First International Workshop on
HyperTransport™ Research and Applications

Proceedings

of the

1st International Workshop on HyperTransport Research and Applications

WHTRA 2009

Editors
Holger Fröning
Mondrian Nüssle
Pedro Javier García García

ISBN: 978-3-00-027249-3

February 12th, 2009, Mannheim, Germany

University of Heidelberg, Computer Architecture Group

Proceedings of the
First International Workshop on HyperTransport Research and Applications (WHTRA2009)
Feb. 12th, 2009, Mannheim, Germany

Exploiting the HTX-Board as a Coprocessor for Exact Arithmetics

Fabian Nowak Rainer Buchty David Kramer
and Wolfgang Karl
Institute of Computer Science & Engineering
Universität Karlsruhe (TH)
Zirkel 2
76131 Karlsruhe, Germany
{*nowak, buchty, kramer, karl*}*@ira.uka.de*

Abstract

Certain numerical computations benefit from dedicated computation units, e.g. providing increased computation accuracy. Exploiting current interconnection technologies and advances in reconfigurable logic, restrictions and drawbacks of past approaches towards application-specific units can be overcome. This paper presents our implementation of an FPGA-based hardware unit for exact arithmetics. The unit is tightly integrated into the host system using state-of-the-art HyperTransport technology. An according runtime system provides OS-level support including dynamic function resolution. The approach demonstrates suitability and applicability of the chosen technologies, setting the pace towards broadly acceptable use of reconfigurable coprocessor technology for application-specific computing.

1 Introduction

One major requirement of certain numerical applications is computation accuracy. Depending on the application, using standard floating-point implementation does not provide necessary resolution, especially due to accumulating rounding errors. This may lead to more required iterations, thus degrading the overall application performance. The solution to such problems is the use of dedicated precise arithmetic routines, ideally provided as a dedicated application-specific hardware accelerator.

Such approaches were already targeted in the past, but were typically hampered by insufficient hardware resources and, most notably, the used interconnection technology: peripheral buses usually do not provide significant bandwidth to enable a tight cooperation between an application running on the host CPU and the according hardware accelerator.

With the advances in programmable logic and interconnection technology, using so-called FPGAs as application-specific computation devices tightly integrated into the host system via current interconnection technology such as HyperTransport delivers a promising concept for arbitrary coprocessor solutions. Suiting the programmers' needs, such a solution should be easily accessible, not requiring a new programming model or dedicated, heavyweight execution environments.

We therefore chose the original work of Kulisch et al. for exact arithmetics [21], which delivered impressive numbers on raw computation throughput. Unfortunately, the back then available state-of-the-art interconnection bus turned out to be severely limiting. Using current HyperTransport and FPGA technology, we adopted the original design principle. Based on the UoH HTX-Board [2], a dedicated unit supporting exact arithmetics was developed. We furthermore provide a runtime system supporting dynamic resolution and mapping of function calls that enables seamless switching between standard and exact computation. Such a runtime system not only fulfils the requirements for advanced, parallel arithmetics as mentioned before, but also for irregularly memory-demanding applications such as numerical applications on adaptive meshes and graph analysis problems [14] as it allows redirecting resource accesses, which is substantial to tackling these issues.

In the following section, we will give an overview over the field of dedicated application accelerators and their integration into the host system, outlining the individual peculiarities of each approach. Section 3 then introduces the concept of exact arithmetics as proposed by Kulisch, and its implementation on the UoH HTX-Board using the Open-Source HyperTransport Core [27]. An overview of our runtime system and its general usage is given in Section 4. Our experimental setup is presented in Section 5, its results are shown in Section 6. Finally, we give an outlook over future work in Section 7 after drawing some conclusions.

2 Related Work

A significant amount of research on the arithmetic part, dynamically reconfigurable application accelerators, and according runtime systems and programming models exists. For this paper, we will therefore restrict to some focal points, outlining the intended problem solution and eventual drawbacks.

Targeting the domain of error estimation, interval arithmetic was developed in the 1950s and 1960s [23] and is still actively researched today [25, 8].

Several approaches targeting hardware implementations of more exact arithmetics exist, ranging from multi-precision fixed-point vector MAC [29] to quad-precision floating-point units [11] or fixed-point computations. A major problem with floating-point operations is the accumulation of rounding errors: for this, the extension of the radix avoiding time-consuming rounding operations is a potential solution [26]. Using reconfigurable logic lays the foundation for arbitrary-precision arithmetic units on FPGAs for rational numbers [7]. While the cited approach lacks floating-point support, it demonstrates feasibility and achievable speed-up in comparison to software emulation by the GMP library [12].

As of now, a vast number of also commercially available hardware accelerators exists for different goals, among them acceleration of computation and algorithms on dedicated hardware. The ClearSpeed series of accelerators [5] might serve as an example of current state-of-the-art accelerator architectures, providing 96 floating-point processing cores. An even bigger number of processing cores is provided by current graphics cards being used as floating-point accelerators, also employing a multi-level memory hierarchy to overcome bus bandwidth limitations resulting from the used PCI express bus (PCIe).

All these hardware approaches have in common that interconnection bandwidth and therefore data transport between host system and accelerator becomes a bottleneck. This makes the use of such units only sensible when the gained speedup outweighs transfer time.

In order to avoid transport delay issues when tackling precision, the available precision can be extended inside floating-point units: in the IBM P6 [30], this technique is well-employed with buffering of intermediate results and providing further rounding operations in addition to those specified in the 754 standard. Still, precision is not enough for numerically unstable algorithms.

Approaches such as IRAM [24] and PIM [28] target the problem of data transport and transport latencies directly by calculating inside memory, what requires their own programming models as well. Both exact arithmetic and interval arithmetic are also available as software libraries for a large number of programming environments and systems, even bindings for high-level programming languages are available [20, 12, 9].

Reconfigurable logic offers the possibility of providing required arithmetic operations as demanded by the computation, therefore offering the same flexibility as software libraries but at significantly higher speed. With the advent of high-performance interconnection buses such as HyperTransport [16], this approach has gained widespread acceptance ranging from reconfigurable accelerator cards like Nallatech's FPGA Computing platforms [17] to reconfigurable supercomputing systems like Cray XD-1 [6]. On these accelerator cards, the available FPGAs are big enough to hold a couple of acceleration units.

Our approach follows the reconfigurability concept, making use of reconfigurable FPGA hardware to model a dedicated, high-performance acceleration unit focusing on exact arithmetics. To avoid limitations from communication bottlenecks, we employ HyperTransport as state-of-the-art interconnection technology. The case study for our approach is Kulisch et al.'s first implementation of a hardware unit for exact arithmetic. As mentioned before, this work suffered from bandwidth limitations of the PCI bus while the unit was sufficiently fast [18].

The use of application-specific and potentially reconfigurable hardware from within the application is targeted by several approaches ranging from description languages to integrated environments, consisting of dedicated compilers and runtime systems. The scope ranges from programming languages such as Handel-C [4], abstraction via ISA extensions such as MOLEN [31] and EXOCHI [32], and runtime environments such as LIME [15], and combined API/Runtime systems like the recent OpenCL [13].

The above solutions typically require either additional software layers for accessing accelerator hardware, are focused on a dedicated system setup, or both. What is missing from these approaches is an easy and lightweight method to dynamically resolve function calls at runtime so that a different implementation or a different library be used for the same function call. This is very desirable, as it offers the possibility to hide transport and computation latency by switching to different implementations as long as the hardware is occupied, and it is also important for dynamic systems where hardware resources are allocated and freed at runtime for the best mapping of an application onto the respective hardware, enabling more precise, faster or less power-consuming execution of the application.

We therefore developed a lightweight extension to the Linux OS's runtime system providing a method for dynamic control of function mapping as a convenient means to change between different arithmetic implementations as required by the running computation.

In the following, we will present the implementation and integration of the accelerator into HT-equipped systems, fo-

cusing on the hardware design, and will include software and runtime implications where appropriate.

3 Exact Arithmetics

As basic design of an exact accumulator, we implemented the approach proposed by Kulisch et al. [21]. The idea behind their approach is to avoid rounding results as much as possible, because rounding leads to accumulation of rounding errors after a couple of computations in conventional FPUs, such as adding small values multiple times to a rather big value. For double precision, the computation window is much larger than for single precision operation, but breaks with three more orders of magnitude as well. Such computation schemes however, are common to a wide range of numerical applications.

As already mentioned, their implementation in CMOS technology was fast enough, but did not keep pace with advances in processor technology as the interconnection relied on the PCI bus.

3.1 Concept

In order to avoid rounding, a different presentation than the conventional one consisting of the well-known sign-characteristic-mantissa encoding where numbers are composed like

$$number = 2^{exponent} * 1.mantissa \tag{1}$$

with m the length of the characteristics field and n the length of the mantissa field and $exponent = characteristic - 2^{m-1} + 1$ (i.e. the characteristic is the exponent plus the required bias), is needed for intermediate representation of the values, because with the exponent representation, only a "window" of a float number is precise.

The flat two's complement encoding suits very well as rational numbers can be stored in fix-comma representation easily enabling both very large and very small numbers to be represented at once:

$$number = whole\text{-}number.fraction \tag{2}$$

with a length of k and l respectively where $whole\text{-}number$ ranges from -2^{k-1} to $2^{k-1} - 1$ and $fraction$ is in between 2^{-l} and $\sum_{i=1}^{l} 2^{-i}$ for both positive and negative numbers as well as zero is possible. Considering IEEE 754 single-precision format with $m = 8$ and $n = 23$, $0x00000001 = 2^{-2^{m-1}+2-n} = 2^{-149}$ is the smallest representable number; whereas the largest number is $0x7FFFFFFF = 2^{2^{m-1}} - 2^{-n} < 2^{128}$. Thus, for accumulating these precisely, the following inequation must be held:

$$k >= 128; l >= 149 \tag{3}$$

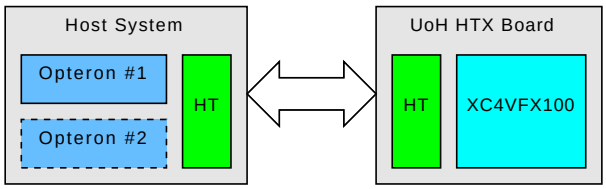


Figure 1. HTX testsystem

For double-precision with $m = 11$ and n=52, the requirements are huge:

$$k >= 1024; l >= 1074 \tag{4}$$

Adhering to these requirements does however only allow accumulation within this range, but accumulation of the largest possible values would result in infinity. Thus, additional bits are recommendable; and for multiplication of float-format encoded numbers, we need to double the amount of bits at least. Accommodating the possibility for a billion accumulations only does not seem very reasonable as numerical applications frequently iterate over meshes with some million elements per dimension. Hence, Kulisch proposes 86 additional bits for single-precision and 92 bits for double-precision accumulations.

With this in mind, when implementing an exact accumulator for single-precision arithmetics, $2 * 277 + 86 = 640$ seem to suffice.

3.2 Implementation

For accessing the exact arithmetics unit (EAU), we used the HyperTransport (HT) evaluation design of the University of Heidelberg with a Xilinx Virtex-4 FX100 in an AMD Opteron system (cf. Figure 1).

The EAU as an accelerator unit is wrapped in a memory-mapping HT interface, which is linked to the IO buffer wrappers for the evaluation board and the HTX socket, as is shown in Figure 2. Note that the reset and clock wires are not drawn for simplicity. A simplification unit has been inserted to merge the posted and non-posted requests. The memory-mapping interface allows addressing of up to 16 EAUs; however, only 14 MAC-equipped EAUs fit onto the FPGA, while 16 simpler accumulation-only units fit very well. The resource usage is discussed in more detail in Section 6.1. The design runs at a clock frequency of 100 MHz, hence allowing a theoretical unidirectional throughput of 800MB/s with 16 bits per each clock edge of the 200 MHz HT clock.

Accumulation of the decoded IEEE-754 represented values happens in small blocks of 32 bits each. The exponent determines both the block's position inside the big accumu-

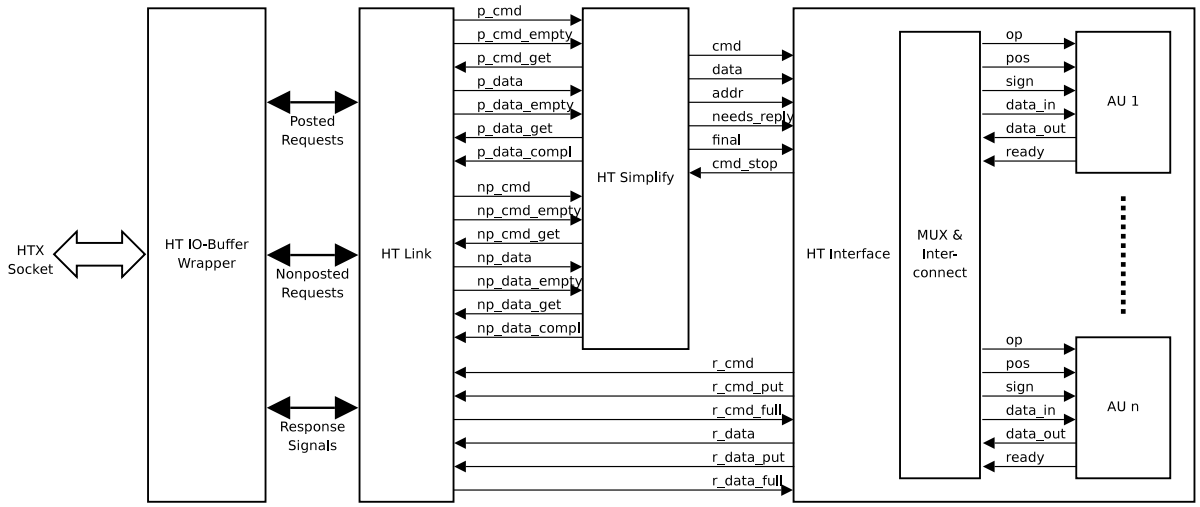


Figure 2. Device architecture

lation register and the proper shifting of the mantissa value before accumulating.

When adding the possibility for multiplication of two floating-point format data values, the sum of parts of their exponents is used for addressing the register position, with each data being shifted by the lower parts of its own exponent only. Using dual-port registers, reading the data to be accumulated to while writing the last sum to the last position is possible during the same cycle.

By using bit masks, fixing carry-resolution becomes an easy task: it requires only one additional cycle. Speaking of cycles, a regular accumulation of a single value requires up to 6 cycles after transmitting the data, where the last cycle may even overlap with the first cycle of another computation; so already with 6 accumulators we could hide the total latency completely if the calculation was split and data transfer lasted one cycle only. Multiply-accumulate, however, is finished no earlier than 18 cycles after starting the transfer of the first data word. The data transfers account for 8 of these cycles, among which 6 are due to the internal communication structure between the HT Link and the HT interface. These 6 additional transport cycles cannot be hidden by simply addressing other accumulations units as only one connection is possible during theses transfers. But one cycle can be saved when starting the next computation already during the last cycle. This leads to a latency of at least 17 cycles. With data transfers for a multiply-accumulate operation taking 3 and 5 cycles respectively, it is clear that three accumulation units can already make up for at least the computation latency; a higher number does not provide any additional advantage with regard to processing speed and hiding this latency. Throughput is therefore limited to less than 50MB/s because of the aforementioned 8 internal transport cycles when multiplying-accumulating. 4 cycles are needed for computing the addendum, and another three for writing the shifted sum in blocks to the accumulation register. This way, a clock rate of more than 100MHz can still be obtained, making the implementation suitable for the HTX bus specification, but currently suffering from high latency.

Speaking of addressing the accumulation units via the HT bus, all the units on the HT-core enabled HTX Board are memory-mapped into the processor's address space with 4kB page size, where different areas of a memory-mapped page denote different operations such as value-reading, writing flags or adding a product to the current accumulator value. This is illustrated in Figure 3. For example, writing to a page offset of 136 will prepare a multiply-accumulate operation and require a subsequent write to offset 140 for the next single-precision value to be multiplied. When reading from the first five words (i.e. addresses 0, 4, 8, 12, 16), the register value is returned as single-precision floating-point number rounded to 0, away from 0, towards negative infinity, towards positive infinity or towards nearest number, respectively.

All the accumulators are completely independent from each other except for the data transfer, which is multiplexed by the HT Memory-Mapping Interface to only one of the EAUs based on the target address. This interface passes the bundles of command, data, and address to the arithmetic units as operation code and associated data, thereby also handling replies.

The HT Simplify module then merges posted and non-posted requests, preferring the posted requests, passes them to the memory-mapping interface and cares for fetching and buffering new commands or data.

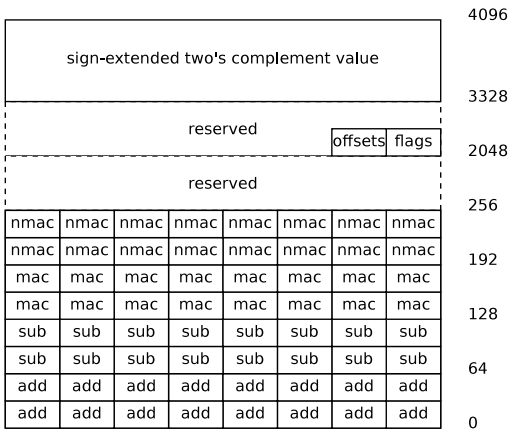


Figure 3. Memory-mapping of EAUs

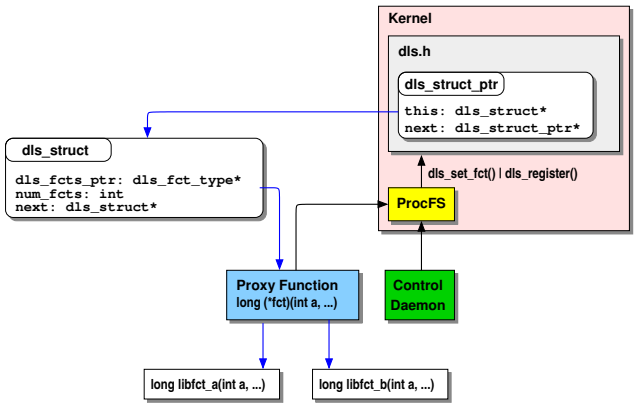


Figure 4. Switching of proxy functions

4 Runtime System

When executing numerical programs exploiting external hardware units, it is crucial to allow regular program continuation during calculation in hardware while also maintaining flexibility for choosing which software, hardware or hybrid implementation of a function to use. We developed a runtime system [3] coming in two different flavours, a Global Linking System (GLS) and a Dynamic Linking System (DLS). Both can be controlled via the Proc file system.

4.1 Global Linking System

The Global Linking System, also referred to as GOT-based Linking System for the Global Offset Table in the Linux kernel's task management, extends the lazy-linking technique of the kernel by means of the Executable and Linkable Format (ELF) [19]: the kernel extracts the function names from an application's ELF file and resolves each function symbol when the respective function is accessed for the first time. Using the Proc file system [22], the GLS resets a function symbol's structures and target function so that upon the next access onto the symbol, the dynamic linker resolves the symbol again. Function alternatives both from inside an application and from the linked libraries can be used for function switching.

The GLS is completely independent from compilers and programming languages used and also suits closed-source, proprietary software. However, although providing flexibility for mapping functions to alternate implementations, different threads will always use the same function; so regular program continuation cannot be granted when access to hardware resources used in a function implementation is exclusive-only.

4.2 Dynamic Linking System

To overcome the missing support for individual per-thread function mapping, the DLS changes an application's task state segment (TSS) by adding function proxies per thread for each reconfigurable function. Hence, apart from linking to an additional library, the program code must contain statements indicating which functions may be exchanged at runtime, and must then indicate which functions of which libraries are appropriate implementations for the proxy. An application may hence start a software and a "hardware" thread, each thread adding a software or hardware implementation function to the list of possible alternatives. A quick overview of the kernel extensions, the Proc file system interface and the proxy functions is given in Figure 4.

The DLS can be used in two different ways, either the static version or the dynamic version.

In the static version, right from the beginning the function proxy points to a valid function implementation. The pointer may be changed via the Proc file system to a different implementation. In contrast, in the dynamic version the unresolved proxy calls a fix-up function, which in return starts the dynamic loader and adjusts the function pointer. When changing that proxy's target at runtime, a fix-up function must be given first that again resolves the proxy's pointer.

In both versions, functions can be exchanged even from inside the application itself.

4.3 Employment

Targeting matrix multiplication using exact arithmetics as a first example for numerical applications, there are two levels at which the runtime system can be employed: at the coarse algorithmic level for selecting which of the specific implementations should the proxy function be resolved to,

and at the fine-grained operation level for deciding whether calculation on regular FPU, in enhanced-precision library or on external hardware unit is preferred.

The runtime system offers ultimate flexibility for both the programmer, the user, and system administrators by (dis)allowing access onto special hardware for more precision and numerical robustness or, in contrast, fast computation. For most of the systems, the programmer wants to use the DLS, where he picks the static linking variant if all implementations are already known at compile time, or the dynamic version, if new implementations become available at runtime, e.g. for long-running applications where new acceleration functions are constantly being developed. The system hence is also usable as runtime testing system for algorithms and their implementations.

5 Architecture

Our approach of combining exact arithmetics with a general runtime system for program adaptivity is based on the afore-mentioned HTX Testsystem (cf. Figure 1). A modified Linux kernel in version 2.6.20 runs both the runtime system and the programs, which may connect to the UoH HTX Board via the HyperTransport Interface of the AMD Opteron that runs at 2.0 GHz.

On the FPGA of the HTX Board, up to 16 EAUs are created and interconnected as illustrated in Fig. 2. The system clock is 100 MHz, cooperating with the 200 MHz HT clock (double-edge clocking) at a bandwidth of four 16 bit-wide transfers per system clock cycle.

The runtime system is responsible for mapping accumulate and multiply-accumulate operations onto the respective library functions, enabling emulated exact arithmetics, hardware support for exact arithmetics, or execution in regular single/double floating-point format on the Opteron's FPU.

The described 100 MHz system clock is due to hardware constraints. In order to foster maximum use of the available accelerator hardware, we employ the dynamic linking system to allow different threads to run with different implementations concurrently, allowing computation in parallel.

6 Results

In this section, we first present our hardware implementation results for the Virtex-4 FX100 1152-10 as available on the UoH HTX-Board [10], obtained with XST (ISE 9.2.04). We then give the results of some preliminary benchmarks results for different libraries both without and with use of our runtime system. We conclude by showing some potential benefit for numerical applications that suffer from convergence problems.

6.1 Implementation Results

Targeting ultimate flexibility, we implemented a basic EAU with accumulation support only and an enhanced unit with additional multiply-accumulate support so that based on available hardware resources, the operating system, dynamic runtime system or the user herself can decide how much and which coprocessor support to exploit in the current setup.

Hence, we present in Tables 1 and 2 the results of the synthesis runs for 1, 2, 4, 8, and 16 accumulator-only and MAC-extended EAUs, respectively. As we can see, the HT interface logic and the mapping interface only make up for a minor part of the system and with increasing number of EAUs, the routing costs rise enormously. Note that the number of needed resources depends strongly on several factors such as software, host system used for synthesis, synthesis settings, and the hardware description itself. Consequently, the results cannot be regarded as accurate, but only indicate the approximate amount of required resources.

The actually better results for the multiplication unit arise from the need to specify very strict requirements for synthesis and mapping.

Rather independent from the number of EAUs, the maximum theoretical design frequency is about 120 MHz for the accumulate-only design and about 100 MHz for the MAC design, independent of the number of EAUs. The critical path is determined by selecting from the large register; this can however be circumvented by splitting the large register into several smaller resources or mapping it directly onto the hardware BRAM resources. The HyperTransport core requires its client to run at 100 MHz or 200 MHz respectively, using a differential clocking with 200 MHz for the bus interface and merging the 16 bit connection into an internal 64 bit wide bus interface.

6.2 Runtime Results

For the results below, we indicate the mean value of ten runs, measured in clock ticks as reported by the CPU's timestamp counter.

First, we measured the time in clock ticks for both an IJK and IKJ matrix multiplication with arbitrarily chosen sizes of 30x20 * 20x16 and 100x100 * 100x16. The programs were compiled with `-O2`. Note that the enormous runtime for the 1-MAC-unit version is due to the necessity to read values in between of two same operations or to do any other operation freeing the virtual HyperTransport channel for the accumulator. Also, the software-emulated MAC support for high-precision arithmetics still sometimes produces erroneous results and cannot be compared therefore, but already gives a rough estimate. The results without any additional runtime support are given in Figure 5.

Table 1. Hardware implementation results for accumulation-only units

Resource	Used Resources					Available
	1 EAU	2 EAUs	4 EAUs	8 EAUs	16 EAUs	
Slice Flip Flops	4,835	5,350	6,409	8,536	12,783	84,352
Occupied Slices	7,795	9,501	13,701	19,254	31,923	42,176
4 input LUTs	10,324	13,274	19,312	31,477	55,788	84,352
Logic	10,066	12,888	18,654	30,291	53,562	
Route-thru	234	362	634	1,162	2,202	
Shift registers	24	21	24	24	21	
RAMB16s	26	27	29	33	41	
Equiv. gate count	1,811,662	1,902,807	2,085,366	2,451,628	3,184,303	

Table 2. Hardware implementation results for multiply-accumulation units

Resource	Used Resources					Available
	1 EAU	2 EAUs	4 EAUs	8 EAUs	16 EAUs	
Slice Flip Flops	5,085	5,890	7,505	10,728	17,167	84,352
Occupied Slices	7,955	10,718	14,846	24,008	39,141	42,176
4 input LUTs	11,355	15,535	23,868	40,570	73,868	84,352
Logic	11,059	15,077	23,084	39,134	71,129	
Route-thru	272	434	760	1,412	2,715	
Shift registers	21	21	21	21	24	
RAMB16s	26	27	29	33	41	
DSP48s	4	8	16	32	64	160
Equiv. gate count	1,821,103	1,923,091	2,126,585	2,533,808	3,346,401	

Execution times increase for the IKJ multiplication as both compiler optimization makes up for the non-optimally aligned memory accesses and few potential for overlapping computation in hardware is given due to the layout of the algorithm. The runtime of the software-emulated MAC unit is a rough estimate only as the implementation hampers from incorrect algorithmic implementation and may produce wrong results.

Furthermore, we evaluated three different routines for getting the sine value of the program's argument. The first implementation is the regular call of the sine function in the math library of the GNU C Library, the second uses a look-up table as has been done for example in Quake III Arena to offer sufficient speed while achieving good precision on those days' PCs, and finally, the Taylor series:

$$sin(x) = \sum_{i=1}^{n} (-1)^{i+1} \frac{x^{2i-1}}{(2i-1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \quad (5)$$

The Taylor series implementation is much slower than the others, but it allows external coprocessor support for accumulating each summand in hardware and also swapping the construction of the summands into the hardware accelerator. Figure 6 presents the runtime results for different implementations with and without support for exact arithmetics plus the number of iterations needed until the result was stable. For the three different routines, the runtime system has been used to allow measuring runtimes of implementation alternatives by simply switching the function pointers from inside the application at program runtime, with the static linking of the DLS being sufficient for this purpose. The result was stable after 9 iterations for double precision, 4 iterations for emulation without MAC support, and 5 iterations for the remaining runs.

6.3 Exact Arithmetics

Accuracy is achieved as shown in Table 3: with exact arithmetics, the result is more precise than with regular single-precision operation. The bad runtime for the MAC-enabled unit is again due to the additional operation needed for clearing the HT interface registers. Except for the double-precision runs, after 10 iterations the result is already stable. This is due to obtaining single-precision values only when reading the accumulator value: the convergent Euler series is also convergent for the iterations following iteration number 10, which hence do not influence the single-precision value to any extent. Reading double values from the EAU in subsequent implementations will produce different results. After 39 and 178 iterations respectively, the single and double precision windows are too small for storing the component values of the product to be accumulated.

A more detailed result[1] for the EAU can be obtained from the hardware accumulator when adding the subsequently read value to the intermediate coarse result value after subtracting it from the accumulator. The gain in precision with the MAC unit is ascribed to not losing the re-

[1] 2.71828183528879208097350783646106719970703125 (accumulator only) and 2.71828183518901056459071696735918521881103515625 (MAC-extended)

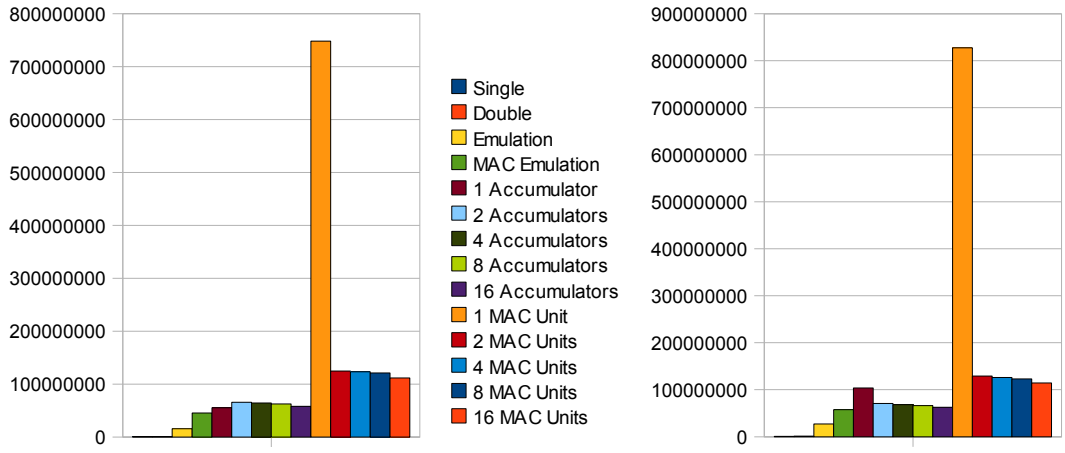
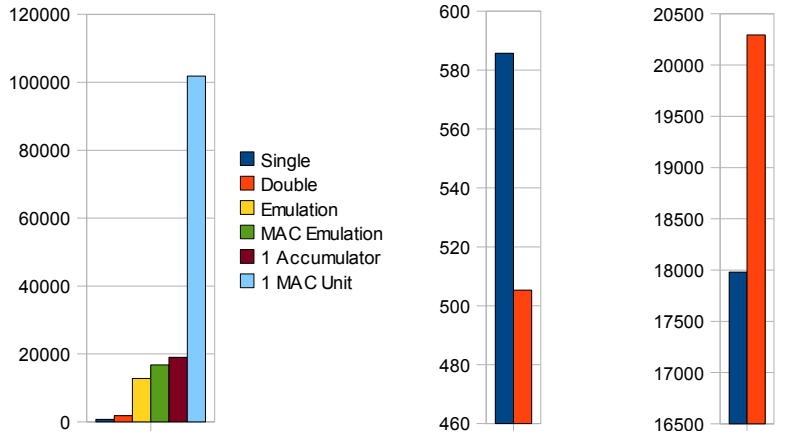


Figure 5. Comparison of runtimes for IJK and IKJ matrix multiplication with and without exact arithmetics



(a) Taylor series (b) Look-up table (c) GNU C Library

Figure 6. Runtimes and iterations for calculating the sine value of 1.0

Table 3. Runtimes and iterations for calculating the Euler number

Implementation	**Result**	**Runtime**	**Iterations**	
			Result fixed	Summand! = 0
Single Precision	2.7182819843292236328125	3,150	10	39
Double Precision	2.7182818284590455348848081484902650117874145507812 5	6,566	17	178
Emulation w/o MAC	2.71828174591064453125	19,935.9	10	39
Emulation w/ MAC	2.71828174591064453125	26,171.3	10	39
Accumulator	2.71828174591064453125	31,486.2	10	39
Multiply-Accumulator	2.71828174591064453125	188,945.7	10	39

mainders of the product. This clearly shows how accurate the obtainable results are in comparison to single-precision format. Similarly, when reading from the coprocessor register, the results of the Taylor sine routine are a little more precise than their native FPU counterpart.

With multiplication in hardware, we gain from not being limited to the format in use, i.e. single floating-point precision with a mantissa of 23 bits, but instead being able to accumulate the exact product onto the previous register value. With accumulation only, the multiplication has to be carried out on the host processor and only the rounded result, limited to the precision in use, can be accumulated, hence loosing valuable information.

For example, multiplying single-precision data converted to double-precision values would achieve a more precise representation of the product than with single-precision only; but as soon as the result is converted for adding regular single-precision values, the additional precision is lots. On the MAC hardware, though, the result of additional MAC and accumulation operations onto the previous multiplication result is still precise and the accumulated error due to rounding and format limits is much smaller. Of course, its additional precision will be lost as well when converting to single-precision format for further processing on the host processor.

7 Conclusions and Outlook

Developments in interconnection and FPGA technologies enable the use of reconfigurable logic as application-specific hardware accelerators.

In this paper, we showed an implementation of an exact arithmetics hardware unit using an FPGA-equipped HyperTransport device as a coprocessor. Through a lightweight runtime systems, dynamic per-function control of which implementation or software to use for calculations is made possible. The hardware is easy to use from a programmer's view and can be completely hidden from the user because the runtime systems offers the necessary abstraction and wrapping and because control is possible from within the application itself, if desired.

The arithmetics unit uses a wide fixed-point representation of the data, enabling accumulation of both very small and very large numbers altogether. The hardware unit is addressed via memory-mapping, which enables separate usage of up to 16 arithmetics units at once, thus speeding up parallel, separate computations by hiding latency. The HyperTransport bus is controlled by an AMD Opteron, the Virtex-4 is connected to the HT bus through an HTX board. For now, only single-precision is supported, but the increase in exactness due to not loosing the additional bits when multiplying and adding in our hardware in contrast to computing on regular floating-point units already proves the benefit of such architectures and proves both feasibility and reliability of such an arithmetic coprocessor for exact arithmetics.

The runtime system is a lightweight extension to the Linux kernel, altering the dynamic loading of libraries that can be controlled via the *procfs* interface. Alternate libraries may emulate hardware, access it directly, or offer fast and unreliable implementations – the user or a runtime system can choose, which one to use in his system based on availability and load.

We conducted several experiments regarding the basic operations required by the targeted numerical applications such as frequent multiply-accumulate as needed in solving linear equation systems. The results deliver further proof that supporting single-precision is far not enough when targeting real-world applications as the regularly obtainable data is not precise enough. We thus deduct that exact accumulation is only useful when appropriate means are given for retrieving those bits that cannot be represented in the regular floating-point format so that additional digits of a calculated high-precision value be obtainable. It also becomes more expedient when supporting double precision, offering more precision than regular double-precision without introducing the need for additional computation libraries such as QD [1].

Thus, we can conclude that 1) using coprocessor technologies for arithmetics is a valid and suitable approach, 2) HyperTransport fits well as interconnection technology, and 3) usage of exact arithmetics needs no longer be an issue for programmers due to the achievable enhancements of dynamic runtime linking.

For the hardware, our ongoing work includes extending the arithmetic unit by exact multiplication, decreasing overall latency and area, and increasing clock frequency of the implementation, thereby enabling usage for even small amounts of computations without a large amount of runtime overhead. Support for double-precision is also a nearby goal, both for reading double values, accumulating double values and accumulating double-precision products. Further plans for the runtime system include using it for a wider range of applications such as dynamic systems and debugging, but also incorporating reconfigurable hardware as memory-mapped devices into the operating system, which allows building and loading custom accelerators per application. Access onto the available resources will have to be managed then by offering only a few virtualized units to applications.

Acknowledgment

The authors would like to thank Reimar Döffinger for the initial work on the exact accumulation unit.

References

[1] D. H. Bailey, Y. Hida, K. Jeyabalan, X. S. Li, and B. Thompson. QD. Web site: `http://crd.lbl.gov/~dhbailey/mpdist/`.

[2] U. Brüning. The HTX board – a universal HTX test platform. Web site: `http://www.hypertransport.org/members/u_of_man/htx_board_data_sheet_UoH.pdf`.

[3] R. Buchty, D. Kramer, M. Kicherer, and W. Karl. A lightweight approach to dynamical runtime linking supporting heterogenous, parallel, and reconfigurable architectures. In *Architecture of Computing Systems – ARCS 2009, 22^{nd} International Conference*, Lecture Notes in Computer Science (LNCS), Delft, Netherlands, March 2009. GI e.V. to appear.

[4] Celoxica. Handel-C Language Reference Manual, 2001.

[5] ClearSpeed Technology plc. ClearSpeed Advance X620 and e620 Accelerator Boards, 2006. Web site: `http://www.clearspeed.com/products/cs_advance/`.

[6] Cray Inc. Cray XD1 Supercomputer, 2004. Web site: `http://www.cray.com/downloads/Cray_XD1_Datasheet.pdf`.

[7] E. Ej-Araby, I. Gonzalez, and T. El-Ghazawi. Bringing High-Performance Reconfigurable Computing to Exact Computations. *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 79–85, Aug. 2007.

[8] C. F. Fang, T. Chen, and R. A. Rutenbar. Floating-point error analysis based on affine arithmetic. In *Proc. IEEE Int. Conf. on Acoust., Speech, and Signal Processing*, pages 561–564, 2003.

[9] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2):13, 2007.

[10] H. Fröning, M. Nüssle, D. Slogsnat, H. Litz, and U. Brüning. The HTX-Board: A Rapid Prototyping Station. *Proceedings of the 3rd Annual FPGA World Conference*, November 2006.

[11] G. Gerwig, H. Wetter, E. M. Schwarz, and J. Haess. High Performance Floating-Point Unit with 116 Bit Wide Divider. In *ARITH '03: Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH-16'03)*, page 87, Washington, DC, USA, 2003. IEEE Computer Society.

[12] T. Granlund. The GNU MP Bignum Library, 2008. Web site: `http://gmplib.org`.

[13] K. Group. Khronos OpenCL API Registry. December 2008. `http://www.khronos.org/registry/cl/`.

[14] B. Hendrickson and J. Berry. Graph Analysis with High-Performance Computing. *Computing in Science & Engineering*, 10(2):14–19, March-April 2008.

[15] S. S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah. Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary. In *ECOOP 2008 Object-Oriented Programming*, volume 5142/2008 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2008.

[16] HyperTransport Consortium. Low Latency Chip-to-Chip and beyond Interconnect, 2005. Web site: `http://www.hypertransport.org/`.

[17] N. Inc. High Performance FPGA Computing Solutions for Defense and HPC, 2005. Web site: `http://www.nallatech.com/`.

[18] J. Kernhof, C. Baumhof, B. Höfflinger, U. Kulisch, S. Kwee, P. Schramm, M. Selzer, and T. Teufel. A CMOS Floating-Point Processing Chip for Verified Exact Vector Arithmetic. *Proceedings ESSCIRC '94*, pages 196–199, September 1994.

[19] J. Koshy. libelf by Example. Web site: `http://people.freebsd.org/~jkoshy/download/libelf/article.html`.

[20] U. Kulisch. The XSC tools for extended scientific computing. In *Proceedings of the IFIP TC2/WG2.5 working conference on Quality of numerical software*, pages 280–284, London, UK, 1997. Chapman & Hall, Ltd.

[21] U. W. Kulisch. *Advanced arithmetic for the digital computer: design of arithmetic units*. Springer, 2002.

[22] M. Tim Jones. Access the Linux kernel using the /proc filesystem. In *IBM developerWorks*, 2006. Web site: `http://www.ibm.com/developerworks/library/l-proc.html`.

[23] R. E. Moore. *Interval arithmetic and automatic error analysis in digital computing*. PhD thesis, Stanford University, Stanford, CA, USA, 1963.

[24] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent RAM. *IEEE Micro*, 17(2):34–44, 1997.

[25] J. G. Rokne. Interval arithmetic and interval analysis: an introduction. *Granular computing: an emerging paradigm*, pages 1–22, 2001.

[26] P.-M. Seidel. High-radix implementation of IEEE floating-point addition. *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on*, pages 99–106, June 2005.

[27] D. Slogsnat, A. Giese, M. Nüssle, and U. Brüning. An open-source HyperTransport core. *ACM Trans. Reconfigurable Technol. Syst.*, 1(3):1–21, 2008.

[28] T. Sterling, J. Brockman, and E. Upchurch. Analysis and Modeling of Advanced PIM Architecture Design Tradeoffs. In *SC'2004 Conference CD*, Pittsburgh, PA, November 2004. IEEE/ACM SIGARCH.

[29] D. Tan, A. Danysh, and M. Liebelt. Multiple-precision fixed-point vector multiply-accumulator using shared segmentation. *Computer Arithmetic, 2003. Proceedings. 16th IEEE Symposium on*, pages 12–19, June 2003.

[30] S. D. Trong, M. Schmookler, E. Schwarz, and M. Kroener. P6 Binary Floating-Point Unit. *Computer Arithmetic, 2007. ARITH '07. 18th IEEE Symposium on*, pages 77–86, June 2007.

[31] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte. The Molen Polymorphic Processor. *IEEE Transactions on Computers*, pages 1363–1375, November 2004.

[32] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 156–166, New York, NY, USA, 2007. ACM.