A Re-evaluation of the Practicality of Floating-Point Operations on FPGAs

Walter B. Ligon III Scott McMillan Greg Monn
Kevin Schoonover Fred Stivers Keith D. Underwood*

Planet Earth Research Lab†
Department of Electrical and Computer Engineering
Clemson University
105 Riggs Hall
Clemson, SC 29634-0915

{walt, spmcmil, gmonn, kschoon, fstiver, keithu}@parl.eng.clemson.edu
http://ece.clemson.edu/perl

Abstract

The use of reconfigurable hardware to perform high precision operations such as IEEE floating point operations has been limited in the past by FPGA resources. We discuss the implementation of IEEE single precision floating-point multiplication and addition. Then, we assess the practical implications of using these operations in the Xilinx 4000 series FPGAs considering densities available now and scheduled for the near future. For each operation, we present space requirements and performance information. This is followed by a discussion of an algorithm, matrix multiplication, based on these operations, which achieves performance comparable to conventional microprocessors. Algorithm implementation options and their performance implications are discussed and corresponding measured results are given.

1 Introduction

Implementation of floating-point operations is often avoided on reconfigurable computing platforms because these operations typically require too much area to be practical on current devices. Unfortunately, this does not mesh well with the way most scientific algorithms are written. Many algorithms require some form of fractional representation. The only option available in most programming languages is to declare a floating-point number. Often, these are based on one of the IEEE floating-point formats. Reconfigurable logic has the potential to yield significant speedup for many of these algorithms, though traditionally it has been unable to handle IEEE floating-point formats. There have been several studies to investigate methods of dealing with this incompatibility. One alternative is the use of fixed-point representations, which is acceptable in situations where the dynamic range of the numbers is limited; however, not all scientific applications have a sufficiently limited dynamic range.

We take another look at the floating-point problem on FPGAs in the context of drastically increasing densities. Our goal is to determine if, and when, FPGAs may become practical for use in algorithms requiring floating-point computations. Our implementations focus on using the IEEE floating-point standard for higher precision and deeper pipelining to achieve higher clock rates and greater performance. As device densities increase, this type of implementation will become more practical for use in various algorithms.

We have implemented an IEEE single precision floating-point adder and multiplier for Xilinx FPGAs. Each operation was implemented as a deep pipeline in order to reduce cycle time at the expense of latency. Thus, pipelined iterative algorithms are considered for the multiplier rather than an array structure,

*Keith Underwood is supported by an NSF Graduate Research Fellowship

†The Planet Earth Research Lab is supported by NASA GSFC under grant number NAG 5-3053. Donations were received from Xilinx under donation number 0000CU7-1978 and Synopsys under agreement number UPS-70004334

S	exp (+127)	mantissa

Figure 1: The IEEE single-precision floating-point format

and the adder performs normalization as a series of stages rather than a single stage barrel shifter. The multiplier is fully pipelined and runs at 33 MHz with a 15 cycle latency. The adder runs at 40 MHz and is fully pipelined with a 15 cycle latency. These functional units were combined into a single multiply- accumulate operation, which can be used to implement matrix multiplication.

This paper discusses the implementation and performance of IEEE single-precision floating-point addition and multiplication functional units and the implication of this performance in the context of a matrix multiplication algorithm. It begins with an introduction to previous work and a discussion of the IEEE floating-point format. The next section discusses the implementation of the floating-point units as well as the performance and device utilization of these units. Matrix multiplication on reconfigurable computing platforms is then discussed giving performance information. Finally, a conclusion with future directions is presented.

2 Background

For the same reason that conventional microprocessors have placed increasing emphasis on floating-point performance, there has been significant interest in processing floating-point numbers in reconfigurable computing systems. Many of the computationally intense algorithms that need the acceleration capabilities of reconfigurable computing require floating-point representations. Unfortunately, FPGAs are just reaching the point where they are dense enough to support a single floating-point unit. Results from [1] indicate that a floating-point adder requires 47% of an Altera FLEX 81188 while only yielding a performance of 7 MFLOPs. The multiplier requires 49% of an Altera FLEX 81188 and only yields 2.3 MFLOPs. An even more drastic resource requirement is seen in [2] where 4 Actel A1280 FPGAs were used to implement a floating-point multiplier and adder with relatively low performance. Since these studies have been done, however, FPGA density has increased dramatically. Groups using fixed logic to perform the multiplication and addition operations [3] have obtained better results. By using FPGAs to perform data manipulation and control operations and an external floating-point multiplier to implement a vector dot product, cumulative computation rates of 20 MFLOPs were achieved for a single board system with the option for extending a system to several boards. Another option that has been investigated is the use of reduced precision floating-point formats to conserve chip area. These alternative formats retain the benefit of a large dynamic range, but sacrifice precision. For a broad range of applications, the precision available from these reduced precision formats is adequate. The performance and space requirements of this type of implementation can be found in [4]. The IEEE single-precision floating-point format is described by ANSI/IEEE Standard 754-1985 and is shown in figure 1. This is the same standard used in most modern microprocessors. The most significant bit is a sign bit, S. The next 8 bits are the exponent in excess-127 representation where 127 is added to the real exponent. The special cases of 0 and 255 are reserved. This means that the exponent has a range of -126 to $+127$. The remaining 23 bits are used to represent the mantissa. The 24th (most significant) bit of the mantissa is an implied one. This gives 24 bits of significance and the value of the number represented is $(-1)S \times 1.m \times 2(exp - 127)$.

2.1 Integer Multiplication

A key component of floating-point multiplication is the multiplication of the mantissas, which is a 24 bit unsigned integer multiplication. Unfortunately, the multiplication of two 24 bit numbers in an FPGA is a significant problem, because there is insufficient area in many FPGAs to support the parallel array multipliers typically used in ASIC designs; however, there are alternative multiplier constructions. We were interested in both the smallest and the fastest implementations. For the smallest implementation, we considered bit serial multiplication and a booth recoding approach. For the fastest implementation, we used a digit serial approach which was a direct extension of the bit serial approach. In addition, we experimented with various degrees of pipelining of the bit serial and booth recoding approaches to explore the space-speed trade-offs.

The bit serial algorithm is the simplest to implement, so it was our first implementation. Unfortunately, it only resolves a single bit per cycle, and therefore requires several cycles to complete. The basic algorithm to multiply $A \times B$ is to start with a running sum equal to zero. On each cycle, take the next highest order bit of A and set:

$$sum = sum \times 2 + A_n \times B \quad (1)$$

Multiplication of two 24-bit numbers produces a 48-bit result. Since the format only stores 24 bits of significance, only the upper 24 bits of the result are required. By modifying the algorithm slightly, we can start with the lowest order bit of A and set:

$$sum = sum/2 + A_n \times B \quad (2)$$

The division is simply a right shift, but there is no need to retain the bits shifted off the right end, as they will no longer contribute to the result.

In addition to the bit serial approach, we decided to try two digit serial approaches. Digit serial approaches were also considered in [1]. Digit serial algorithms are used to resolve n bits of multiplication per cycle, where n is the digit size. In [1], they chose a four bit digit size. We chose a two bit digit to reduce the delay in a stage and increase the clock rate.

The booth recoding multiplication algorithm is designed to resolve two bits per clock cycle using a single adder. To do this, the algorithm considers three bits: the two bits currently being resolved, and one history bit. Assuming the multiplication is $A \times B$, the algorithm proceeds as follows. The history bit is initialized to 0. On each cycle, the history bit is set to the highest order bit of A resolved. A running sum is maintained that will be the result at the completion of the algorithm. This sum is initialized to 0. On each cycle, the new sum is computed by adding $m \times B$ where $m \in \{-2, -1, 0, 1, 2\}$ is selected based on the three bits being examined. This method requires half the number of cycles required by a bit serial algorithm.

Our third approach is a direct extension of the bit serial approach. The bit serial algorithm requires an adder and a multiplexor to resolve one bit of the multiplication. Multiple bits can be resolved in a single cycle by simply connecting several bit serial stages in series and passing data through them in one cycle. Since the combination of the adder and multiplexor forms a four input function, the two parts can be combined into one column of 4-LUTs. Unfortunately, our synthesis tool (Synopsys 1997.01) did not generate this structure, so building a digit serial part of this nature through direct synthesis was not possible. We chose to implement the add-mux construct as a macro and use it to build a digit serial multiplier. We chose a digit size of two since our primary goal was higher clock rates.

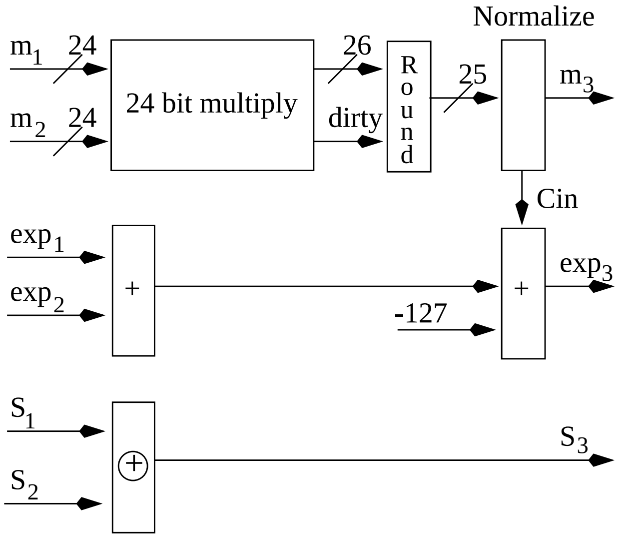

Figure 2: A block diagram of the floating-point multiplier implementation

3 Floating Point Unit Implementation

Two floating-point operations were implemented, multiplication and addition. As background, we describe the basic algorithm for each operation and the special cases of rounding and exceptions. We follow with the details of our implementation, the resources it requires, and the performance of the stand-alone units.

3.1 Floating Point Multiplier

IEEE single precision floating-point multiplication is essentially a 24 bit unsigned integer multiplication enhanced with logic to generate the appropriate sign, generate the exponent, and handle exception conditions. The biggest portions of space and time are consumed by the multiplication operation.

3.1.1 Algorithm

The basic algorithm for floating-point multiplication can best be explained with the formula:

$$s_3 \times m_3 \times 2^{exp_3} = s_1 \times m_1 \times 2^{exp_1} \times s_2 \times m_2 \times 2^{exp_2} \quad (3)$$

$$s_3 = s_1 \oplus s_2 \quad (4)$$

$$m_3 = m_1 \times m_2 \quad (5)$$

$$exp_3 = exp_1 + exp_2 - 127(+1) \quad (6)$$

The three components of the floating-point result (sign, exponent, and mantissa) can be computed independently. Some notes on the above equations are in order. First, m_3 would be a 48-bit number. It must be reduced to a 24-bit number to be stored in the IEEE format. Fortunately, this is a relatively simple task, because m_1 and m_2 are 24 bit numbers defined to be in the range [1,2). This implies that the result will be a 48 bit number in the range [1,4), which must be converted to a number in the range [1,2). This can be accomplished by retaining the 25 most significant bits of the result, and then based on the uppermost bit, shifting the result left one position or incrementing the exponent by one. The uppermost 24 bits of the result are retained. The first case handles results in the range [1,2) by aligning the result into the uppermost 24 bits. The second case handles results in the range [2,4) by reducing their significance and applying it to the exponent. The task is complicated only slightly more by the addition of proper IEEE rounding. A second note is about the generation of the exponent. The subtraction of 127 is necessary since the exponent for each input is in excess-127 notation. An increment by one will also be necessary in cases where the result mantissa is in the range [2,4).

3.1.2 Rounding and Exceptions

There are a few different modes of IEEE rounding specified in [5]. The one we chose to support, round to the nearest, is defined as the default. This rounding mode requires that the result returned be "the representable value nearest to the infinitely precise result" In the case of a tie, the result with a zero in the lowest order bit is returned. We accomplish this by retaining 26 bits of the result of the mantissa multiplication as well as a dirty bit that indicates whether any of the additional bits not retained were a one. We round before normalization, so we must be aware of the potential effects of normalization. If the highest order result bit is one, we will retain the uppermost 24 bits in the normalization stage, so we round the 24^{th} bit. We use a combination of the 25^{th}, 26^{th}, and dirty bits to determine whether to round up or down. If the highest order result bit is zero, we will retain bits 2 through 25, so we use the 26th bit and the dirty bit to perform rounding.

Other features specified by the IEEE specification are exception conditions and exception handling. We currently handle overflow and underflow conditions by setting the result to zero. The implementation will be enhanced in the future to correctly handle these exceptions. The only other relevant exception condition is invalid operation, which is caused by performing arithmetic on the reserved representations of ∞ and NaN. We have chosen not to deal with this exception to reduce resource requirements. This is reasonable since the host could test the data before sending it to the board if NaN or ∞ was likely to be in the data.

3.1.3 Implementation

A block diagram of the floating-point multiplier is shown in figure 2. The mantissas are multiplied, using one of the methods discussed earlier, and 26 bits of result are retained. The appropriate choice of multiplier is based on space and performance requirements. In addition, a single dirty bit indicating if any bits not retained were a one is also produced. Rounding is then performed with awareness of the possible need to normalize the result; thus, the uppermost bit of the result determines the bit position adjusted by rounding. Note that if rounding were performed after normalization, certain cases would lead to a result that needed to be normalized again. The normalization step then chooses to shift or not shift the result. Based on this, the normalization unit optionally sets Cin for the exponent adjustment. The exponent is calculated by first adding the exponents of the two inputs and then subtracting 127. The sign bit is simply computed as the XOR of the two input sign bits.

3.1.4 Resource Requirements

Table 1 shows the resource requirements of several different implementations and compares them to the space available in the two Xilinx devices we use as well as a much larger Xilinx device scheduled for release in the first half of 1998. The space requirements are given in terms of four input lookup tables (4-LUTs) and flip-flops, which are the basic building blocks of the Xilinx devices. It should be noted that these numbers only allow for a very simple data interface. Assuming the two multiplication algorithms can be implemented with the same clock cycle, the booth recoding based multipliers require about half as many cycles to produce a result (17) as the bit serial algorithm (28). Of the two iterative approaches, this would seem to make booth recoding the obvious choice, but the resource requirements shown indicate that the choice is not clear. For example, a 2 stage pipelined booth recoding multiplier has a 9 cycle pipeline stage latency while a 3 stage pipelined bit serial multiplier has a 10 cycle stage latency. The bit serial multiplier requires slightly fewer (4-LUTs), but significantly more flip-flops; however, the use of additional flip-flops is not significant since

Multiply Type	4-LUTs/ Flip-Flops	Xilinx 4020E	Xilinx 4062XL	Xilinx 40250XV
Booth Based	283 / 155	18% / 10%	6% / 3%	1.6% / 1%
Bit Serial	249 / 149	16% / 10%	5% / 3%	1.4% / 1%
2 Stage pipe Bit Serial	328 / 233	21% / 15%	7% / 5%	2% / 1%
2 Stage pipe Booth	410 / 231	26% / 15%	9% / 5%	2% / 1%
3 Stage pipe Bit Serial	390 / 304	25% / 19%	8% / 7%	2% / 2%
3 Stage (6 cycle) Booth	530 / 301	34% / 19%	12% / 7%	3% / 2%
5 Stage (6 cycle) Bit Serial	514 / 446	33% / 28%	11% / 10%	3% / 3%
Fully Pipelined Digit Serial	759 / 1017	48% / 65%	16% / 22%	4.5% / 6%

Table 1: Resource utilization of various floating-point multiplier implementations on various Xilinx devices

many of the flip-flops in the booth recoding design would go unused. A similar scenario can be seen in the comparison of a 3-stage (6-cycle) booth recoding multiplier and a 5-stage (6-cycle) bit serial multiplier. It is not yet clear if there is a significant advantage to either of the multiplier implementations.

The fully pipelined multiplier is far more area efficient than the other two implementations since it takes advantage of the 4-LUTs by combining the adder an mux in one stage. Unfortunately, this design is too big to fit in a single 4020E after interface logic to support the host interface on the implementation platform are added.

3.1.5 Performance

All of the multiplier implementations have been tested at 33 MHz; thus, the fully pipelined multiplier has a peak performance of 33 MFLOPs in a 4020E, if a small enough interface to the chip is used. Moving to larger devices allows several multipliers in a single chip. The 4062XL should support three fully pipelined multipliers. The 40250XV could be expected to hold nearly four times as many. Assuming that a 40250XV would hold 12 such multipliers interconnected in some fashion, and faster speed grades could be used to push the clock rate to 40 MHz, this would yield a single chip performance of 480 MFLOPs.

3.2 Floating Point Adder

3.2.1 Algorithm

The floating-point addition algorithm is more complicated than the multiplication algorithm. The first step is to choose the number with the greater magnitude, place it in A, and place the smaller in B. The difference in the exponents, N, is then computed. This difference is then used as the shift input to a barrel

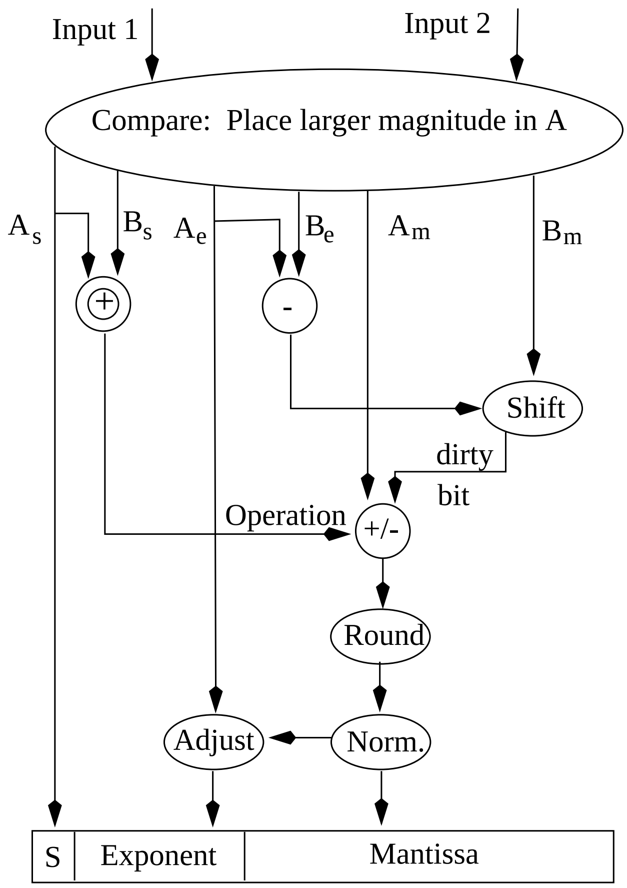

Figure 3: The floating point addition algorithm

shifter. The mantissa of B is shifted right N positions to properly align it with A. The sign bits are compared in parallel with the exponent subtraction to determine if an addition or subtraction operation is required. The final sign bit will be the same as the sign of the larger input. A dirty bit is produced by examining B as it is shifted. The operation is then performed on the mantissas of A and B ($A+B$ or $A-B$). The result must be rounded and then normalized to place the most significant one in the highest order bit. The result is composed of the sign of the largest input, the exponent of the largest input adjusted by the number of shifts in the normalization step, and the normalized result of the operation performed on the mantissas of A and B. Data flow for this algorithm is shown in figure 3.

3.2.2 Rounding and Exceptions

Rounding is handled in a similar fashion to that of the multiplier. The same rounding mode is used, and again we produce a dirty bit. In this instance, however, we only retain 25 bits of result. The 25^{th} bit and dirty bit are used to round the 24^{th} bit according to the round to nearest standard.

Exception conditions are currently handled for the adder in the same way as they are handled for the multiplier. Underflow and overflow are handled by setting the result to zeros. Invalid operation is not handled.

3.2.3 Pipeline Stage Division

Deciding how to divide the implementation into pipeline stages is not trivial. The most important requirement is to produce a final design that runs at 33 MHz; therefore, the compare operation, the initial alignment shift, and the normalization shift has to be sub-divided into iterative or pipelined parts. The comparator requires two stages, with a third stage swapping the data values if necessary, since a 31-bit comparison can not be done in a single 33 MHz cycle. The smaller value is then fed into a shifter, which is broken into three non-identical pipeline stages. The operation, a 24 bit add or subtract, and round elements each require a separate stage. The final normalization requires five stages, four for shifting the final result and a fifth for dealing with exception conditions that can occur during normalization. Three of the four shifting stages from the normalization element are identical. This is the only part of the adder that can reuse logic. This explains the insignificant difference in the size of the pipelined and iterative adders shown in table 2. A third implementation could be constructed in which the normalization element reuses a shifter 3 times, but the minimal logic savings do not justify the loss of performance. The result is an adder with a fairly high overall latency (15 cycles), but a fairly high clock rate (40 MHz).

3.2.4 Normalization

The final normalization is one of the most difficult parts of the design to implement efficiently. It requires that the result be shifted by 0 to 24 bits, based on the data value. The exponent must be adjusted accordingly. In a case where all 24 bits are zero, a special zero value must be loaded into the result. We handle this normalization by passing the data through a series of shifters. Each shifter is accompanied by logic for determining how many bits the data is to be shifted. Each bit of each shifter along the path is designed to fit in a single CLB. The first three stages each shift the result by 0, 1, 4, or 8 bits. The final step performs a shift of 0, 1, 2, or 3 bits. Since each of these shifts can be handled in a single stage of CLBs, they will easily run at 33 MHz, or even higher. The goal is to build four stages of four input muxes with control based on three inputs.

3.2.5 Resource Requirements

Notice from table 2 that the resource requirements for the fully pipelined design are not significantly greater than that of the iterative design in terms of 4-LUTs consumed; however, it consumes significantly more flip-flops. This is because the adder has many different operations that must be performed on the data, whereas an iterative multiplier can repeatedly reuse a single adder. The logic for each of these operations must be present whether the device is purely iterative or fully pipelined. The only reusable logic in the case of the iterative implementation is in the final normalization in which there are three identical shifts. Thanks to the flip-flop rich environment of the FPGA, the penalty in terms of total device area consumed is fairly insignificant. Again, the interface included in these resource utilization figures is based on data full and empty signals.

3.2.6 Performance

The fully pipelined adder has been tested at 40 MHz in –3 speed grade parts; thus, it is capable of performing 40 MFLOPS. The next step is to consider the number of adders that can be implemented. Based on the

Degree of Pipelining	4-LUTs/ Flip-Flops	Xilinx 4020E	Xilinx 4062XL	Xilinx 40250XV
Iterative	563 / 332	36% / 22%	12% / 8%	3% / 2%
1 Cycle (Fully Pipelined)	629 / 671	40% / 43%	14% / 15%	4% / 4%

Table 2: Resource utilization of an IEEE single-precision floating-point adder

numbers in table 2, and assuming that only 70-80% of an FPGA can be effectively utilized, we can see that a Xilinx 4020 device can only support one fully pipelined adder. However, the higher end devices can conceivably support several such adders. A Xilinx 40250XV for example, could possibly support 16 or more interconnected adders. Now, supposing some application which could use sixteen adders interconnected in some fashion, this would yield a throughput of 640 MFLOPs from a single FPGA. Using faster speed grades and increasing the clock to 50 MHz could push this number to 800 MFLOPs.

4 Application Performance

We used the floating-point adder and multiplier to construct multiply-accumulate units (MACs). The multiply-accumulate units were then used to build a matrix multiplication application on a GigaOperations platform and an Annapolis MicroSystems platform.

4.1 Implementation Platform

We use two different platforms to evaluate our design implementations. The first is a GigaOperations G900 board with two XMODs. Each XMOD holds two Xilinx 4020E-3 parts for a total of 4 FPGAs. The two chips have 32 bits of interconnection available and 128 K of SRAM each. The SRAM uses a 16 bit wide data path which can be accessed at 30 MHz allowing a RAM bandwidth of 60 MB/s for each chip. We experimented with configurations involving one multiply-accumulate per chip and one multiply-accumulate per XMOD. The host to chip interface was customized to support broadcast operations.

The second system we evaluate is an Annapolis MicroSystems Wildforce board. It contains 4 Xilinx 4062XL-3 parts. Each chip is connected to 1 MB of SRAM. We expect this to give us a RAM bandwidth of approximately 132 MB/s for each chip.

4.2 Matrix Multiplication

Implementation of matrix multiplication in reconfigurable hardware attempts to exploit the massive amount of parallelism available in such a matrix algorithm. For matrix multiplication, the key performance factor is the rate at which multiply-accumulate (MAC) operations can be performed. This is impacted by the number of MAC units available, the number of cycles to perform a MAC, the cycle time of a MAC, and the rate at which data elements can be delivered to the MAC unit. The general matrix multiply algorithm is unaffected by the type of data element involved.

Consider the case of $A * B = C$, where A, B, and C are matrices of dimension N. The general algorithm we use is to send a different column B to each of K different multiply accumulate units, where K is assumed to be an even multiple of either the number of modules, or the number of chips available. The columns of B are stored temporarily in local SRAMs. The columns of A are then broadcast to all K multiply accumulate units, which results in K rows of C being generated. This process is repeated N/K times for a total computation time of $\big((N/K) \times T_C + N^2 \times T_M\big)$, where T_C is the time required to place K columns of B on the board and T_M is the total time to consume an element. It should be noted that T_M is actually $\max(T_P, T_B)$ where T_P is the time required to complete one pipeline stage and T_B is the time required to broadcast a new data element to all of the functional units. The max relationship is achieved by overlapping processing with communication such that one element is always buffered at the input to the functional unit.

Assuming a fixed T_C and T_B, there are two alternatives to improve algorithm performance: reduce T_P or increase K. Unfortunately, the amount of speedup that can be achieved by decreasing T_P is limited by the data broadcast time T_B, and is ultimately limited by the cycle time of the functional unit. Decreasing T_P and increasing K both require increasing the amount of hardware resources consumed. Since many current hardware platforms have a high cost associated with broadcasting a single 32 bit data element, it is often easier to reap the benefits of increasing K. Unfortu-

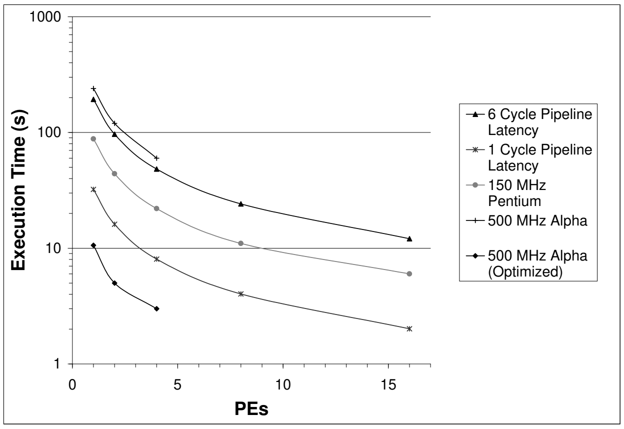

Figure 4: Projected execution times vs. number of multiply accumulate (MAC) units for several different pipelining options at 33 MHz

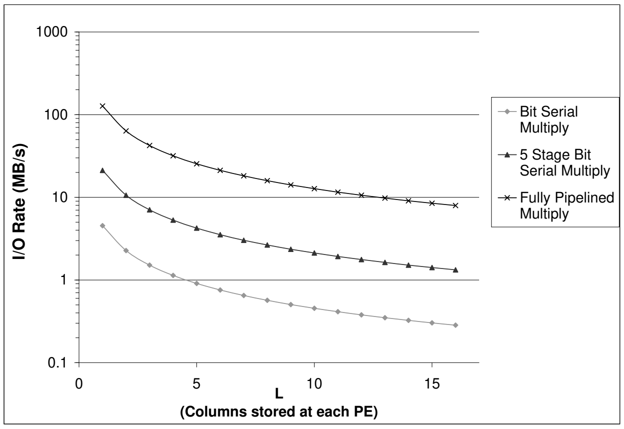

Figure 5: I/O requirements as a function of L for different pipelining options

nately, increasing K has the drawbacks of complicating control logic, replicating control logic associated with a component, and complicating placement and routing. It is easier to place and route one heavily pipelined functional unit in a device than it is to place and route four functional units, especially when those four units would be required to use the same input and output data paths.

An alternative approach would be to modify the algorithm to lower the dependence on T_B. One such enhancement is to simultaneously compute multiple elements of the result matrix C on one functional unit. This can be achieved by storing L columns at each functional unit. When an element of A is broadcast, it is multiplied by the corresponding element of each column of B stored at the functional unit; thus, $L-$ MACs would occur for each element broadcast. T_P could then be reduced to the cycle time of the device, as long as $L \geq J$ where $T_B = J \times T_P$. The drawback here is the need to store L intermediate values for the accumulate operation. Fortunately, a FIFO can be used. The FIFO would be initialized by the insertion of L zeros. Then, for $N \times L$ cycles, the adder would take one input from the multiplier and the other from the FIFO. The output of the adder would go to the FIFO. The results would then be read from the device and the FIFO re-initialized. Figure 4 plots the time in seconds for a matrix multiply versus the number of MACs for various values of T_P. For this figure, N was taken to be 1024, L was assumed to be greater than or equal to J, and T_C was considered to be negligible. The values of T_P used were computed by assuming a 30 ns clock cycle and multiplying by the number of clock cycles required for a pipeline stage. In addition, it plots timing data for three different multi-processor implementations using PVM. The first was done on a 16 node 150 MHz Pentium based machine interconnected with a 100 Mb/s Fast Ethernet switch. The other two were done on an 4 node 500 MHz Alpha 21164 based machine interconnected by a 100 Mb/s Ethernet hub. The first of these two was written as a tripply nested loop and compiled with the DEC C compiler and all relevent compiler optimizations. The second was blocked and unrolled by hand and achieved a speedup of over 20 (listed as optimized). Both numbers are given as performance comparisons since there are many who trust the compiler to perform all of the necessary optimizations for peak performance.

One further complication is that the functional units cannot process data faster than it can be delivered. For this reason, it is necessary to consider the I/O and RAM bandwidth requirements for each implementation. The I/O dependency is already accounted for by the dependency on T_B. T_B is actually based on the average time required to deliver an element from host memory to the functional unit. The I/O requirement is computed based on the rate at which data must be delivered from the host to the functional units in order to fully utilize the functional unit. The RAM bandwidth required is different from the I/O requirements in some cases because some configurations involve a greater than one to one ratio between the number of elements broadcast and the number of MACs performed. Assuming the data broadcast to the functional units is a 32 quantity, the I/O rate required in MB/s can be computed as: $4 \times (1/T_P)/L$, where L is the number of columns simultaneously

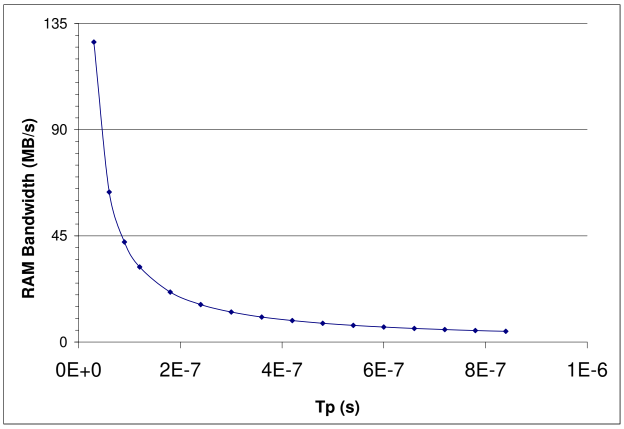

Figure 6: The RAM bandwidth required as a function of T_P

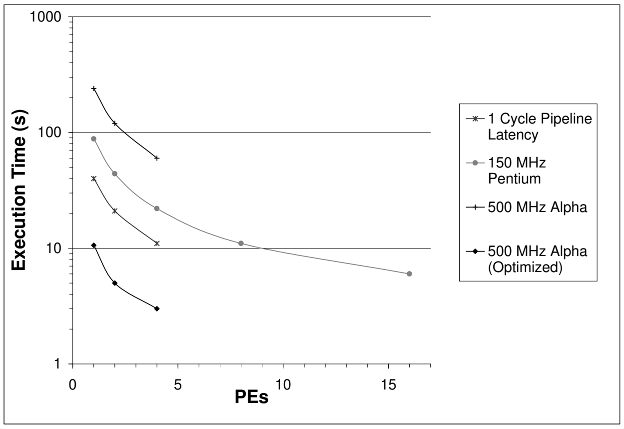

Figure 7: Measured execution time vs. number of multiply accumulate (MAC) units for a fully pipelined implementation at 33 MHz

Number of MAC Units	Predicted Performance	Measured Performance
4	354 s	385 s
8	177 s	190 s
16	87 s	95 s

Table 3: Predicted and measured performance for an integer matrix multiply

processed at a MAC. The required RAM bandwidth per processing element assuming 32 bit operands is: $4 \times (1/T_P) \times (K/PE)$ where PE is the number of FPGAs and K/PE is the number of MAC units per FPGA. Figures 5 and 6 show the theoretical I/O requirements and corresponding RAM bandwidth requirements of a variety of potential implementations.

4.3 Signed Integer Matrix Multiply

In order to validate some of the performance estimates, the matrix multiply was initially implemented with signed integers. This provided a good initial test since a 32 bit signed integer multiply has similar space and time requirements to the floating-point multiply, but was easier to implement. Initial testing was done at 15 MHz. The cases that have been tested were based on an iterative booth recoding multiplier that required 20 cycles to produce a result. Tests were done with varying numbers of MAC units. The results are shown in table 3. These test points offer a preliminary indication that the equations predict performance fairly well (within 10%). The accuracy is expected to improve as the number of columns simultaneously processed per MAC unit, L, is increased, since, most of the difference is attributed to card I/O issues.

4.4 Floating-Point Matrix Multiply

The floating-point matrix multiply requires slightly more space than the integer matrix multiply since the floating-point adder is significantly bigger than the integer adder. For example, a single iterative floating-point multiply accumulate when combined with interface logic requires enough chip area to make it difficult to place and route on a 4020E, while two iterative integer multiply accumulates would fit on a single FPGA. Our only floating-point test on the G900 system split one multiply-accumulate across two chips on one XMOD at 15 MHz. This gave us 2 MAC units with floating-point multipliers pipelined to 6 cycles with L set to 16. It required 264 seconds to complete a 1024x1024 matrix multiply.

Figure 7 shows the performance results measured on the Annapolis MicroSystems board. This implementation used four 4062XL-3 parts with one fully pipelined multiply accumulate running at 33MHz on each chip and L set to 16. The best time for a 1024x1024 matrix multiply was 11 seconds giving a performance of 195 MFLOPs, compared to a theoretical peak of 264 MFLOPs. Again, the difference in peak and actual performance is attributed to I/O issues. Figure 7 also includes the three workstation based implementations for comparision. Note that the four PE (one system) reconfigurable implementation is comparable to the optimized Alpha implementation, and outperforms the other Alpha implementation as well as the one, two, and four processor Pentium implementations.

5 Conclusions and Future Work

This paper has offered evidence that floating-point implementations in FPGAs should be considered as a possible alternative for high precision applications. We have shown an adder implementation fitting in a Xilinx 4020E that can achieve a performance of 40 MFLOPs. We have also shown a multiplier implementation fitting in a Xilinx 4020E that is capable of performing at 33 MFLOPs. Although these performance numbers are not comparable to those achieved by modern microprocessors, the use of several of these devices in a reconfigurable computing system can achieve speedup for some algorithms, particularly matrix multiplication. A system with four 4062XL parts can be configured to perform a peak of 264 MFLOPs with a realized application performance of 195 MFLOPs. Xilinx projects the availability of 40250XV parts, which are nearly four times as dense, in the first half of 1998. By increasing the number of MAC units in each part, we could expect to achieve six times the performance, if adequate I/O and RAM bandwidth were available. Further performance gains could be achieved by moving to better speed grades. We are currently using -3 parts, but -09 parts are becoming available. In addition, even higher density devices are projected by the end of 1998.

The point of this investigation was to explore when, or if, the use of FPGA technology would offer a performance boost for floating-point applications. Although we do not get a significant performance improvement over some of the fastest available processors, we do see comparable performance. In fact, we get a slight speedup with a single FPGA based solution over a multi-processor implementation on a recent architecture (the Pentium). If we can achieve comparable performance from a pure floating-point application, it is a good indication that applications which require a few floating-point operations intermixed with fixed-point computations can now be considered as implementation targets for reconfigurable computing. Furthermore, these results indicate that if device density and speed continue to increase, reconfigurable computing platforms may soon be able to offer a significant speedup to pure floating-point applications.

We intend to continue to investigate applications which require some floating-point computation and can benefit from the reconfigurability of FPGAs, focusing first on matrix algorithms. Our next target applications are Gaussian Elimination and a Gauss-Seidel iterative solver.

References

[1] L. Louca, T. A. Cook, and W. H. Johnson, "Implementation of IEEE single precision floating point addition and multiplication on FPGAs," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 107–116, 1996.

[2] B.Fagin and C. Renard, "Field programmable gate arrays and floating point arithmetic," *IEEE Transactions on VLSI*, vol. 2, no. 3, pp. 365–367, 1994.

[3] P. Bakkes, J. du Plessis, and B. Hutchings, "Mixing fixed and reconfigurable logic for array processing," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 118–125, 1996.

[4] N. Shirazi, A. Walters, and P. Athanas, "Quantitative analysis of floating point arithmetic on FPGA based custom computing machines," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 155–162, 1995.

[5] I. S. Board, "IEEE standard for binary floating-point arithmetic," Tech. Rep. ANSI/IEEE Std 754-1985, The Institute of Electrical and Electronics Engineers, New York, 1985.