PARAMETERISED FLOATING-POINT ARITHMETIC ON FPGAS

Allan Jaenicke

Vision Wizards Limited
The Surrey Technology Centre, 40 Occam Road
The Surrey Research Park, Guildford, Surrey UK

Wayne Luk

Department of Computing
Imperial College
180 Queen's Gate, London UK

ABSTRACT

This paper describes the parameterisation, implementation and evaluation of floating-point adders and multipliers for FPGAs. We have developed a method, based on the Handel-C language, for producing technology-independent pipelined designs that allow compile-time parameterisation of design precision and range, and optional inclusion of features such as overflow protection, gradual underflow and rounding modes of the IEEE floating-point format. The resulting designs, when implemented in a Xilinx XCV1000 device, achieve 28 MFLOPs with IEEE single precision floating-point numbers. These designs are used in an optimised implementation for computing Two-Dimensional Fast Hartley Transform. Preliminary results suggest that our implementation is faster than many programmable DSP processors and supercomputers.

1. INTRODUCTION

Floating-point operations are useful for computations involving large dynamic range, but they require significantly more resources than integer operations. The rapid advance in Field-Programmable Gate Array (FPGA) technology makes such devices increasingly attractive for implementing floating-point arithmetic. FPGAs offer reduced development time and costs compared to application-specific integrated circuits, and their flexibility enables field upgrade and adaptation of hardware to run-time conditions [10].

Early implementations either involved multiple FPGAs for implementing IEEE 754 single precision floating-point arithmetic [4], or they adopted custom data formats to enable a single-FPGA solution [9]. To overcome device size restriction, subsequent single-FPGA implementations of IEEE 754 Standard employed serial arithmetic [6] or avoided features, such as supporting gradual underflow, which are expensive to implement [5].

Table 1 compares various implementations of floating-point arithmetic units on FPGAs. Our implementation, undertaken as a final-year undergraduate project, achieves 28 MFLOPS and improved performance is expected from further hardware optimisations and advances in FPGA technology.

2. PARAMETERISATION

Our main aims of designing floating-point units (FPUs) for reconfigurable hardware implementation are: (a) to parameterise the precision and range of the floating-point format to allow optimising the FPUs for specific applications, and (b) to support optional inclusion of features such as gradual underflow and rounding. An approach meeting these aims will achieve effective design tradeoff between performance and required resources.

Our intention is to adhere to the structure and features of the IEEE 754 Standard to produce implementations compliant with this standard, while allowing the precision and the range of the format and inclusion of features to be parameterisable. When determining whether to pipeline the implementations, we conclude that it is necessary so as to support implementations in which high data throughput is more important than low latency.

To support custom arithmetic formats, it is necessary to allow the width of the exponent and fraction fields to be set at compile time. Allowing such parameterisation of the format considerably increases the complexity of the design. It is also necessary to ensure that the parameterisable design does not lead to significantly larger or slower designs.

The IEEE 754 Standard specifies a number of features which, depending on the data to be processed, may not be needed in the implementation. It is hence desirable to have the capability of selecting whether to include these features. This would often reduce the design size as well as improve the performance.

The following three features have been identified as optional. First, gradual underflow. Gradual underflow is especially costly to implement for the multiplier, as it requires a full-length shifter in the normalisation stage. Gradual underflow is not needed, if it is safe to regard values of magnitude smaller than $1.0 \times 2^{E_{MIN}}$ as 0, where E_{MIN} depends on the bias of the exponent. Second, overflow protection. Overflow is usually handled by setting the result to a specified bit pattern to signal that the result has the value infinity. Overflow protection is not needed if results are always within the range representable by the format used. Third, rounding modes. If rounding is not necessary, or if different rounding modes are available, this can be chosen at compile time.

Of the three features described above, gradual underflow and rounding modes are expected to have the largest impact on the logic requirements of the FPUs. When designing the FPUs, it becomes necessary to address the issue of pipeline integrity when different combinations of features can be included as proposed here. A modular approach is desirable to avoid design complexity. For instance, inclusion of optional features affecting one pipeline stage in the implementation should not require significant changes to be made to other pipeline stages.

We have developed methods for producing pipelined floating-point adders and multipliers with variable number of pipeline stages, such that designs with different time/space/feature tradeoffs can be implemented in hardware. The key difficulties in designing parameterisable pipeline units include: (a) allowing the number of pipeline stages to be varied while ensuring pipeline integrity, and (b) developing individual operations such that they will work with variable width of operands and optional support for algorithmic features, such as gradual underflow.

Table 1. FPGA Implementations of floating-point operations. The speed entries from [9] are based on 16-bit format, while those for this paper are based on 32-bit format. [5] predicts a performance of 33 MFLOPS for their multiplier on a Xilinx 40250XV FPGA.

	Shirazi [9]	Louca [6]	Ligon [5]	This paper
FPGA used	Xilinx 4010	Altera FLEX 81188	Xilinx 4020E	Xilinx XCV1000
Data format	Custom: 16/18 bit	IEEE single precision: 32 bit	IEEE single precision: 32 bit	Parameterisable: IEEE compliant
Addition	bit parallel, 3 stages	bit parallel, 3 stages	bit parallel, 13 stages	bit parallel, 8 stages min.
	9.3 MFLOPS	7 MFLOPS	40 MFLOPS	28 MFLOPS
Multiplication	bit parallel, 3 stages	digit serial, 6 stages	Booth, 3 stages	bit parallel, 5 stages min.
	6 MFLOPS	2.3 MFLOPS	5.5 MFLOPS	28 MFLOPS

To illustrate our method, we explain below the design of a pipelined floating-point multiplier such that the number of pipeline stages is determined by a user-provided parameter N.

Stage 1: extracts signs, exponents and fractions from the operand and makes the implicit MSB explicit. If gradual underflow is supported, a check for whether the operands are denormals is necessary. The outcome will affect the value of the MSB and the exponent, setting both to 1 if a denormal value is detected.

Stage 2 to N ($N \geq 2$): computes the product of the mantissas.

Stage $(N+1)$: evaluates the distance and direction to shift the product; only required if gradual underflow is supported.

Stage $(N+2)$: checks if the normalisation shift, determined to be necessary in the previous stage, is indeed allowed, so that the exponent will not underflow if the mantissa is left shifted the required amount; only required if gradual underflow is supported.

Stage $(N+3)$: provides the normalisation shift, and if rounding is required then the round and sticky bits are updated to reflect the effect of the normalisation shift; only required if gradual underflow is supported.

Stage $(N+1/4)$: performs rounding; only required if rounding is supported.

Stage $(N+1/4/5)$: writes the result to the destination register. If gradual underflow is not supported, then a right-shift must be performed if there is a carry-out from the multiplier.

A floating-point multiplier based on the above pipeline stage division will have at least 3 stages, when the multiplication takes a single cycle and gradual underflow and rounding are not included.

3. IMPLEMENTATION

The designs shown in Table 1, except ours, were produced using the VHDL language. We have considered three languages for developing our parameterised FPUs: VHDL, Pebble [7] and Handel-C [2],[10]. VHDL and Pebble support structural hardware description, while Handel-C captures designs similar in style to the C language. Handel-C is chosen because: (a) we wish to focus on algorithmic level parameterisation and optimisation, and a software-style language appears best for this purpose; (b) the Handel-C compiler contains interface libraries for the RC1000-PP FPGA system [2] which simplified our implementation.

The FPUs implemented have the characteristics shown in Table 1. The implementation addresses all the issues discussed in the preceding section. They can be fully pipelined and can support both custom formats and the standard IEEE formats.

The arithmetic operations in the FPUs have been implemented using the standard Handel-C adders and multipliers. Our floating-point adder is implemented as a multi-stage addition, with the width of adders and hence the number of cycles involved as parameters. Our multiplier includes a parameterisable multiplier-adder tree for computing the product of the mantissas. We adopt this approach mainly for performance [9], and also it would have been cumbersome to implement, for instance a pipelined multiplier array, in a non-structural language like Handel-C.

A comparison of cost and performance for different parameterisations has been conducted to gain an insight into: (i) the cost of the different features that can be included optionally, and (ii) their effect on performance. For this purpose the following configurations for each FPU are considered. (a) Basic configuration: denormals are set to zero. (b) Overflow configuration: the Basic configuration plus overflow protection. (c) Gradual Underflow: the Basic configuration plus gradual underflow. (d) Round: the Basic configuration plus IEEE round to nearest rounding. (e) Complete: the Basic configuration plus overflow protection, gradual underflow and IEEE round to nearest rounding.

Different configurations of the arithmetic operation of each FPU have also been considered, using the Basic configuration and the IEEE single precision format. Implementation costs are given by: (a) device independent figures from the Handel-C compiler, and (b) device specific figures from the Xilinx tools which produce an FPGA configuration file from the output of the Handel-C compiler. The Handel-C compiler provides two useful figures for resource usage: the number of gates and latches. The Xilinx tools provide the number of 4-input lookup tables (LUTs) and flip-flops (Figure 1) as well as the number of Slices (Figure 2) used by a design in a Xilinx XCV1000 FPGA. Standard settings have been used in both the Handel-C compiler and the Xilinx tools.

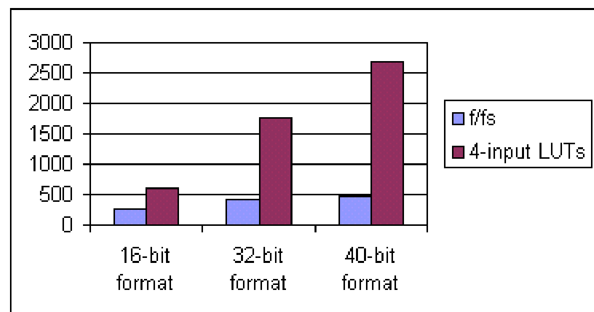

Fig. 1. The cost in number of LUTs and flip-flops for floating-point multiplication using the Basic configuration, from Xilinx tools.

The cost of implementing a floating-point multiplier with different floating-point bit-size formats is shown in Figure 1. The 16-bit format uses a 4-bit exponent and an 11-bit fraction, while the 32-bit format is equivalent to the IEEE single precision format, namely 8-bit exponent and 23-bit fraction. The 40-bit format uses a 10-bit exponent and a 29-bit fraction.

The costs of different optional features for floating-point multipliers are shown in Figure 2. There is significant cost for adding gradual underflow, since a shifter is required for normalising the product. The normalisation units contribute significantly to the cost of the adder which uses a barrel-shifter, and a cheaper implementation should be sought. Moreover, inclusion of proper rounding adds 10–15% to the cost, which is similar to the cost of adding rounding in the adder implementation.

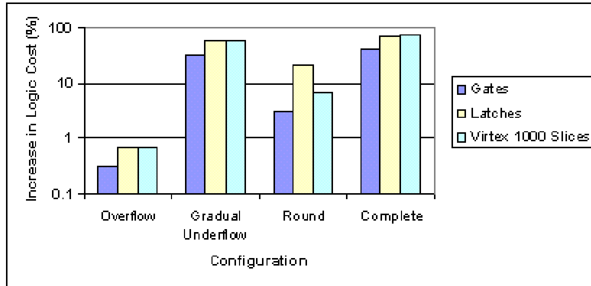

Fig. 2. Floating-point multiplier implementation: the cost of support for overflow is negligible, while support for gradual underflow nearly doubles its size. Note that a logarithmic scale is used.

As with adders, different multiplier configurations have been observed to have similar longest delay paths. Various parameterisations of the arithmetic operation have been tried. We divide each input into k partitions and use a multiplier-adder tree with k^2 multipliers and $2^k - 1$ adders, where each multiplier has two n/k-bit inputs and one $2n/k$-bit output. The implementation cost rises rapidly with k, however: sixty-four 3-bit multipliers are about 75% more costly than four 12-bit multipliers. This increase in cost is partly due to an increase in the number of latches needed as the number of pipeline stages increased from 5 to 9.

Our current implementations do not involve FPGA-specific optimisations, making them portable across different devices. Such optimisations, while likely to be tedious and error-prone, should improve performance and device utilisation. For instance, it has been reported that 4-2 adders and delayed addition techniques work well for FPGAs, achieving a clock speed of 97 MHz on a Xilinx XCV100E device for 32-bit floating-point accumulation with overflow detection and handling [8].

4. CASE STUDY: 2D FAST HARTLEY TRANSFORM

This section describes the use of our floating-point building blocks for implementing the Fast Hartley Transform (FHT) on a XCV1000 FPGA in an RC1000-PP system [2]. The FHT can be considered an optimisation of the FFT when only real values are required [3]. Our FHT processor achieves a significant reduction in the number of clock cycles by accelerating the updating of the control variables, and by fusing the transpose, permutation and the first stage of the FHT. Its performance is compared to various dedicated and programmable FFT processors, and the results are encouraging.

The target for our implementation is a Xilinx XCV1000 FPGA in an RC1000-PP system, containing four banks of 2MB SRAM. Due to the cost of setting up DMA channels between the FPGA board and the host PC, it has been decided that the entire FHT would be implemented in hardware. This means that to process the data set, the FPGA would need to have access to all four memory banks while computing the FHT, not allowing the interleaving of computation and data transfer.

The first stage is split from the subsequent stages as it only uses the add/subtract unit. Due to memory access constraints, we are limited to reading one operand from each memory bank per cycle; it would be inefficient to process a single row of data at a time with the FPUs. Our solution is to interleave the processing of two rows from two memory banks, reading a pair of operands and writing a pair of results every cycle.

An elegant solution has been found which allows us to fuse the transpose, permutation and the first stage of the FHT. We have observed that a special transpose operation is not required, since the same effect can be obtained by addressing the rows as columns and vice versa in the memory space. Examination of the permutation table reveals that the first stage, which always computes the addition and subtraction of two adjacent pairs of operands, does not have to be performed interleaved if combined with both the transpose and the permutation. The permutation function always replaces the pairs of operands needed by the add/subtract operation in the first stage with one operand from the lower half of the row, and the other operand from the upper half. By combining the first stage with both the transpose and the permutation operation, these accesses to the two halves of a row become accesses to two different halves of a column, involving two memory banks.

Another improvement has been achieved by iterating over all the rows for each state of the control variables rather than computing the FHT of two rows at a time. This optimisation reduces the number of cycles spent updating control variables, halving the number of cycles required to compute the FHT.

A further optimisation concerning memory access has been applied. We have observed that the inner most loop contains two operations which use an identical pair of operands. One of these operations is conditional upon two control variables being unequal. During the iterations where both are executed, which is the majority, the two operations could be interleaved such that the operands are not read from memory twice. This enables us to read a third operand required by the operation.

We employ a Double Loop approach instead of using shift registers to delay control data specifying the destination address of results of computations carried out by the FPUs. This approach involves two similar implementations of the same loop construct, one starting execution delayed by the number of cycles given by the latency of the FPU, from which it reads the results. One of the loops then issues operands to the FPU, using the state of its control variables to index RAM reads, and the other receives the results from the FPU and writes them into two RAMs using its local control variables to index RAM writes.

Our FHT implementation provides a case study for analysing the suitability of FPGAs for floating-point arithmetic. The cost of the implementation varies with the size of the data set, as the width of the control variables changes and different amounts of on-chip RAM is needed to store the trigonometric and permutation tables.

We have found that the implementations scale well in terms of logic and flip-flop requirements. For the IEEE single precision

format with 32-bit data, the FHT compiled for 1024^2 element data sets uses around 59% of Slices in a Xilinx XCV1000. The on-chip RAM requirements are affected by the size of the data set, as they are used to store the trigonometric and permutation tables, An M^2-element data set will require two $M/4$ word P-bit of RAMs to store the trigonometric tables, where P is the width of the floating-point format used, and one M word $\log M$-bit RAM for the permutation table. As there are 131,072 Block RAM bits in the Xilinx XCV1000, FHT implementations with up to 4096^2 elements requiring 115 Kbits can be accommodated in the chip.

As expected, the number of cycles that our processor takes to compute the 2D FHT is roughly proportional to $O(N \log N)$, the number of operations required to perform the calculation. In calculating the 2D FHT of small data sets, more cycles are lost relative to the size of the data set than in the larger data sets. This is due to the implementation waiting for the FPU pipeline to empty before moving on to the next state of control variables.

Our design is scalable, since more rows can be processed in parallel. Computation time can be reduced by 45–50% by having eight 1MB RAMs rather than four 2MB RAMs, and instantiating another set of FPUs. Such systematic halving can be continued as long as sufficient logic is available for the additional FPUs.

The current implementation can be clocked at around 22 MHz, producing the 2D FHT of a 1024^2-element data set in around 0.52 second. Table 2 compares it against other systems from [1] and a lab PC. Our 2D design performs 2048 1D FHT transforms of a 1024-point data set and a data transpose in 520 ms, so the figure of 254 μs (=520ms/2048) is an overestimate of the time for each 1K-point transform. However, the other systems (except the Pentium) compute the FFT, which involves twice as many computations as the FHT. Dedicated FFT processors, however, may not be able to take advantage of the reduced computations in the FHT unless designed to do so. We have not included the FHT processor in [3] since it is based on fixed-point arithmetic with block-floating-point scaling.

We observe the following from Table 2. (a) Our FHT processor has the lowest clock speed, while it is faster than most programmable DSP and supercomputer implementations. Moreover, unlike some FPGA implementations [5], our design is compliant with IEEE 754 format. (b) Handel-C has proved useful particularly for algorithmic level optimisations. The combination of Handel-C and the RC-1000PP system provides a powerful vehicle for rapid prototyping hardware designs; it enables, for instance, final-year undergraduate projects such as this project to involve complex circuit implementations. (c) There is much scope for improving our design, such as using a faster FPGA, including device-specific optimisations [8], and having multiple FHT processors and custom external interfaces on the same chip if desired.

Table 2. Performance comparison of our FHT processor with other systems in [1], which compute the FFT. The first two are dedicated FFT devices, while the rest are programmable DSP processors or supercomputers. The Pentium, in a PC, runs 2D FHT.

Processor	Time for 1K-point transform (μs)	Clock Speed (MHz)
DoubleBW powerFFT	10	128
Texas Mem Sys TM-66	65	50
Our FHT Processor	254	22
Sharc ADSP-21061	460	40
Pentium-III	469	800
Cray Y-MP (1-CPU)	600	159
Cray 2 (1-CPU)	1000	244
TMS 320C40	1298	60
Lucent DSP16000	2110	80

5. CONCLUSION

We have presented an approach for developing parameterised FPUs for hardware implementation. Our designs can be used as building blocks for floating-point applications customised to meet user constraints, for instance by varying the precision, rounding modes, or the number of pipeline stages. Current and future work includes optimising our hardware implementation to exploit FPGA-specific features, and developing tools which can automatically produce designs that meet given numerical characteristics as well as performance, size and power consumption requirements.

Acknowledgements. Many thanks to George Constantinides, Roger Gook, Karel Hruda, Philip McLauchlan, Oskar Mencer, Nabeel Shirazi and Tim Todman for their comments. The support of Celoxica Limited, UK Engineering and Physical Sciences Research Council (Grant number GR/54356) and Xilinx, Inc. is gratefully acknowledged.

6. REFERENCES

[1] Baas, B.M. *FFT Chip Comparisons*, http://www-star.stanford.edu/∼bbaas/fftinfo.html.

[2] Celoxica Limited, http://www.celoxica.com.

[3] Erickson, A.C. and Fagin B.S. "Calculating the FHT in hardware", IEEE *Trans. on Sig. Proc.*, 1992, pp. 1341–1353.

[4] Fagin, B. and Renard, C. "FPGAs and floating point arithmetic", *IEEE Trans. on VLSI*, 1994, pp. 365–367.

[5] Ligon III, W.B. et al. "A re-evaluation of the practicality of floating-point operations on FPGAs", *Proc. IEEE Symp. on FPGAs for Custom Comput. Machines*, 1998, pp. 206–215.

[6] Louca, L. et al. "Implementation of IEEE single precision floating-point addition and multiplication on FPGAs", *Proc. IEEE Symp. on FPGAs for Custom Comput. Machines*, 1996, pp. 107–116.

[7] Luk, W. and McKeever, S. "Pebble: a language for parametrised and reconfigurable hardware design", *Field-Programmable Logic and Applications*, LNCS 1482, Springer, 1998, pp. 9–18.

[8] Luo, Z. and Martonosi, M. "Accelerating pipelined integer and floating-point accumulations in configurable hardware with delayed addition techniques", *IEEE Trans. on Comput.*, 2000, pp. 208–218.

[9] Shirazi, N., Walters, A. and Athanas, P. "Quantitative analysis of floating-point arithmetic on FPGA based custom computing machines", *Proc. IEEE Symp. on FPGAs for Custom Comput. Machines*, 1995, pp. 155–162.

[10] Styles, H. and Luk, W. "Customising graphics applications: techniques and programming interface", *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, 2000, pp. 77–87.