# An FPGA-based floating-point processor array supporting a high-precision dot product

Fritz Mayer-Lindenberg, Valerij Beller
*Institute of Computer Technology, Technical University of Hamburg-Harburg*
*Schwarzenbergstraße 95, D-21073 Hamburg, Germany*
mayer-lindenberg@tuhh.de

***Abstract -*** **This note reports on the design of a pipelined floating-point processor on a Spartan-III FPGA. It is implemented as a coprocessor to a novel, universal controller for pipelined data path designs that provides a high-level API and compiler support for general FPGA applications. The controller adds multi-threading and networking to the processor design, and the option of SIMD processing. The complexity issue of high precision floating point in an FPGA implementation is taken care of by efficiently implementing a recent algorithm of Rump that computes the dot product of two vectors at the same level of precision as a double precision processor yet using single precision operations only including a few non-standard primitives. For these special operations, our FPGA based processor actually outperforms hardwired floating-point DSP chips performing them in software. Through the inclusion of sequential control and networking our design provides a realistic estimate of the floating point system performance of FPGA in standard applications.**

## I. INTRODUCTION

Low-to-medium volume embedded systems often build on programmable or configurable hardware, typical components being microcontroller and DSP (digital signal processor) chips and the FPGA (the field programmable logic array). Hardwired processors are more cost effective for the standard functions and interfaces they implement, while the FPGA is most attractive when its special architectural features can be exploited, e.g. the availability of multiple separate memory blocks and multipliers, LVDS interfacing, and can implement non-standard operations and interfaces. Standard functions such as simple processors can be implemented as well with the remaining resources. There are a number of commercial and open-source processor designs [2,3,4]. Most are integer CPUs intended as auxiliary processors executing the 'software part' in a digital system. Actually, the use of simple CPU-like circuits within an FPGA is mandatory. FPGA circuits must be reapplied as fast as possible to changing data in order to use the FPGA efficiently, and therefore need sequential control. For such control, a fairly simple CPU with a versatile coprocessor interface appears as a better choice than a full-fledged system processor. The choice of floating point processors on FPGA is more restricted. A library of floating-point functions on FPGA is reported on in [16]. Floating point processors need to be based on a CPU architecture and are often described as modules within or coprocessors attached to a CPU. High-performance floating-point processing depends on pipelining which is not supported by the simple coprocessor interfaces of most standard processor designs. A pipelined floating-point data path design supporting the standard single-precision operations for normalized operands only is available at [4]. A comparison of it to ours is given in [6] and reveals a similar performance for add and multiply operations and a slightly higher complexity. A more cost-effective way to provide the standard floating-point capabilities to a digital system is to use some hardwired low-cost floating-point DSP chip, however. FPGA only and definitely becomes a viable choice for implementing arithmetic when non-standard operations or non-standard number formats such as logarithmic encodings [7] come in.

Our floating-point processor consists of a floating-point data path design with some non-standard functions and private memory for the floating-point data and a controller allowing the use of the data path for various algorithms. The controller also provides on-chip networking. Several controller plus floating-point or other coprocessor subsystems fit onto a medium-size FPGA. Compiler support is available for the individual processor and for networks of several processors.

## II. CPU-ON-FPGA DESIGN

The controller CPU is not dedicated to the floating-point processor design but is intended as a control and communications facility for general application-specific coprocessor designs. Another CPU design proposed for controlling custom compute circuits is proposed in [8]. Requirements on our CPU have been

- low complexity design with cop interface and support for cop input and output
- the CPU should permit useful programming with a small program memory
- extra cycles for calls, returns, branches etc. w/o a cop operation must be avoided
- coprocessors may be pipelined and generally need to serve several tasks/threads
- the FPGA may hold multiple, communicating CPU plus coprocessor subsystems

Our controller design is a 16-bit CPU design building on the CPU-II sketched in [5]. The original CPU design has been adapted to the special FPGA resources, in particular, to the dual-port feature of the RAM blocks. It interfaces to its local memory by two ports which seems to be a novel choice half way between Harvard and von-Neumann. The extra memory bandwidth is exploited in various ways including dual instruction fetches. The CPU provides an elaborate coprocessor interface and coprocessor instructions for the application-specific logic.

Communications between several CPU sub systems and I/O modules occur via a 16-bit bus defined as a part of the CPU architecture and supported by special instructions and a DMA channel. Fig.1 shows a typical CPU subsystem as supported by our architecture. A controller circuit for an external memory shared by the CPU circuits is optionally attached to the IO bus and application specific coprocessors are attached to the individual CPUs. In the present implementation, the memory controller is a protocol converter automaton requesting the memory transfers from a hardwired processor attached to the FPGA that provides the function of SDRAM control at low costs. Moreover, the implementation links the CPU subsystem to a board-level network supporting multi-FPGA applications with optional attached standard processors [19]. These facilities are available to every coprocessor design and let it gain scalability and become an array design.

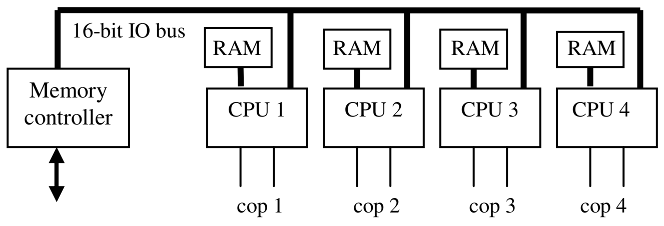

Fig.1: Typical CPU subsystem on a 3S1600E FPGA

The coprocessor interface not only generates control codes for the attached circuit, but also memory control signals and dual addresses for a private memory of the coprocessor as needed by many DSP algorithms using the CPU data registers as address registers. Fig. 2 shows the resulting, non-standard memory architecture. The separate coprocessor memory may be wider than the CPU memory if the data to be processed are so (for the floating-point coprocessor, a 32-bit wide data memory is used). This scheme directly extends to SIMD coprocessor arrays attached to a single controller CPU and having multiple data memories. These are treated as a single memory of multiple width and are addressed in parallel. The CPU further supports the coprocessor by organizing DMA block transfers between the IO bus and the coprocessor memory.

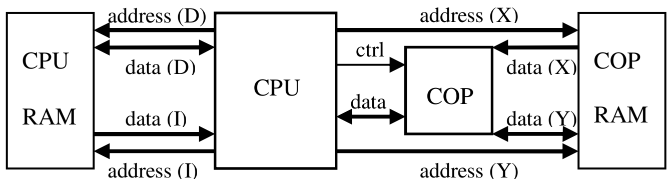

Fig.2: CPU plus coprocessor memory architecture

Pipelining support at the instruction level is by providing independent instruction fields for inputting to and outputting from the pipeline. Op-code and operand loads apply to its input stage, and store operations and status transfers to the output stage. For a coprocessor with a two-level pipeline an individual operation needs three coprocessor instructions (not three cycles):

```
cop  op        .. issue operation into pipeline,
               .. optional operand loads
cop nop        .. no operation, advance pipeline,
               .. could issue next op
cop nop        .. optionally store result here
               .. and/or issue next op
```

Pipelining is further supported by providing some simple multithreading facilities [9] not available in other soft processor designs. Coarse grained multi-threading hardware support by the CPU is for up to four threads. By implementing all pipelining registers of a coprocessor as register banks, fast context switches are possible, allowing e.g. an interrupt service routine to also use the coprocessor without having to store and restore its state. Fine grained multitasking is supported as well for all pipelined coprocessors (including the floating point coprocessor) by allowing coprocessor instructions from two threads to be executed in parallel. Then operations of the second thread can fill up pipeline slots that can't be used by the first. Both threads actually need to fill up their respective unused slots with cop-nop operations to advance the pipeline which are executed in parallel with operations from the other thread.

The choice of the standard controller design and its coprocessor interfacing scheme allow for further design tools and FPGA design automation. The CPU plus coprocessor systems and, more generally, parallel MIMD system on an FPGA built from such are supported by the compiler for the system-level language $\pi$–Nets [5]. This support actually extends to board-level networks of several FPGA chips and attached hardwired standard processors, and to being able to define application-specific coprocessor circuits in $\pi$–Nets, too, letting the compiler generate FPGA net lists extending the pre-compiled CPU subsystem net list. There is no explicit reference to the controlling CPU in this case. The FPGA simply performs a $\pi$–Nets application process based on the operations of a selected data type obeying to the same non-sequential semantics as a sequential processor would do. For applications using sequential floating-point operations, CPU plus floating-point coprocessor subsystems are synthesized. The FPGA net lists are further processed with the aid of the standard place and route and bit stream generation tools.

The design of FPGA circuits is thus essentially reduced to defining coprocessors conforming to the coprocessor interface of the CPU without having to care for sequential control and communications. Coprocessors can be reused as design modules, allowing new application systems to be configured from a mix of existing coprocessors. Coprocessors can also be considered as modules for partial reconfiguration similarly to the concept of runtime reconfigurable functional units proposed in the literature [17].

## III. Floating point coprocessor design

The single-precision (32-bit) floating-point coprocessor conforms to the coprocessor interface of the CPU and adds to the CPU what is missing to implement a full-featured embedded processor supplying integer and floating point operations. The coprocessor executes the basic add and multiply operations in a 4-stage pipeline which is advanced by cop instructions as explained before and also provides a combined ('fused') multiply/add. It operates at the full instruction rate of the CPU and achieves a peak throughput of 50Mflops. As usual, the reciprocal and square root operations are only supported by a low-precision starting value of an approximation to be carried out in software. All transcendental functions also rely on software evaluating polynomial approximations [18].

The coprocessor takes further advantage of the flexibility of FPGA implementations by adding to the standard operations a few non-standard ones. It is well known [11] that the true sum x + y of two floating point numbers x,y can be represented as

$$x + y = z \;\; f{+}\; e\,, \quad \text{with z being defined by}$$
$$z = x \;\; f{+} \;\; y$$

and f+ denoting the rounded floating point add operation. The error term e is a floating point number as well that can be computed using six extra floating point operations:

$$e = x \; f{-} \;((x \; f{-} \;(z \; f{-} \; u)) \; f{+} \;(y \; f{-} \;(y \; f{-} \; u))\,,$$
$$\text{with } \; u = z \;\; f{-} \;\; a\,.$$

Similarly, the true product of two floating point numbers can be expressed as

$$x * y = (x \; f{*} \; y) \; + \; h$$

with f* denoting the rounded floating-point multiply and h being a floating point number that can be obtained through as many as 17 standard operations [12], or 2 operations on processors providing the fused multiply-add with a single rounding step only. These operations are needed to reconstruct the rounding error in terms of available FPU operations.

Both error terms correspond to places that need to be computed anyhow during the standard processing for the floating point operations but which are used for rounding only and then dropped. In our design, they are computed and made available in parallel to f+ and f*. The standard multiply-add operation is complemented by one based on these extended operations. With the error terms, floating-point computations can, in principle, proceed without any loss of precision, adding extra places in extra error words.

As usually, pipeline stages are defined according to the data flow and the depth of the combinatorial functions. For the floating-point multiplication, the first pipeline stage unpacks the operands, calculates the exponent and multiplies the mantissas. The second stage performs normalization including the case of un-normalized operands. The third stage handles the required extra normalization for the error term and adjusts exponents. The fourth stage treats de-normalization in case of underflow (fig.3). The combined multiply-adds are executed at the full rate of the pipeline in dot product computations. The pipelined combined, extended multiply-add is implemented to efficiently support a higher precision dot product.

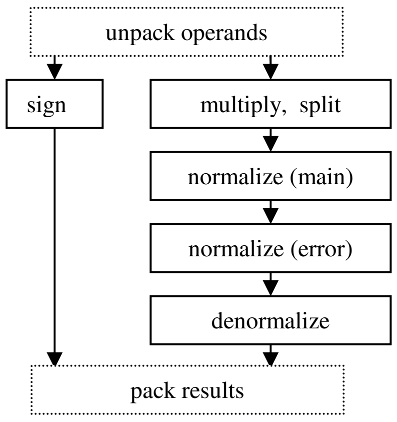

Fig.3: Data flow for the multiply operation

Floating point algorithms suffer from the accumulation of rounding errors. While a standard DSP chip like the Sharc [13] computes a fixed point dot product in full precision, the floating point version uses rounded operations only and produces invalid places in the final result. Lower relative errors result from double precision computations, but a double precision floating point processor is about four times as complex as a single precision one. Rump shows in [1] that by accumulating both the main multiplier outputs and the error terms, dot product approximations similar to computing with twice the number of places are obtained for every chosen rounding mode using the single-precision hardware. A modified algorithm using some extra primitives in which the dot product result is even produced up to an LSB is presented in [15]. With a special combined multiply add only one extra add is needed for the error term, thus performing the size-n dot product in about 2*n cycles only (the standard dot product takes n cycles). With an extra adder not included in the current design, the high-precision dot product can even be reduced to n cycles, too.

The high-precision dot product is one of the rare FPGA applications where the FPGA actually outperforms a state-of-the-art hardwired floating-point processor chip (see below). One can thus state that through implementing non-standard primitives high-precision floating point processing can be better supported on FPGA than on standard processors at the present time.

## IV. Comparing FPGA and DSP

The system shown in fig.1 and extended by four floating point coprocessors fills up a 3S1500 FPGA and achieves a system clock rate of 50MHz. Each coprocessor delivers a throughput of 50MHz for standard multiply-add operations and 25MHz for high-precision multiply-add operations. This amounts to a total throughput of 200MHz of standard and 100 MHz of non-standard multiply-add operations. We use these performance figures for a comparison with two hardwired floating-point processors. They are rather conservative and include costs for sequential control as required in most applications.

The hardwired single precision floating point processor ADSP21363 from Analog Devices' Sharc family achieves a throughput of 300MHz on standard multiply-add operations (600 MHz using SIMD) and costs about half as much as the FPGA chip, clearly indicating that the standard operations are provided at significantly lower costs by the standard chip. The non-standard extended multiply-add must however be realized in software by the Sharc which costs extra 23 operations per multiply-add. The rate for the high-precision multiply-add hence drops to about 12MHz (24 using SIMD), and the FPGA is the clear winner for it. The performance gap between the Sharc and the FPGA for the high-precision dot product becomes still larger if the second floating-point adder is implemented in the coprocessor, or if SIMD processing is applied on the FPGA, too.

A competing DSP chip from Texas Instruments, the TMS320C6727 also runs at 300MHz [10] and offers double precision at a rate of 150MHz. The processor is deeply pipelined yet without supporting fine-grained multithreading. This processor computes a double precision dot product at a higher rate than the FPGA with four CPU plus coprocessor subsystems and is preferable. In the SIMD configuration, the rates become similar, and with the extra adders, both achieve similar cost/performance ratios. The relation definitely changes in favor of the FPGA, however, if a double-precision floating-point coprocessor is implemented on the FPGA having the special operations to compute a dot product as in quadruple precision (and rounding the result to double precision).

Applications requiring both standard and non-standard operations and interfaces will be best served by mixed FPGA plus DSP systems. If the processor chips specialize in particular operations like integer and floating-point DSP chips do then a mix of different processors plus FPGA is most cost-effective. A design algorithm selecting a cost-effective mix for a given application has been outlined in [19].

## V. Conclusion

We have outlined the design of a floating point processor implemented on low-cost FPGA. On larger FPGA chips, several such processors or mixes of floating point processors and CPU subsystems with other or no coprocessors at all can be implemented. Our CPU design provides the required communications support for such purposes, and raises FPGA efficiency through its low control overheads and its multithreading support. The non-standard coprocessor interface supports the pipelining required for high throughput processing and facilitates SIMD processing and the use of wide data types. Even then, an FPGA based processor could hardly compete with hardwired chips of the same feature size for standard floating-point processing in terms of cost/performance.

The FPGA gains a specific advantage by supporting non-standard operations, in particular for high-precision computing. With the added error outputs a low-cost FPGA even outperforms a high-speed DSP. The design can be upgraded using a second adder to double dot product performance. High-precision embedded floating point processing thus seems to be better supported on FPGA than on mainstream processors at the present time. The same applies to the arithmetic operations using non-standard encodings of numbers.

The arithmetic performance of the FPGA can be raised by using SIMD processing and composite data types like a complex or a quaternion type as applied for DSP and 3D graphics. Extra complex, vector or quaternion operations may be implemented into the two to four way SIMD coprocessor, as well as conditional operations.

The CPU-on-FPGA used as the controller of the floating-point processor consumes about as many FPGA resources as the floating point coprocessor due to its pipelining and to its networking support. It has recently been implemented as a hardwired chip that can be used to control and FPGA-based coprocessor through its coprocessor interface. Ideally, our CPU or another controller CPU with similar capabilities should be implemented as a hardwired component on the FPGA in order not to use the configurable resources for such standard functions as communications and sequential control.

## References

[1] T.Ogita, S.M.Rump, S.Oichi, Accurate Sum and Dot Product, SIAM J. on Sci. Comp. 26(6), 2005

[2] www.xilinx.com, Spartan-III, Virtex-II pro, Virtex-IV families, µBlaze soft processor

[3] www.altera.com, Cyclone-II, Stratix-II families, NIOS II soft processor

[4] www.opencores.org, MIPS processor core, floating point processor by R.Usselmann

[5] F. Mayer-Lindenberg, Dedicated Digital Processors, Wiley&Sons 2004

[6] V.Beller, Floating-Point-Koprozessor für die CPU II, Diploma Thesis, TUHH 2005

[7] M.G.Arnold , 21st Century Slide Rules with Logarithmic Arithmetic, U. of Manchester, Institute of Science and Technology, U.K.

[8] A.Niyonkuru, H.C.Zeidler, Designing a runtime reconfigurable processor for general purpose applications, IPDPS'04, 2004

[9] Hennessy, Patterson, Computer Architecture, 3rd edition, Morgan Kaufman Pub.2003

[10] www.ti.com, TMS32C67xx DSP family

[11] D.E.Knuth,The Art of Computer Programming Seminumerical Algorithms,Addison-Wesley '98

[12] T.J.Dekker, A floating point technique for extentending the available precision, Num.Math.18,1971

[13] www.analog.com, Sharc DSP family

[14] Gelernter, Carriero, Coordination Languages and their Significance, CACM 32(2), 1992

[15] S.M.Rump, T.Ogita, S.Oichi,Accurate Floating Point Summation, Technical Report 05.12,TUHH

[16] Arenaire project (ENS of Lyon): FPLibrary – a Library of Operators for Real Arithmetic on FPGAs, available via the web

[17] D.A.Buell,, K.L.Pocek, Custom Computing Machines: An Introduction, Journal of Supercomp. 9.3, 1995

[18] J.F.Hart, Computer Approximations, R.Krieger Pub.Comp. 1978

[19] F. Mayer-Lindenberg, Design and Application of a scalable architecture for embedded systems, DSD 2006, Cavtat