on large data, $T_{Startup_DMA}$ is far less than the $\max\{T_{Read}, T_{Cal}\}$ and T_{Write} . Thus, in this case, the running time is proportional to amount of calculated and transmission data.

6.2. Synthesis of Accelerator Chip on ASIC

The proposed hardware design, including the QP_VELP unit and Quad arithmetic accelerator chip, is implemented on ASIC using structural-level Verilog HDL, which is simulated with ModelSim 6.5h, and synthesized using Synopsys Design Compiler with TSMC 45-nm CMOS standard cell libraries respectively. All the syntheses on ASIC are tuned for delay optimizations with maximum effort and reported after placement and routing.

Table II provides the area, power, and delay estimation results of the QP_VELP unit; it also shows the Quad arithmetic accelerator chip equipped with a varying number

Table II. Synthesis Result with TSMC 45nm CMOS Standard Cell Libraries

Module	Area (μm^2)	Power (mW)	Delay (ns)
128-bit multiplier	138,935	98.34	0.70
QP_VELP	326,614	171.97	0.70
QAA(4 QP_VELP)	2,023,531	910.84	0.72
QAA(8 QP_VELP)	3,346,204	1614.9	0.72

of QP_VELP units. Although the design optimization technique reduces the hardware cost, the area for the 128-bit pipeline truncated multiplier takes the largest area consumption module, occupying roughly 42.5% of the QP_VELP. Meanwhile, most of the other areas implement the on-chip memories in the parallel computation module and the two-level VLIW instruction RAM.

The achievable maximum frequency of QP_VELP unit is 1.43 GHz. Meanwhile, the critical delay occurs in the first stage of the 128-bit multiplier to generate multiple partial products. The QP_VELP unit can be exploited for a deeper pipeline by performing more stages in order to achieve higher clock frequency. However, evaluating polynomial and elementary functions requires more cycles, thereby enlarging the latency for the given functions.

For the Quad arithmetic accelerator chip with 4 QP_VELP units and with 8 QP_VELP units, 64.6% and 78.1% of area is used to instantiate the QP_VELP units, respectively. In order to reduce the area cost, the result buffer is implemented with one port RAM. It consumes about 21.1% of area for an accelerator chip with 8 QP_VELP units. The maximum power dissipation of 1.61 W for the 8 QP_VELP units is much lower than the 80 W consumed by an Intel Xeon E5620 CPU.

In the following, this accelerator with 8 QP_VELP units, run at 1.39 GHz, is employed to accelerate scientific applications with Quad arithmetic.

7. EXPERIMENTS

We verified the correctness of the proposed designs through a prototype of the QP_VELP unit and Quad arithmetic accelerator with the QP_VELP array built on the FPGA board. Both the FPGA prototype and ASIC accelerator chip use active DMA mode to transfer data between host and accelerator. The differences lie in the bandwidth and latency. The performance of the ASIC implementation is predicted on the condition that the bandwidth and latency between the host and ASIC accelerator are similar to the DDR3 1333 MHz memory, that is, bandwidth is about 10GB/s and latency is about 65-ns [Kaseridis et al. 2011].

Next, we applied the 256-bit arithmetic in MPFR library [Fousse et al. 2007], MPFR3.0.0, developed by Hanrot et al., to measure the results accuracy. As a base for performance comparison, we measured delay time by applying the Quad arithmetic, that is, REAL(16), on the Intel Visual Fortran Compiler (V10.1.014) for Intel 64 [Intel 2012], which features an automatic vectorizer that can generate SSE, SSE2, SSE3, and SSE4 SIMD instructions for Intel processors. The software environment includes a host PC with Intel Xeon E5620 CPU at 2.40 GHz and 32GB DDR3 1333 MHz memory, microsoft Windows 7, and Microsoft Visual Studio 2005 at a level O3 release compiler optimization.

7.1. FPGA Prototype

As shown in Figure 6, the FPGA prototype mainly consists of two FPGAs (i.e., Virtex-5 XC5VLX50T-1FF1136 and Virtex-6 XC6VLX760-1FF1760) and two 2GB DDR2 DRAM modules. The XC6VLX760 chip implements the proposed design. The XC5VLX50T chip

Fig. 6. Structure of FPGA prototype.

provides a link between the XC6VLX760 and host PC through an eight-line PCI-express bus with a bandwidth of 900MB/s.

The QP_VELP unit and Quad arithmetic accelerator with eight QP_VELP units on XC6VLX760-1FF1760 FPGA synthesized by ISE 12.3 achieved maximum frequency of 225 MHz and 198 MHz, respectively, which are much lower than that of the ASIC implementation, because the circuit is emulated with a large number of configurable elementary blocks and network of wires on FPGAs.

7.2. Precision and Performance of QP_VELP

Precision comparison. Figure 7(b) shows the precision test results of QP_VELP for Quad basic operation as well as Quad elementary and coarser-grain functions. For each Quad function, a test set is randomly generated from the defined area. Each test set includes 100,000,000 elements. Finally, results are obtained by comparing the results of the QP_VELP unit and 256-bit arithmetic in the MPFR library.

For each function, a corresponding VLIW algorithm is designed, in which the internal precision is carefully planned and the guard words are used to guarantee the results' accuracy, thus faithful rounded results [IEEE 2008], the error of which is smaller than 1 ULP (unit of the last place), can be obtained for all inputs. However, it is very difficult to guarantee correctly rounding, in which the error is smaller than 0.5 ULP, for all inputs because of the so-called Table Maker's Dilemma [Lefevre et al. 1998], which requires exceedingly high intermediate accuracy to obtain a correct rounded result. Specifically, Lefevre and Muller reported that the worst case in correct rounding requires 158 bits of the intermediate accuracy in the double-precision exponential function [Lefevre et al. 2001]. Therefore, in the current study, one bit error may occur due to the rounding operation, and the percentage of correct rounding is over 99% for the basic Quad operations and common Quad elementary functions.

For the second implementation method of coarser-grain functions, such as the CF1-CF5 shown in Figure 7(a), the VLIW instruction is customized similar to that of the common elementary functions; moreover, rounding operations are executed only once. Thus, a faithful rounded result (one bit uncertainty at most) is guaranteed for all inputs. For the first implementation method, such as CF6-CF9, a two-bit uncertainty occurs for a very small part of the results due to the accumulation of rounding errors.

Performance comparison. Table III compares the performance of the Quad basic operation and common elementary functions on an Intel CPU and ASIC.

The Intel Fortran performance in Table III is the average time of 100,000,000 random inputs. Compared to this performance, one QP_VELP unit with 1.4 GHz can achieve a 5.7x average performance improvement. The current study used the polynomial approximation (or high-order Newton method) with high convergence for hardware implementation, and maximized the highest possible level of parallelism with Estrin's

Table III. Performance Comparison between Intel Fortran and QP_VELP Unit on ASIC, where “CRC” Refers to the Compression Ratio of VLIW Code and “UoM” Refers to the Utilization of 128-Bit Truncated Multiplier

<i>Func</i>	Method	Intel Fortran	QP_VELP (1.4 GHz)					
			Cycle	Latency	Speedup	CRC	ILP	UoM
$x \pm y$	[Dou 2005]	40.8 ns	11	7.9 ns	5.19	36%	0.77	0%
$x \times y$	[Dou 2005]	70.3 ns	8	5.7 ns	12.30	31%	0.27	13%
x/y	Newton	153.5 ns	39	27.9 ns	5.51	33%	0.55	23%
$\sqrt{x}$	Newton	203.3 ns	47	33.6 ns	6.06	45%	1.22	49%
e^x	Taylor	140.9 ns	57	40.7 ns	3.46	42%	0.91	40%
$\ln(x)$	Taylor	151.1 ns	58	41.4 ns	3.65	50%	1.47	52%
$\sin(x)$	Taylor	244.9 ns	70	50.0 ns	4.90	43%	1.13	49%
$\cos(x)$	Taylor	241.2 ns	70	50.0 ns	4.82	43%	1.13	49%

scheme for polynomials. However, the Intel Fortran library encountered unavoidable system overhead for software emulation (e.g., function call), although the Quad arithmetic in the Intel Fortran library is a special optimization according to the processor features. Compared with our previous works in Lei et al. [2011], the execute cycle for Quad elementary functions is reduced by a factor of 2 to 12 due to Estrin’s scheme for the polynomial evaluation.

If one QP_VELP module is integrated into the CPU for Quad arithmetic and the datapath of CPU is 128 bits, we can read a 128-bit data from the register file of the CPU or write a 128-bit result to the register file in one clock cycle. Therefore, for two-operand basic Quad operations in QP_VELP three cycles are added to read two 128-bit operand and write one 128-bit result, and to one-operand Quad elementary functions in QP_VELP two cycles are added to read one 128-bit operand and write one 128-bit result. The average performance boost with this hardware module will be $5.2\times$.

Due to the strong data dependency between the basic operations for Quad elementary functions, the available ILP is limited. Therefore, the Quad elementary functions are latency-dominated kernels. In the current article, Estrin’s scheme is employed to provide more ILP for the evaluation of the polynomial and a pipeline parallel schedule is used in QP_VELP to improve the utilization of hardware. The average ILP for common Quad elementary functions is about 1.1 and the utilization of 128-bit truncated multiplier reaches 41.4%.

There are many NOP operations in our scheduled VLIW instructions due to lowering of ILP, so the two-level VLIW instruction RAM scheme in QP_VELP for common Quad elementary functions greatly reduces the storage requirements for VLIW instructions, and the average compression ratio for these functions is approximately 42.3%

7.3. Performance of Quad Arithmetic Accelerator

In this subsection, we evaluated the performance and scalability of the Quad arithmetic accelerator using Quad basic operations, elementary functions, coarser-grain functions, and several key program kernels on a set with 10,000,000 random inputs. We also used the SAR radar application on a $4k \times 4k$ SAR row data. The Intel Xeon E5620 CPU supports a hyper-threading technique (i.e., four physical cores and eight virtual cores). Thus, in order to compare the performance of the Quad arithmetic accelerator, we implemented two versions of software approaches, namely, single-thread and eight-thread based on the OpenMP parallel technique. For multithreading implementation, the test dataset is divided into eight one-dimensional arrays with the same size, and the *Sections* directive in OpenMP is used specify the enclosed section for the evaluation of the elementary function in one test array.

Fig. 7. Precision and performance comparison between different platforms.

(A) Quad Basic Operations and Elementary Functions

Comparing the single-thread and the eight-thread parallel software approaches, the Quad arithmetic accelerator achieved an average speedup factor of 25.9 and 5.1, respectively, as shown in the front part of Figure 7(a). However, the speedup for Quad addition and multiplication operations is smaller than that for Quad elementary functions. Since the overhead for communication is bigger than computation overhead, the

Fig. 8. Program flow chart of Chirp scaling algorithm.

performance for Quad basic operations is limited by the IO bandwidth. As shown in Figure 7(d), the ratio of execution time to the total running time for Quad addition and multiplication are 22% and 16%, respectively.

(B) Coarser-Grain Functions

A larger computation granularity induces a smaller ratio between the communication and computation overheads. Thus, performance of coarser-grain functions is better than that of the Quad elementary functions. As shown in Figure 7(d), the average ratio of execution time to total time is 82.6% for coarser-grain functions. This accelerator outperforms the serial and the parallel software approach by factors of 27.6 to 94.9 and 5.5 to 17.9, respectively. The performance of the second way in the implementation of coarser-grain functions (CF1-CF5) is higher than that of the first way (CF6-CF9). This is because more atomic operations can be scheduled and more ILP can be exploited in the VLIW instruction sequence design.

(C) Key Program Section and CS Application

We used several key program kernels in two applications (i.e., UZ1 and CS), to evaluate the scalability of the Quad arithmetic accelerator on a larger computation granularity. The two key program kernels in the UZ1 application, which is used to solve unbalanced stiff equations and simulate the nonequilibrium phenomena of radiation, include division, square root, exponential and logarithm functions, as described in Zhou et al. [2008]. The proposed accelerator outperforms the parallel software approach by factors of $10.3\times$ and $12.5\times$, respectively.

As shown in Figure 8, the program flow diagram of the SAR radar algorithm (Chirp scaling algorithm) consists of four two-dimensional (2D) FFT, three compression factor generators, and three factor compression computations. The key operation in the 2D FFT is the butterfly computation, and that in factor compression is the complex multiplication. Figure 7 also presents the equations for the three compression factors. Therefore, the entire SAR radar application can be decomposed into the aforesaid five key program kernels.

Compared with three compression factor generations, the FFT butterfly and complex multiplication are merely composited with Quad addition and multiplication operations. In addition, the computation granularity is small. Therefore, the FFT butterfly and complex multiplication demonstrated lower performance than the factor generations. However, relative to only the addition and multiplication operations, the ratio

between the communication overhead and computation overhead is reduced. Thus, the Quad arithmetic accelerator outperforms the parallel software by a factor of $7.7\times$.

In the SAR radar application on a $4k \times 4k$ SAR row data, the five key program kernels have a ratio of 18:3:1:1:1. The FFT butterfly computation serves as a significant factor in the entire SAR radar application. Therefore, the performance of the Quad arithmetic accelerator for SAR radar applications is similar to that for the FFT butterfly computation. However, since the data dependency is more complex in the SAR radar application, the speedup between parallel and serial software is only $3.7\times$. Therefore, the proposed accelerator can outperform the parallel version by a factor of $9.3\times$.

8. CONCLUSIONS

In this article, we have presented a custom VLIW coprocessor for the IEEE-754 Quad basic operations and elementary functions in the unified hardware, equipped with multiple basic atomic operation units. Several optimization schemes, such as the 128-bit truncated multiplier, the two-level VLIW instruction RAM scheme, and Estin's scheme for polynomial evaluation, are proposed to achieve high performance, scalability, and customizability with less hardware cost. Finally, the FPGA prototype and ASIC implementation of the Quad arithmetic accelerator are built with a QP_VELP array.

The proposed accelerator provides advantages in terms of better performance and power consumption in the evaluation of Quad elementary and coarser-grain functions, as well as Quad applications.

Our future work aims to explore the feasibility and potential performance improvement of equipping the processor with a Quad arithmetic accelerator. Moreover, we will further explore the autogeneration technique of VLIW instructions in a compiler.

REFERENCES

- AKKAS, A. 2008. Dual-mode floating-point adder architectures. *J. Syst. Archit. Embed. Syst. Des.* 54, 12, 1129–1142.
- AKKAS, A. AND SCHULTE, M. J. 2006. Dual-mode floating-point multiplier architectures with parallel operations. *J. Syst. Archit.* 52, 10, 549–562.
- BAILEY, D. H. 2005. High-precision floating-point arithmetic in scientific computation. *Comput. Sci. Engin.* 7, 3, 54–61.
- BRENT, R. P. AND ZIMMERMANN, P. 2010. *Modern Computer Arithmetic*. Cambridge University Press.
- BRODTKORB, A. R., DYKEN, C., HAGEN, T. R., HJELMERVIK, J. M., AND STORAASLI, O. O. 2010. State-of-the-art in heterogeneous computing. *Sci. Program.* 18, 1, 1–33.
- CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms*, 2nd ed. The MIT Press.
- DETREY, J., DINECHIN, F. D., AND PUJOL, X. 2007. Return of the hardware floating-point elementary function. In *Proceeding of the 18th IEEE Symposium on Computer Arithmetic (ARITH'07)*. 161–168.
- DOU, Y. 2005. 64-bit floating-point fpga matrix multiplication. In *Proceedings of the 13th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'05)*. 86–95.
- DOU, Y., LEI, Y., WU, G., GUO, S., ZHOU, J., AND SHEN, L. 2010. FPGA accelerating double/quad-double high precision floating-point application for exascale computing. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS'10)*. 325–335.
- ERCEGOVAC, M. 1973. Radix-16 evaluation of certain elementary functions. *IEEE Trans. Comput.* 22, 6, 561–566.
- FISHER, J. A. 1983. Very long instruction word architectures and the eli-52. In *Proceedings of the 10th Symposium on Computer Architecture (ISCA'83)*. 140–150.
- FOUSSE, L., HANROT, G., LEFEVRE, V., PELISSIER, P., AND ZIMMERMANN, P. 2007. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.* 33, 2, 1–15.
- GOK, M. AND OZBILEN, M. M. 2008. Multi-functional floating-point maf designs with dot product support. *Microelectron. J.* 39, 1, 30–43.
- HUANG, L., MA, S., SHEN, L., WANG, Z., AND XIAO, N. 2012. Low cost binary128 floating-point fma unit design with simd support. *IEEE Trans. Comput.* 61, 5, 745–751.

- IEEE. 2008. Standard for binary floating point arithmetic ANSI/IEEE standard 754-2008. <http://grouper.ieee.org/groups/754/>.
- INTEL. 2012. Intel compilers and libraries. <http://software.intel.com/enus/articles/intel-compilers/>.
- KASERIDIS, D., STUECHELI, J., AND JOHN, L. K. 2011. Minimalist open-page: A dram page-mode scheduling policy for the many-core era. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 24–35.
- LEFEVRE, V., MULLER, J. M., AND TISSERAND, A. 1998. Towards correctly rounded transcendentals. *IEEE Trans. Comput.* 47, 11, 1235–1243.
- LEFEVRE, V., MULLER, J. M., AND TISSERAND, A. 2001. Worst cases for correct rounding of the elementary functions in double precision. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH'01)*.
- LEI, Y., DOU, Y., SHEN, L., ZHOU, J., AND GUO, S. 2011. Special-purposed vliw architecture for ieee-754 quadruple precision elementary functions on fpga. In *Proceedings of the 29th IEEE International Conference on Computer Design (ICCD'11)*. 219–225.
- MULLER, J. M. 2006. *Elementary Functions: Algorithms and Implementation*, 2nd ed. Birkhauser, Basel, Switzerland.
- OSBERMAN, S. F. AND FLYNN, M. J. 1995. Implementing division and other floating-point operations: A system perspective. In *Proceedings of the Conference on Scientific Computing and Validated Numerics (SCAN'95)*. 18–24.
- PAUL, G. AND WILSON, M. W. 1976. Should the elementary functions be incorporated into computer instruction sets? *ACM Trans. Math. Softw.* 2, 2, 132–142.
- RANEY, R. K., RUNGE, H., BAMLER, R., AND WONG, F. H. 1994. Precision sar processing using chirp scaling. *IEEE Trans. Geosci. Remote Sensing* 32, 4, 786–799.
- SCHWARZ, E., SMITH, R., AND KRYGOWSKI C. 1999. The s/390g5 floating point unit supporting hex and binary architecture. In *Proceedings of the 13th IEEE Symposium on Computer Arithmetic (ARITH'99)*. 258–265.
- VALLADO, D. A., CRAWFORD, P., HUJSAK, R., AND KELSO, T. S. 2006. Revisiting space track report #3. In *Proceedings of the AIAAAstro Dynamics Specialists Conference and Exhibit*.
- WILKES, M. V. 1951. The best way to design an automatic computing machine. In *Report of Manchester University Computer Inaugural Conference*.
- WRATHALL, C. AND CHEN, T. C. 1978. Convergence guarantee and improvements for a hardware exponential and logarithm evaluation scheme. In *Proceedings of the 4th Symposium on Computer Arithmetic (ARITH'78)*. 175–182.
- ZHOU, J., DOU, Y., LEI, Y., AND XU, J. 2008. Double precision hybrid-mode floating-point fpga cordic co-processor. In *Proceedings of the IEEE International Conference on High Performance Computing and Communications (HPCC'08)*.

Received January 2013; revised March 2013; accepted April 2013