Analysis of High-performance Floating-point Arithmetic on FPGAs*

Gokul Govindu, Ling Zhuo, Seonil Choi and Viktor Prasanna
Dept. of Electrical Engineering
University of Southern California
Los Angeles, CA 90089 USA
{govindu, lzhuo, seonilch, prasanna}@usc.edu

Abstract

FPGAs are increasingly being used in the high performance and scientific computing community to implement floating-point based hardware accelerators. In this paper we analyze the floating-point multiplier and adder/subtractor units by considering the number of pipeline stages of the units as a parameter and use throughput/area as the metric. We achieve throughput rates of more than 240Mhz (200Mhz) for single (double) precision operations by deeply pipelining the units. To illustrate the impact of the floating-point units on a kernel, we implement a matrix multiplication kernel based on our floating-point units and show that a state-of-the-art FPGA device is capable of achieving about 15GFLOPS (8GFLOPS) for the single (double) precision floating-point based matrix multiplication. We also show that FPGAs are capable of achieving upto 6x improvement (for single precision) in terms of the GFLOPS/W (performance per unit power) metric over that of general purpose processors. We then discuss the impact of floating-point units on the design of an energy efficient architecture for the matrix multiply kernel.

1 Introduction

The recent FPGA devices have multi-million gate logic fabrics capable of achieving frequencies upto 300MHz, reasonably large on-chip memory and fast I/O resources (for off-chip communication to either processors or memory). Platform FPGAs also include embedded processors for either processing or general housekeeping. Most signal processing, linear algebra and scientific algorithms use kernels using matrix and vector operations and are suitable for pipelined implementations. Pipelined architectures for such kernels, can use deeply pipelined floating-point units to achieve an order of performance more than general purpose floating-point processors.

*This work is supported by the National Science Foundation under award No. CCR-0311823 and in part by an equipment grant from HP.

For most of the previous implementations of the floating-point units, the precision of the mantissa and exponent was usually assumed to be the critical parameter. The floating-point units were optimized for either a high frequency or a low slice count after considering a given bitwidth as a parameter for design trade-offs. Also most of the available floating point units were assumed to be stand-alone units and weren't considered with the encompassing kernel in mind. We use the throughput achieved per unit area metric, since optimizing the floating-point units in terms of only area or only frequency at the cost of the other, might reduce the performance per device for the overall kernel.

The computational power of FPGAs is finding acceptance in high performance embedded computing where power is an issue. Though platform FPGAs are not low-power, they compare favorably with floating-point processors. We use the performance obtained(in GFLOPS) per unit power(in Watts) consumed as a metric metric to judge both performance and power. Energy efficiency of fixed-point architectures is obtained by considering various tradeoffs between architectural parameters like the parallelism, number of compute elements and storage elements, throughput/latency, etc. Floating-point units themselves can be parameterized with respect to depth of pipelining, throughput/latency, area, etc. Hence, the impact of the pipelined floating-point units on the overall architecture and the architectural parameters have to be considered together to obtain energy efficient floating-point based architectures.

In this paper, we analyze the area, latency and achievable throughput of a floating-point adder/subtractor and multiplier by considering the pipelining of the units as a parameter. Pipelining can exploit the unused flipflops present in the slices of the recent FPGA architectures like the Virtex2Pro and cause only a moderate increase in area. However, we see that pipelining beyond a certain point achieves diminishing returns. For this, we present a frequency/area vs. pipelining stages analysis. We then analyze the performance of the floating-point units in the context of the matrix multiplication kernel as it is widely used in high performance and scientific computing. Designs for matrix multiplication for large sized matrices typically occupy the whole device and contain many floating-point units. Hence we analyze the performance of the complete device along with that of the floating-point units.

In this paper, we have used optimizations for and resources of the Xilinx Virtex2Pro architecture. Most of the recent FPGAs have a similar architecture. The optimizations we did were limited to pipelining the subunits and using the options provided by the synthesis and place and route tools. We did not manually place and route the design. Thus, our designs are not specific to any single device and also the designs can be manually optimized further.

The rest of the paper is organized as follows. In Section 2, we describe the applications using floating-point units, related work and the floating-point format used. In Section 3, we describe our floating-point units. In Section 4, we present the quantitative details of the floating-point units. We also present the performance of a matrix-multiply kernel with our floating point units. In Section 5 we analyze the impact of floating-point units on the energy efficiency of the matrix multiply kernel. Section 6 draws conclusions.

2 Floating-point Operations on FPGAs

2.1 Applications

High performance and scientific computing applications are typically compute-intensive and require high throughput rates. Examples of such applications are radar/sonar signal processing, image processing, molecular analysis, etc. These applications demand high numerical stability and accuracy and hence are usually floating-point based. FPGAs are increasingly being used for such applications when processor implementations do not satisfy the real-time and high performance requirements. Moreover, since embedded signal processing systems operate under low-power constraints, FPGA implementations can be preferred over processors. These applications typically involve kernels like matrix and vector operations. In order to utilize deeply pipelined, throughput optimized floating-point units, the kernels should essentially have a simple computational pattern and the logic surrounding the floating-point units should also be able to achieve high throughput rates. These kernels can be built using a linear array of processing elements, with each processing element having one or more floating-point units. In the case of large sized matrices or vectors, data dependencies occur after long and definite intervals. These intervals can be determined during the architectural design and a designer can hide the latency of the deeply-pipelined floating-point units. For example, in the matrix multiplication algorithm [5] that we use, there will be read-after-write hazards only if the matrix size is less than the number of pipeline stages. Such architectures need to have throughput rather than latency optimized floating-point units. However, since such architectures might use a large number of units, the throughput obtained for the units shouldn't be too area prohibitive for the whole kernel. Hence, we use throughput/area as a metric for the floating-point units.

2.2 Related Work

Much work in the past has been done on designing floating-point units on FPGAs [9]. Some of the work has concentrated on optimizing the large bitwidth fixed-point adder/multipliers, shifters, priority encoders etc. In implementations on previous devices, these took up a lot of slices and were slow. However designing floating-point units nowadays has become much easier with resources like adders or priority encoders using fast carry chains, fast embedded multipliers, etc available. Moreover pipelining can utilize the large number of flipflops already present in the fabric. Also, at the synthesis tool level, area/speed optimizations, forced synthesis of subunits etc also result in good designs. IP cores like parameterised fixed-point multipliers and adder/subtractors can also be incorporated into the floating-point units. Hence the focus is shifting from designing the floating-point units to optimally utilizing the available subunits.

Some of the commercial [7] and open source floating-point cores [1] assume the precision to be the critical parameter. However, the number of pipelining stages also has to be considered as a parameter to get optimal throughput rates. Some of the commercial floating-point cores use a custom format with conversion to and from the IEEE754 standard at interfaces to other resources in the system. Various tools [6] have been built to automate the optimization and generation of the floating-point units.

3 Description of our Floating-point Cores

We describe our floating-point adder/subtractor and the floating-point multiplier in this section. In this paper, though we have followed the IEEE754 format for the single and double precision, we haven't provided for denormal or NaN numbers. Denormal and NaN numbers are generally considered rare and may not justify the usage of a lot of hardware required for their handling. We also implemented only the rounding-to-nearest and truncation options. We used the standard three stage algorithm consisting of denormalization/preshifting, mantissa addition/subtraction

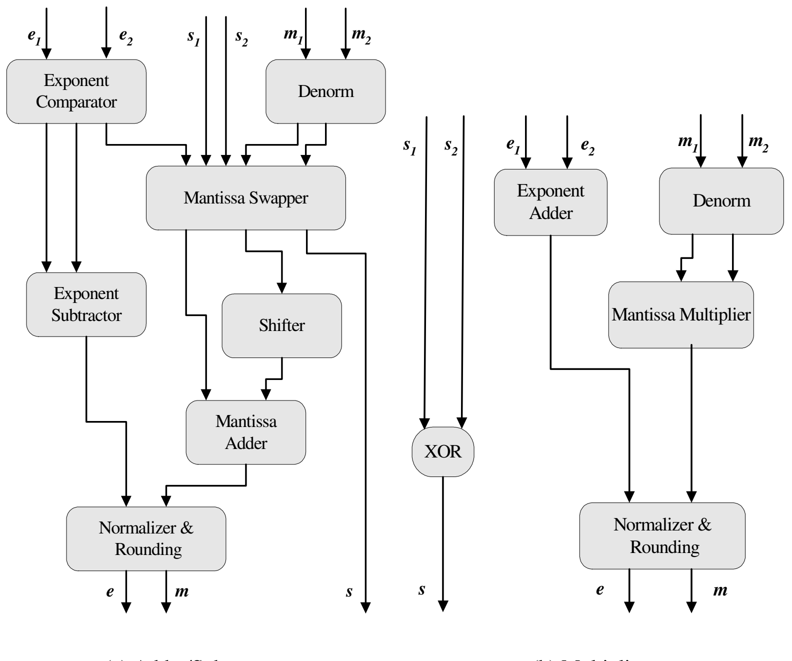


Figure 1. Block Diagrams for Floating Point Cores (m_1, m_2 are the mantissas of the operands, e_1, e_2 are the exponents, and s_1, s_2 are the sign-bits; m, e, s are the mantissa, exponent and sign-bit of the result respectively)

and normalization/rounding. Figure 1 shows the block diagrams of the well known[4] floating-point algorithms for addition/subtraction and multiplication that we use.

The first stage of the floating-point adder/subtractor is the denormalization/preshifting module. The denormalization/preshifting module is made up of:

1. A denormalizer to make the hidden bit explicit. It uses a comparator for checking whether the exponent is zero. The outputs of the denormalizer are registered. Comparators of a bitwidth less than or equal to 11 can achieve 250MHz. Comparators take about $n/2$ slices for a bitwidth of n.
2. A swapper to swap the mantissas after comparing the exponents. A comparator and a multiplexer are used and a pipeline stage can be inserted between the comparator and multiplexer. The mantissa comparator for double precision can achieve a frequency of 220MHz and requires pipelining for higher frequencies. The outputs of the swapper are registered.
3. A shifter to align the mantissas depending upon the difference in exponents. The shifting operation is done by using large muxes (takes up about $nlogn/2$ slices for a bitwidth of n). Muxes implemented using tristate buffers are faster, but the tristate buffers are limited. Synthesis options for forcing MUX synthesis with MUXCY, MUXF attributes can be also used. Typically the shifting operation has to be pipelined to achieve a better frequency. Three muxes in serial can be considered as a stage and a frequency of more than 200Mhz can be achieved by doing so. Higher frequencies require two-mux stages.

The second stage is the fixed-point adder/subtractor. It is made up of

1. A fixed-point mantissa adder/subtractor which can use the Xilinx library cores which have the number of pipeline stages as a parameter. For example a 54bit adder/subtractor can achieve 200MHz with 4 pipelining stages. It takes about $n/2$ slices for a bitwidth of n excluding pipelining.
2. A pre-normalizer module which shifts the sum by one if there was a carryout and adds a one to the exponent. A pipeline stage can be inserted between the adder and 1bit shifter. The outputs are registered.

The third stage is a normalizer/rounding module. It is made up of

1. A normalizer consisting of a priority encoder/shifter and an exponent subtractor to bring the first one in the

mantissa to the MSB. The priority encoder is a critical subunit for large bitwidths and its synthesis by the tool has to be forced. For 54bits it has to be broken into two smaller priority encoders and a 3bit adder and related muxes, to achieve a frequency greater than 200MHz. The shifter also can be pipelined as described above for the shifter in the first stage. A pipeline stage between the priority encoder and the shifter can be used. The outputs of the shifter are pipelined.

2. A rounding module which essentially is made up of constant adders for the mantissa and the exponent. The constant adders can be from the library cores. Pipeline stages can be inserted between the adders. The outputs of the rounding module are registered.

Floating point multiplication is easier than addition/subtraction to implement. The first stage of the floating-point multiplier is the same Denormalization module used in addition to insert the implied 1 to the mantissas of the operands.

The second stage is made up of

1. A fixed-point mantissa multiplier. We use the Xilinx library cores which have the number of pipeline stages as a parameter. Typically, for the 54bit fixed-point multiplication, seven pipelining stages are required to achieve a frequency of 200MHz. Also, speed optimizations for the synthesis tool might result in more embedded multipliers being used up. The outputs of the module are registered.
2. A fixed-point adder and subtractor to add the exponents and subtract the bias from the sum. A pipeline stage can be inserted between the adder and subtractor to achieve a greater frequency.

The third stage is a Normalizer/Rounding module and is made up of

1. The normalizer which is simpler than the one used for the adder. It is made up two bit shifter and a subtractor. Since we do not consider denormal numbers, we shift the mantissa of the result atmost by two bits. The exponent is adjusted accordingly with each shifting. Pipeline stages can be inserted between the shifter and the exponent subtractor. The outputs of the normalizer are registered.
2. The rounding module which is the same as used for the adder.

For both multiplier and adder, registers are inserted between or inside the modules when pipelining is needed. At every stage exceptions are detected and carried forward into the next stage. An output signal "DONE" is also used to indicate that the operation of the module is completed.

For each floating point unit, we can begin with an implementation with least pipeline stages. After synthesize, place & route, we identify the critical path of the implementation. A new pipeline stage is then inserted to break down the critical path to decrease clock cycle and improve throughput. We repeat this process until diminishing returns occur, that is, new pipeline stage yields no improvements in throughput. Note that using a different optimization objective (speed or area) for the synthesis and place and route tool gives vastly different results. Thus the throughput/area metric should be obtained for all implementations with different pipelining stages and also for different optimization objectives for the tools. Also, though extensive pipelining increases achievable clock frequency, it also requires a lot of area for the registers between the pipeline stages. For the floating point adders, the pipelining to split the adder/multiplier, the priority encoder and the shift registers for the normalizing unit, shows an immediate improvement in frequency, without much increase in area. Further pipelining, however, shows diminishing returns in frequency and the area increases significantly. As for the floating point multipliers, the pipelining inside the mantissa multiplier contributes most in throughput improvement.

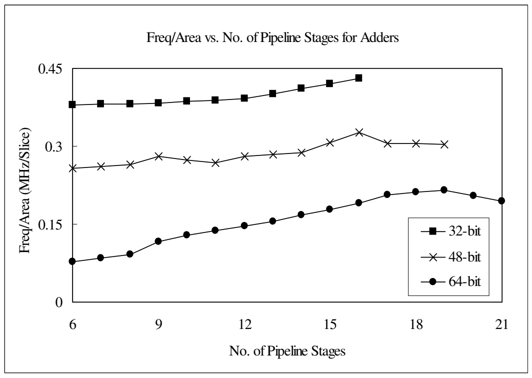
4 Experimental Results and Analysis

We implemented our floating-point unit designs using VHDL. These were synthesized and placed and routed using the Xilinx ISE5.2i, on a Virtex2Pro XC2VP125-7f1696 device [10].

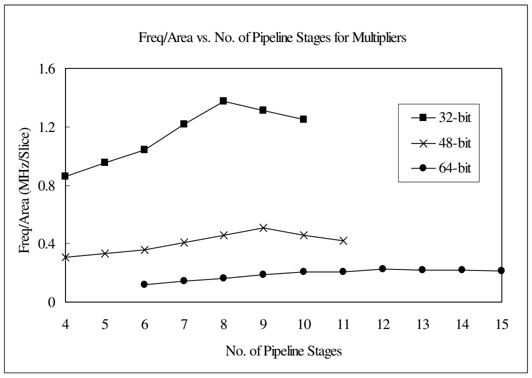
4.1 Performance of the Floating Point Cores

Figure 2 shows the variation of the frequency/area metric with the number of pipelining stages. We see that for both the adder/subtractor and the multiplier, the curves flatten out towards the end and may dip for deep pipelining. This shows that pipelining beyond a certain limit gives diminishing returns.

Table 1 shows the areas and throughputs obtained for 32,48 and 64 bit precisions floating-point addition/subtraction with minimal, maximal, and optimal implementation. By "optimal", we mean that the implementation reaches highest freq/area ratio. Table 2 provides the corresponding values for floating point multipliers. Note that optimization objectives (area or speed) for the synthesis and place and route tool result in vastly different results. A speed optimization objective for the synthesis tool might result in logic replication to reduce logic levels and will result in larger area. Similarly speed optimization objective for the place and route tool will result in more slices being used only for routing resources and will result in a greater area. Similarly, tight timing constraints without efficient pipelining can result in area overhead, which can otherwise be saved by analyzing and pipelining the critical path.



(a) Adder/Subtractor



(b) Multiplier

Figure 2. Analysis of Frequency/Area vs. Number of pipeline stages in the floating point cores

Table 1. Analysis of 32, 48, 64-bit Floating Point Adders

Precision	32-bit			48-bit			64-bit		
	min	max	opt	min	max	opt	min	max	opt
No. of Pipeline Stages	6	19	16	6	19	14	6	21	19
Area (slices)	390	551	520	504	820	703	633	1133	933
LUTs	549	548	548	674	780	706	1020	1049	1032
Flip Flops	391	801	520	479	1126	950	443	1543	1148
Clock Rate (MHz)	150	250	230	130	250	230	50	220	200
Freq/Area (MHz/slice)	0.38	0.45	0.44	0.257	0.304	0.327	0.078	0.194	0.216

Table 3 compares our 32-bit floating-point cores and the cores obtained from Nallatech [7] and Quixilica [8]. Note that the Nallatech and Quixilica cores use custom formats and require additional modules to perform format conversions at interfaces to other resources in the system. Hence, due to a lower area, their Frequency/Area metric is sometimes better than ours. As there are few 64-bit floating point cores available, we compared our 64-bit cores with the designs generated using the parameterized library of [1]. The comparison results are shown in Table 4.

We also show the power consumption of different designs at 100MHz. The power values were obtained using Xilinx XPower [10]. Figure 3 shows the variation of the power with the number of pipelining stages. These power values include only the clocks, signal and logic power. Inputs, outputs and quiescent power which have to be counted for a design on the complete device, are not counted.

4.2 Floating-point based Matrix Multiplication

In this paper, we use matrix multiplication as an example to illustrate the tradeoffs in selecting the floating-point units. Our architecture for matrix multiplication is a linear array of identical PEs (Processing Elements), each of which contains a floating-point adder and a floating-point multiplier. For a detailed description of the matrix multiplication architecture and algorithm, refer to [5]. Since the number of slices on a given device is constant, the device will accommodate fewer PEs if deeply pipelined units occupying a large area are used. Hence, the performance of the device might be lower even if the frequency of the units is high. Also, the overall architectures operating frequency should be considered while choosing the floating-point units. Since our matrix multiplication architecture is capable of achieving 250MHz for single precision, we use the optimal implementations discussed in 4.1. If the overall architectures operating frequency is less than the optimal frequency for the floating-point unit then floating-point units with the best frequency/area metric considering a lower frequency have to be chosen.

Using our designs, a Xilinx Virtex-II Pro XC2V125 device is able to achieve 19.6 GFLOPS for 32-bit matrix multiplication. This is a 6X improvement over the 2.54 GHz Pentium 4 processor, and a 3X improvement over the 1.25

Table 2. Analysis of 32, 48, 64-bit Floating Point Multipliers

Precision	32-bit			48-bit			64-bit		
	min	max	opt	min	max	opt	min	max	opt
No. of Pipeline Stages	4	9	7	4	11	9	6	15	12
Area (slices)	116	220	201	227	508	423	491	1019	910
LUTs	186	203	196	382	448	383	750	777	764
Flip Flops	65	324	249	91	795	647	320	1749	1546
Clock Rate (MHz)	100	250	240	70	215	215	60	215	205
Freq/Area (MHz/slice)	0.862	1.13	1.24	0.308	0.423	0.508	0.122	0.211	0.225

Table 3. Comparison of 32-bit Floating Point Units

	32-bit Adder			32-bit Multiplier		
	USC	Nallatech	Quixilica	USC	Nallatech	Quixilica
No. of Pipelines	19	14	11	8	6	6
Area (slices)	551	290	306	182	126	326
Clock Rate (MHz)	250	184	169	250	188	152
Freq/Area (MHz/slice)	0.45	0.63	0.55	1.37	1.49	0.467

GHz G4 processor [3].

5 Impact on Design of the Matrix Multiplication Architecture

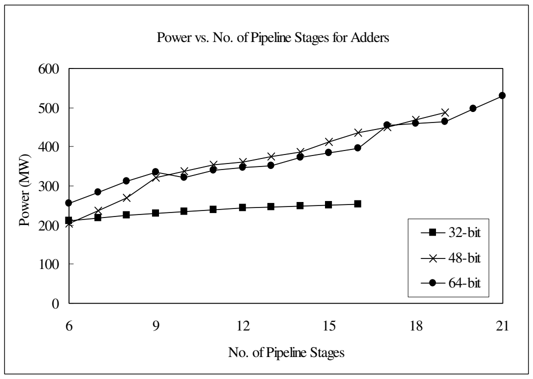
In this section, we show the impact of the floating-point units on architectural design. To quickly estimate the performance of the various designs and also perform high level design tradeoffs, we have employed domain-specific modeling proposed in [2]. Domain-specific modeling is a hybrid (top-down plus bottom-up) approach to performance modeling that allows the designer to rapidly evaluate candidate algorithms and architectures in order to determine the design that best meets criteria such as energy, latency, and area. Initially, the architecture is split into it's individual components. The components for the matrix multiply architecture are the floating-point units, storage elements like registers, slice and block-based memory and other elements like multiplexers, counters, etc. From the algorithm, we know when and for how long each component is active and its switching activity. Additionally, with estimates for the power dissipated by each component, we can estimate the energy dissipated by the design. The error between the results from actual implementation and the results from the domain-specific modeling approach is usually less than 10% [2].

For floating-point based matrix multiplication, in order to satisfy data dependencies due to the long latencies of the units, the problem size should be greater than the sum of the adder and the multiplier latencies. Similarly, the control signals also have to be shifted using shift registers so that the correct schedule of operations is maintained. For smaller problem sizes, zero padding has to be used, to satisfy the above latency constraint. This zero padding constitutes wasteful energy dissipation and has to be minimized. In [5], block matrix multiplication was employed for matrices with large problem sizes. Block size b was used as a parameter while performing design tradeoffs. In the floating-point architecture, for small block sizes, zero padding has to be used to satisfy the latency requirement. With energy-efficiency as the objective, the block size b can be used as a parameter.

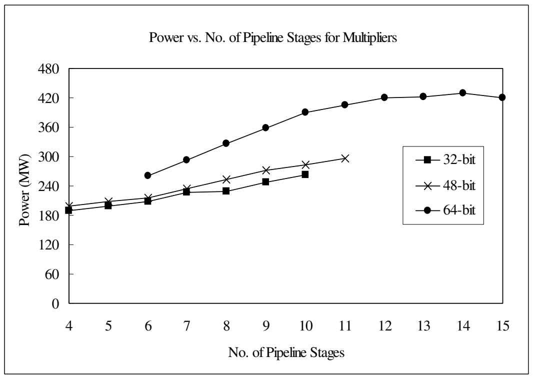
Figure 4 shows the energy distribution in a PE for problem sizes of $n = 10$ and $n = 30$ when minimum, moderate and maximum pipelined floating-point units are used. We see that for the smaller problem size using deeply pipelined floating-point units result in lot of energy wastage due to zero padding. Figure 5 shows the energy, resources and latencies for various problem sizes when minimum, moderate and maximum pipelined floating-point units are used. PL implies pipeline stages and is the sum of the latencies of the multiplier and adder. We see that even though the deeply pipelined architecture consumes a lot of area, it might consume the least energy due to less latency. Thus based upon the area, latency and energy constraints, architectural choices can be made from Figure 5. Figure 6 shows the energy, latency and resources for various block sizes when minimum, moderate and maximum pipelined floating-point units are used for a problem size $n = 16$. We see that there is large amount of wasteful energy dissipation when the block size is much smaller than the latency of the floating-point units.

Table 4. Comparison of 64-bit Floating Point Units

	64-bit Adder		64-bit Multiplier	
	USC	NEU	USC	NEU
No. of Pipeline Stages	19	8	12	5
Area (slices)	933	770	910	477
Clock Rate(MHz)	200	54	205	90
Freq/Area (MHz/slice)	0.216	0.07	0.225	0.188



(a) Adder/Subtractor



(b) Multiplier

Figure 3. Analysis of Power vs. Number of pipeline stages in the floating point cores

6 Concluding Remarks

In this paper, we showed that FPGA based floating-point architectures like the matrix multiplication employing efficient floating-point units can achieve a significant improvement in performance over that of processors. Moreover, performance per unit power expended, for designs on FPGAs is lower than that for designs on processors. We also showed the impact of the floating-point units on the overall design. And since, the floating-point units can be resource/latency/energy dominant than the other resources like storage, control, etc, they have to be analyzed in the context of the overall architecture.

References

[1] P. Belanovic and M. Lesser. A Library of Parameterized Floating Point Modules and Their Use. In *Proc. of 12th International Conference on Field Programmable Logic and Application*, Montpellier, France, September 2002.

[2] S. Choi, J. Jang, S. Mohanty, and V. K. Prasanna. Domain-Specific Modeling for Rapid System-Wide Energy Estimation of Reconfigurable Architectures. *ERSA*, 2002.

[3] G4 Executive Summary. http://developer.apple.com/hardware/ve/summary.html.

[4] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* 2003.

[5] J. Jang, S. Choi, and V. Prasanna. Area and Time Efficient Implementation of Matrix Multiplication on FPGAs. In *Proc. of The First IEEE International Conference on Field Programmable Technology*, CA, USA, Dec 2002.

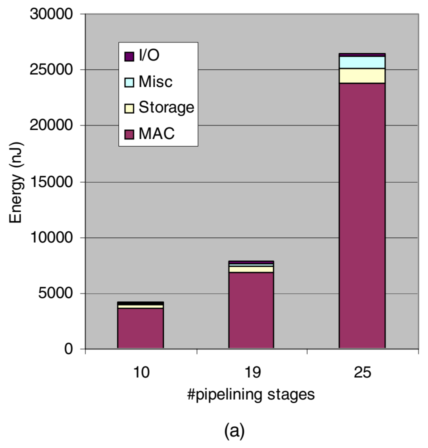
[6] J. Liang, R. Tessier, and O. Mencer. Floating Point Unit Generetion and Evaluation for FPGAs. In *Proc. of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'03)*, California, USA, April 2003.

[7] Nallatech. http://www.nallatech.com/.

[8] Quixilica. http://www.quixilica.com/.

[9] N. Shirazi, A. Walters, and P. Athanas. Quantitative Analysis of Floating Point Arithmetic on FPGA based Custom Computing Machines. In *Proc. of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'95)*, April 1995.

[10] Xilinx. http://www.xilinx.com.



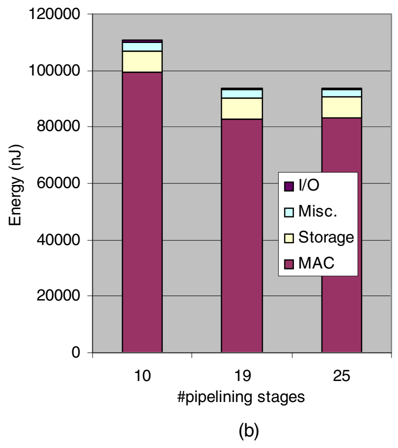
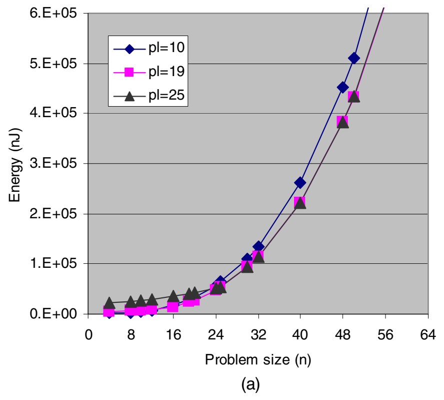
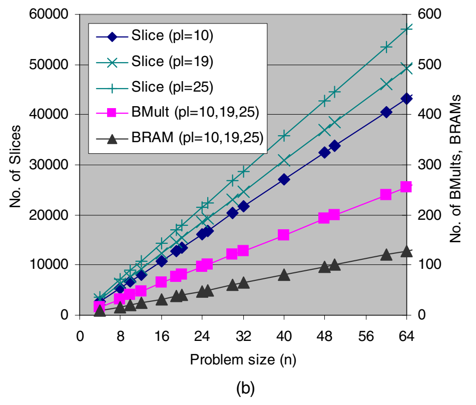


Figure 4. (a) Energy distribution (for problem size $n = 10$ and $n = 30$) for three different sets of floating-point units.





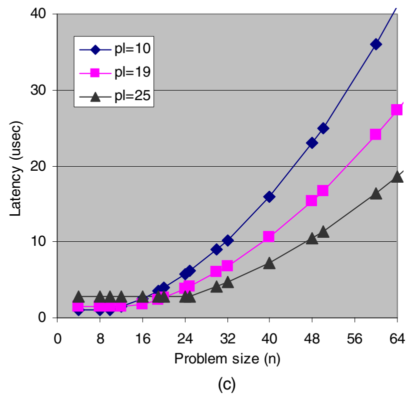
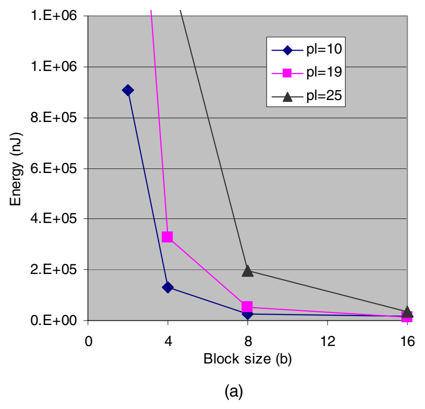
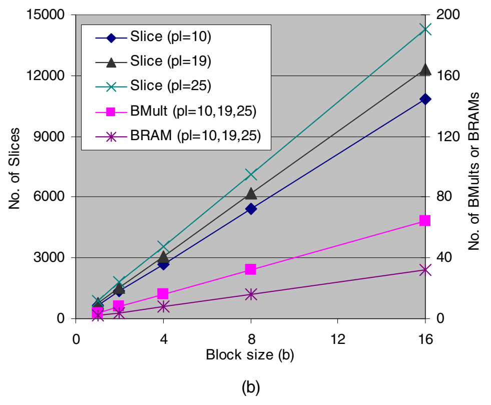


Figure 5. (a) Energy, (b) Resources, and (c) Latency for various problem sizes for three different sets of floating-point units.





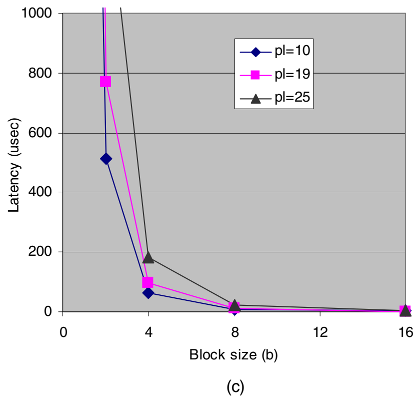


Figure 6. (a) Energy, (b) Resources, and (c) Latency for various block sizes for three different sets of floating-point units for problem size $n = 16$.