# Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs

Loucas Louca, Todd A. Cook, William H. Johnson
Dept. of Electrical and Computer Engineering
P.O. Box 909
Rutgers University
Piscataway, NJ 08855-0909
{louca,tac}@ece.rutgers.edu

## Abstract

Floating point operations are hard to implement on FPGAs because of the complexity of their algorithms. On the other hand, many scientific problems require floating point arithmetic with high levels of accuracy in their calculations. Therefore, we have explored FPGA implementations of addition and multiplication for IEEE single precision floating-point numbers. Customizations were performed where this was possible in order to save chip area, or get the most out of our prototype board. The implementations tradeoff area and speed for accuracy. The adder is a bit-parallel adder, and the multiplier is a digit-serial multiplier. Prototypes have been implemented on Altera FLEX8000s, and peak rates of 7MFlops for 32-bit addition and 2.3MFlops for 32-bit multiplication have been obtained.

**Keywords:** Computer arithmetic, FPGAs, custom computing machines, digit-serial arithmetic, floating point arithmetic.

## 1 Introduction

Floating point operations are hard to implement on FPGAs because of the complexity of their algorithms. They usually require excessive chip area, a resource that is always limited in FPGAs. This problem becomes even harder if 32-bit floating point operations are required.

On the other hand, many scientific problems require floating point arithmetic with high levels of accuracy in their calculations. Furthermore, many of these problems have a high degree of regularity that makes them good candidates for hardware accelerated implementations. Thus, the necessity for 32-bit floating point operators implemented in FPGAs arises.

In an effort to accommodate this need, we use the IEEE 754 standard for binary floating point arithmetic (single precision) to implement a floating-point adder and a floating-point multiplier on Altera FLEX 8000 FPGAs. The adder is built in a parallel structure and pipelining techniques are used to increase its throughput. The multiplier was implemented with a digit-serial structure because a 32-bit parallel multiplier is so large that it is virtually impossible to fit in a single FPGA chip. Using digit-serial arithmetic, the 32-bit multiplier fits in 1/2 of a FLEX 81188. Earlier attempts to fit various bit-parallel designs failed because more than 100% of the logic elements were required. The design details, speed, and area requirements of these operators are discussed in this paper.

We begin the paper by briefly reviewing the IEEE 754 standard for binary floating point arithmetic. In the next two sections, the adder and the multiplier are described in detail. Then we demonstrate how to interconnect the adder with the multiplier, and following that, we present a small study that demonstrates the area-speed tradeoff. Finally, we conclude with applications of the operators and prospects for future improvements in our implementations.

## 2 Related Work

Similar work was presented by Shirazi *et. al* [5], and as is noted there, mapping difficulties occur due to the complexity of floating point arithmetic. This complexity implies a large chip area. In their work, they investigate ways of implementing the floating point operators in the best combination of speed and area. In order to minimize the area requirements, small floating point representation formats (16 bits and 18 bits wide instead of the full IEEE 32-bit wide format) are used. Furthermore, pipelining techniques are applied in order to produce a result every clock cycle. The operators presented there were implemented for a signal processing application, so the customized format used for the representation of floating point numbers

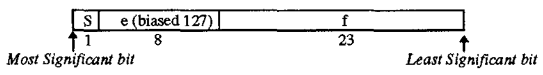


Figure 1: IEEE Floating Point Format

has the property that it suits the specific needs of that application.

By using a small format, smaller and faster implementations can be built. However, less accuracy is achieved in the calculations. In our implementations, accuracy is the main objective, which is why 32-bit operators were designed. This translates to more chip area in the case of our bit-parallel adder, and less speed in the case of our digit-serial multiplier. These trade-offs are considered fair since accuracy is very important for certain algorithms, such as the gravitational n-body problem for which these operators were originally developed [6].

## 3 Floating Point Format

As mentioned above, the IEEE Standard for Binary Floating Point Arithmetic (ANSI/IEEE Std 754-1985) [1] will be used throughout our work. The single precision format is shown in Figure 1 . Numbers in this format are composed of the following three fields:

**1-bit sign, S:** A value of '1' indicates that the number is negative, and a '0' indicates a positive number.

**Bias-127 exponent,** $e = E + bias$**:** This gives us an exponent range from $E_{min} = -126$ to $E_{max} = 127$.

**Fraction, f:** The fractional part of the number.

The fractional part must not be confused with the significand, which is 1 plus the fractional part. The leading 1 in the significand is implicit. When performing arithmetic with this format, the implicit bit is usually made explicit. To determine the value of a floating point number in this format we use the following formula:

$$Value = (-1)^S \times 2^{e-127} \times 1.f_{23}f_{22}f_{21}.....f_0$$

For a detailed discussion with examples, see [4].

## 4 Floating Point Adder

Floating point addition is difficult to implement because the significands are represented in sign-magnitude format. The difficulty arises because depending on the signs of the two operands, the addition could become a subtraction, thus requiring one of the operands to be complemented. For addition, there could be a carry-out, in which case the result will be de-normalized. For subtraction, a negative result may be obtained which means that both the sign bit and the significand need to be inverted [4]. An algorithm implementing floating-point addition must take into account all of these possibilities. The algorithm we use is explained in the following subsection.

### 4.1 Addition Algorithm

In this section, we explain the algorithm we use for implementing floating point addition. Given two numbers $N_1$ and $N_2$, we can use the flowchart in Figure 2 to compute their sum, given that $e_1$,$e_2$ and $s_1$,$s_2$ are the exponents and significands of the numbers, respectively. A detailed description of the algorithm follows:

1. Make the 24th bit (hidden bit) explicit. If $e_i = 0$, ($N_i = 0$) make it a '0', otherwise make it a '1'. At this point 33 bits are needed to store the number, 8 for the exponent, 24 for the significand and 1 for the sign.

2. Compare $e_1$ and $e_2$. If $e_2 > e_1$, swap $N_1$ and $N_2$. Note that if a swap takes place, future references in the flowchart to $s_1$ ($e_1$) will be referring to the old $s_2$ ($e_2$) and vice versa. Also, the absolute difference in the exponent values ($|e_2\text{-}e_1|$) needs to be saved.

3. Shift $s_2$ to the right by an amount equal to $d = |e_2 - e_1|$. Fill the leftmost bits with zeros. Note that both numbers are now in a simple sign/magnitude format.

4. If $N_1$ and $N_2$ have different signs, replace $s_2$ by its two's complement.

5. Compute the significand, $S$, of the result by adding $s_1$ and $s_2$.

6. If $S$ is negative, replace it by its two's complement. For $S$ to be negative, all of the following conditions should be true:

   (a) $N_1$ & $N_2$ have different signs.

   (b) The most significant bit of $S$ is '1'.

   (c) There was no carry-out in step 5.

7. Normalization Step.

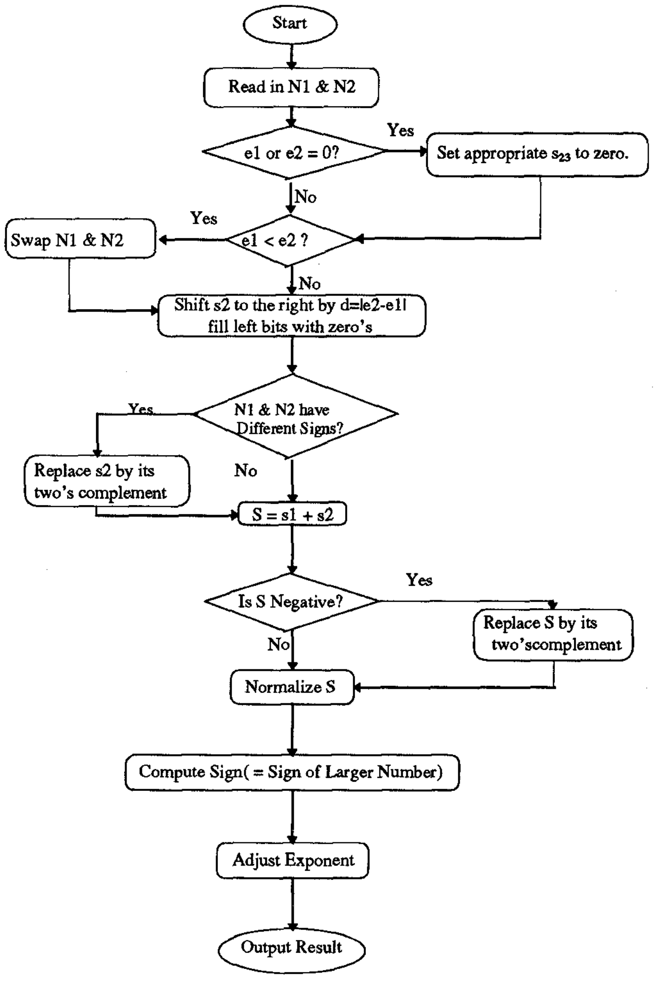


Figure 2: Floating Point Addition Algorithm

(a) If $N_1$ & $N_2$ have the same sign and there was a carry-out in step 5, then shift $S$ right by one, dropping the least significant bit and filling the most significant bit with a '1'.

(b) Otherwise, shift $S$ left until there is a '1' in the most significant bit. The number of left shifts must be stored.

(c) If $S$ was shifted left more than 24 times, the result is Zero

8. The sign of the result is determined by simply making the output sign the same as the sign of the larger of $N_1$ and $N_2$. The most significant bit of (the 24-bit wide) $S$ is replaced with this sign bit.

9. The resultant exponent ($e_1$) is adjusted by adding the amount determined in step 7. If it was determined in step 7(c) that $S = 0$, set the exponent to zero.

10. Assemble the result into the 32 bit format.

(The various rounding options defined by the IEEE standard are not yet implemented.)

## 4.2 Adder Implementation

The hardware implementation of the floating point adder is outlined in Figure 3. This basically reflects the algorithm presented above in Section 4.1. Two points worth noting are the hidden bit extraction and the re-assembly of the result into the 32-bit format.

First, the implicit bit for each of the operands must be made explicit. Although most of the time this will be a '1', the possibility of the number being zero must not be neglected. According to the IEEE standard,

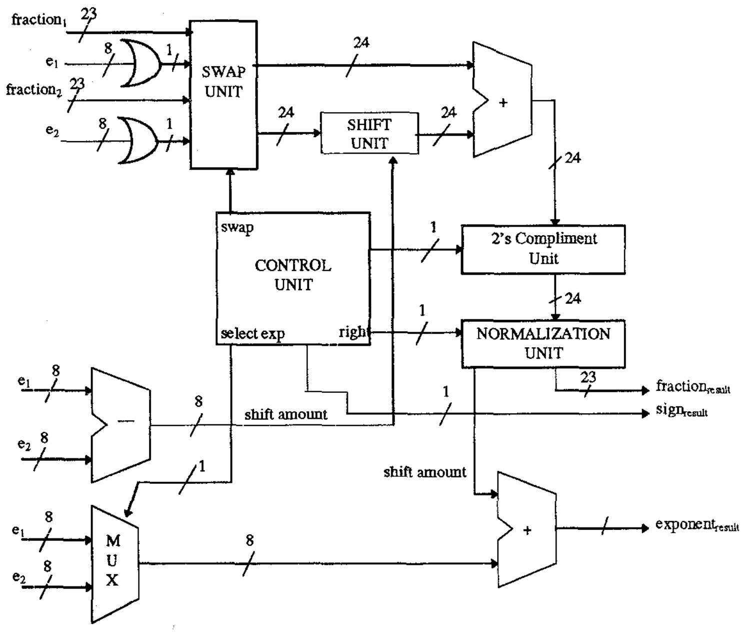


Figure 3: Floating Point Adder Circuit

when the biased exponent and the fraction field are '0', then the number represented is zero. Thus, in order to extract the correct bit, 2 8-input OR gates are used. If all bits of the exponent are '0', then this number is zero and its $24^{th}$ bit will be zero. Otherwise a '1' is inserted.

Once the result of the addition is obtained, it must be converted back into the 32-bit format. The normalization unit shifts the result left until the most significant bit set to '1' is in position 24. After the number is normalized, the $24^{th}$ bit is replaced by the sign of the result. Also, the exponent selected as the result exponent must be adjusted to reflect the shiftings that took place; thus, this shifting amount is added to $e_1$ to get the correct exponent.

At this point, all 32 bits of the result are available so they can be passed to the next operator or stored in memory.

## 4.3 Timing analysis

When a worst case timing simulation is performed with the MAX+Plus II Timing Analyzer tool, a delay of 385ns is estimated (when targeting an EPF81188-3). By using pipelining, the design can be broken into 3 stages, which significantly increases the throughput. In the first stage algorithm steps 1 and 2 are performed, in the second stage steps 3 through 6, and in the third step, steps 7 through 10. With this pipelining, a peak rate of about 7MFlops can be achieved.

## 4.4 Area Requirements

The main focus in the design (after accuracy) was for area minimization. The actual logic design of a floating point adder is not too complex, and most of the design time was spent trying to minimize the logic resources used. The design in its final state takes up slightly more than 47% of the chip. The largest components are the normalization unit and the shift unit, which take 152 and 123 logic cells, respectively. The swap unit, two's complement unit, and 24-bit adder for the significands also consume significant area, though not as large as the other two. The swap unit and the adder take 48 logic cells each, and the two's complement unit takes 38 logic cells. The remaining logic cells are consumed by the two 8-bit adders/subtractors and the logic for the control unit.

The initial version of the design took approximately 725 logic cells, which is about 72% of an Altera 81188 chip. The area reduction was mainly achieved by using the arithmetic mode of the FLEX 8000 logic elements wherever possible. This final result of 47% is much better, but future effort will be allocated to make it even smaller.

## 5 Floating Point Multiplier

Although multiplication is a simpler operation than addition, we are still faced with speed and chip area minimization problems when trying to implement it using FPGAs. This is especially true if we are using a 32-bit format, as in our case. In this section, we present the multiplication algorithm, introduce digit-serial arithmetic, and explain the 24-bit multiplier in detail.

### 5.1 Multiplication Algorithm

The sign-magnitude format simplifies the multiplication operation because it is very similar to an integer format. The only additional step required is the calculation of the correct exponent.

As explained in Section 3, a binary floating-point number is represented by a sign bit, the significand and the exponent. Thus, if two numbers are to be multiplied, the result will be given by this formula:

$$S_1 \cdot 2^{e_1} \times S_2 \cdot 2^{e_2} = S_1 \times S_2 \cdot 2^{e_1+e_2}$$

The addition of the exponents is a trivial operation as long as we keep in mind that they are biased. This means that in order to get the right result, we have to subtract 127 (bias) from their sum. The sign of the result is just the XOR of the two sign bits. The multiplication of the significands, as explained above, is just an unsigned, integer multiplication. A full explanation of the method used to perform this multiplication is given in Section 5.3, after we first introduce digit-serial arithmetic.

### 5.2 Digit-Serial Arithmetic

Arithmetic operators (both integer and floating-point) can be implemented using different styles, such as bit-serial, bit-parallel, and digit-serial [3]. The first one processes one bit of the operands per clock cycle, whereas the second processes all bits of the operands in one clock cycle. It should be obvious that the bit-parallel yields maximum speed, whereas bit-serial yields minimum area.

Digit-serial arithmetic fulfills the need for something in between these two extreme points. With digit-serial operations, we process more than one, but not all the bits, in one clock cycle [3].Thus, we can trade-off speed for area or vice-versa.

*Digit-size* is defined as the number of bits processed per clock cycle. Bit-serial and bit-parallel are special cases of digit-serial arithmetic. The first one has a digit-size of 1, and the second one a digit-size of $D$, where $D$ is the width of the data to be processed. For example, a 24-bit wide operation would be executed in 1 cycle in bit-parallel, 24 cycles in bit-serial, and 6 cycles in digit-serial with a digit-size of 4.

Digit-serial arithmetic was chosen in this implementation because it has the advantage that by varying the digit-size, the best trade-off of speed and area can be found. This is extremely important for applications developed on FPGAs, where logic elements are a scarce resource. Complex applications, such as the gravitational n-body problem [6], would be too large to implement using 32-bit floating point numbers and bit-parallel arithmetic.

### 5.3 Digit Serial Multiplier

Although our multiplier is a 24-bit implementation with a digit-size of 4, we will illustrate the method using a 4-bit multiplier with a digit size of 2. This simplifies the presentation of the multiplier without sacrificing any important details.

#### 5.3.1 Example Multiplication

In order to understand digit-serial multiplication, it is appropriate first to study the way it is carried out by hand. As an example, the multiplication

$$1011 \times 1001 = 01100011$$

(in unsigned binary) will be demonstrated.

In digit-serial multiplication, the bits of the multiplier, 1011, are presented in parallel and the bits of the multiplicand, 1001, are presented in digit-serial format, least significant digit first. In the first step, the least significant digit, 01, of the multiplicand is multiplied by the multiplier. The result of this operation will include the least significant digit of the result, 01100011. At this point, it is necessary for the partial sum and carry words to be saved so that they can be combined with the result of the following digit multiplication. An illustration of this multiplication is given below:

| | | | | | | |
|---|---|---|---|---|---|---|
| | | 1 | 0 | 1 | 1 | |
| | | | | 0 | 1 | $\times$ |
| | | 1 | 0 | 1 | 1 | |
| | 0 | 0 | 0 | 0 | | + |
| | 0 | 1 | 0 | 1 | 1 | |
| Saved Sum: | 0 | 1 | 0 | 1 | | |
| Saved Carries: | 0 | 0 | 0 | 0 | | |

In the next step, the second least significant digit, 10, is multiplied with the multiplier and new partial products are obtained. However, before proceeding with their addition, the previously saved sum and

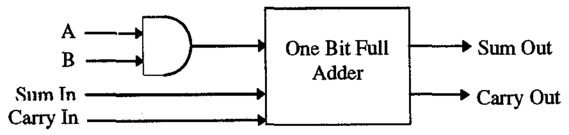


Figure 4: Bit Multiplier

carry words must also be added to them. This is to compensate for the fact that partial products of the second digit multiplication have to be added to the partial products of the first digit multiplication in order to reproduce the results of the bit-parallel multiplication. The sum word is shifted once to the left because it has half the weight of the carry word, and then the addition is performed. This is demonstrated below:

| | | | | | | |
|---|---|---|---|---|---|---|
| | | 1 | 0 | 1 | 1 | |
| | | | | 1 | 0 | × |
| | | 0 | 0 | 0 | 0 | |
| | 1 | 0 | 1 | 1 | | |
| Shifted Sum: | | | 0 | 1 | 0 | (1) |
| Carries: | | 0 | 0 | 0 | 0 | + |
| | 1 | 1 | 0 | **0** | **0** | |
| Saved Sum: | 1 | 0 | 1 | 0 | | |
| Saved Carries: | 0 | 0 | 0 | 1 | | |

Now that all digits have been used, the updated sum and carry words are saved once more. Their addition will produce the upper half of the result. Note that the sum is again shifted once to the left for the reasons explained above:

| | |
|---|---|
| 0101 | (0) |
| 0001 | + |
| **0110** | |

Thus, we now have the whole result: 01100011

### 5.3.2 Bit Multiplier Block

The main building block for implementing digit-serial multipliers, called the Bit Multiplier (BM), is shown in Figure 4. This is a 1-bit multiplier (AND gate) together with a 1-bit full adder. This circuit is used to perform a 1-bit multiplication and accumulate the total (bit) result by adding the carry-in and the sum-in. The outputs are then connected to the next bit multiplier (see below).

### 5.3.3 Digit-serial Multiplier

We are now ready to see how the digit-serial multiplier is implemented. Figure 5 shows a $2 \times 4$ array of bit multipliers designed for unsigned binary multiplication. The way the multiplication is executed reflects the method demonstrated in Section 5.3.1 The bits of the multiplier, B, are passed down the columns of the array, and the bits of the multiplicand, A, are passed across the rows, one digit at a time. Before the first digit of A is supplied, the buffer called LATCHES is reset so that all sum-in and carry-in bits are zero. The reset signal is asserted at the beginning of every new operation.

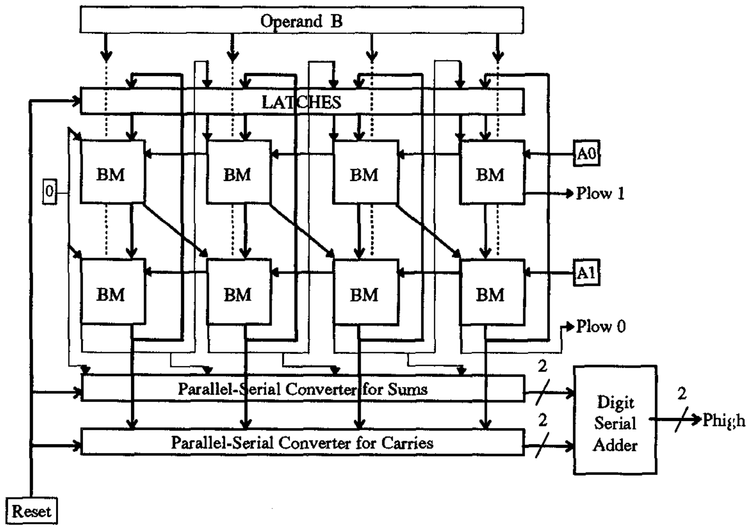
Each bit multiplier in the array multiplies the respective bits, and the result is added to the sum-in and carry-in bits. The outputs of the bit multipliers accumulate the sum and carry bits and pass them down to the next row. In order to shift left each new product, the sum-out bit is passed down diagonally to the right. The carry-out is simply passed down. The 2 least significant digits are produced at the Plow bits, one digit per clock cycle. The 2 most significant digits are the result of the addition of the remaining sum and carry words and are produced at the Phigh bits. By using the parallel/serial converters and a digit serial adder, it takes another 2 cycles to produce them. During that time, however, another multiplication can start. By the time the digit-serial adder completes the first multiplication, the array of bit multiplies produces the two least significant digits of the second result at the Plow bits.

Two last points worth noting in this digit-serial multiplier are the shifting of the sum word to the right for reasons explained in 5.3.1 and the filling of the empty bits with zeroes. At the left of the array, a zero signal fills the leftmost bit of every sum word, since the multiplication is unsigned.

After the digit-serial multiplier was designed and tested, the most difficult part of the implementation was over. In order to transform it into a complete multiplier, however, two 8-bit adders/subtractors and a normalization unit are needed. This is shown if Figure 6. The normalization unit receives the preliminary 48-bit product and produces the 24 most significant non-zero bits. The two adders adjust the exponent accordingly.

## 5.4 Area Requirements

As has been already noted, the main objectives of these designs is minimum chip area together with a reasonable speed. The above implementation of the digit-serial multiplier satisfies both of them. The final version takes 193 logic cells for the matrix array and a total of 490 logic cells for the whole multiplier. The extra hardware is required for the digit-serial overhead and the normalization unit. The former in-



Operand B
LATCHES
BM
0
A0
Plow 1
A1
Plow 0
Parallel-Serial Converter for Sums
Parallel-Serial Converter for Carries
2
Digit Serial Adder
Phigh
Reset

Figure 5: Digit-Serial Multiplier

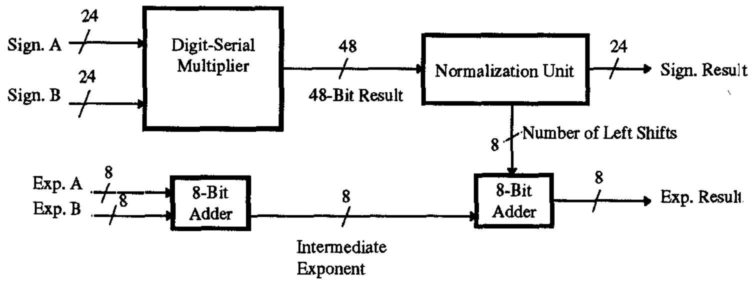


Figure 6: Floating Point Multiplier

cludes the serial-to-parallel and parallel-to-serial registers, the latches, and the FSM required by the digit-serial architecture, and it takes 140 logic cells. The latter takes 170 logic cells. Although this means that the multiplier will take about 49% of an Altera FLEX 81188 FPGA, we can effectively reduce that by having two or more multipliers sharing the same normalization unit; for more details on this, refer to Section 8. It is worth mentioning here that attempts to fit a 24 bit multiplier, implemented in various bit-parallel methods, in this chip were unsuccessful; more than 100% of the chip area was required.

### 5.5 Timing Analysis

Using the MAX+plus II Timing Analyzer tool, it was found that we can clock the multiplier as fast as 15.9 MHz, which translates to one digit output per 63ns. Taking into account that a 24 bit multiplication will produce a 48-bit result, this requires 12 clock cycles for the complete result to be available. The normalization unit requires one clock cycle to produce the 24-bit, normalized significand. The multiplier, however, is easily pipelined, thus giving a 48-bit product every 6 clock cycles. A peak performance rate of 2.3MFlops can then be obtained.

## 6 Connecting the Two Units

In almost every scientific application using floating point arithmetic, the result of an addition will be the operand of a multiplication or vice-versa. For this reason, it is essential that the two units we developed be able to interconnect with each other.

When the result of an addition is used as an operand of a multiplication, there are two cases to consider. First, if the result is to be used as the multiplier, it can be latched into a parallel-serial converter, and the individual digits can be shifted out as needed. Second, if the result is to be used as the multiplicand, it can be directly latch into the `Operand B` register of the multiplier.

In the case that the result of a multiplication is used as an operand of an addition, some extra effort is required. The reason is that the 24-bit multiplication will produce a 48-bit result, and this needs to be truncated down to 24 bits. This truncation is performed by normalizing the product as it is shifted into a serial-to-parallel shift register.

## 7 Area Vs Speed of the Multiplier

One thing we have learned from recent computer history is that today's hard-to-implement designs frequently become tomorrow's commodities. The same holds true for the development of FPGAs. Every new generation of FPGA chips can hold more logic and more on-chip memory than the previous one. So the question might be, why go through all this trouble to design floating point operators that are not fast enough when you can wait until enough area will be available? The answer to this is that we cannot always wait. And even if we wait until we can fit a parallel multiplier on 1 FPGA, it may still be expensive because we may need a lot of them, thus a lot of FPGAs. With the digit-serial approach, however, we can adjust the digit size to get the right balance of performance and cost.

In an effort to demonstrate the problem of the size of a parallel single precision floating point multiplier, Figure 7 shows how the size of a multiplier increases as its matrix size increases from $24 \times 4$ to the fully parallel multiplier with matrix size $24 \times 24$. For comparison purposes, the sizes of 3 members of the FLEX8000 are also shown on the graph.

The most important point to notice is that the parallel multiplier (matrix size of $24 \times 24$) cannot fit on the Altera FLEX 81500, which is the largest FPGA in the FLEX 8000 series. The Altera FLEX 81188, which is the one we have available for testing these designs, cannot even fit the matrix for the parallel multiplier.

In the table shown in Figure 8, similar data is given, but this time, the matrix and multiplier sizes are also given as percentages of the Altera FLEX 81188. Both the matrix size and the multiplier size quoted for the $24 \times 4$ matrix size were obtained from an actual design. The first line in the table is highlighted to emphasize this. For the rest of the entries, the matrix size was obtained from the actual designs, but the full multiplier size was estimated. The factors taken into account for these calculations were the matrix size, the hardware required for the digit-serial architecture, and the hardware required for the normalization unit. This breakdown of logic cell usage is shown in Figure 7.

The normalization unit is always the same, so a constant number of cells is added to the matrix size. As it turns out, the overhead hardware for the digit-serial architecture is also approximately constant so this is also added to the previous result. The important thing to notice in this graph is that the last column, the multiplier with matrix size of $24 \times 24$ does not include the overhead for the digit-serial architecture, since this is a parallel multiplier.

One last point worth making is that the results shown in Figure 7 can be somewhat misleading if the FPGAs to be used are on a prototype board. In that case, the chip pins are pre-assigned and the place-and-route tools have a harder time fitting the designs. Thus, one might think that a multiplier with matrix size of $24 \times 12$ should easily fit in an Altera FLEX 81188 since it takes only 87% of the chip area; this is not the case, however. When the chips have pre-assigned pins, the compiler can usually fit designs that are only 70% of the chip, in which case the largest multiplier would be the one with matrix size of $24 \times 8$.

## 8 Applications

As mentioned elsewhere in this paper, these operators were designed to be used in implementing the gravitational n-body problem on an FPGA prototyping board [6]. N-body methods are used to simulate the evolution and interaction of galaxies, and they are very expensive in terms of computing power. There are various methods to compute the forces between the particles. The one we use is the direct method, which is $O(n^2)$ algorithm. In an effort to provide an alternative solution to customized hardware (such as the GRAPE/HARP approach [2]) which is expensive and non-flexible, we designed a pipelined implementation of the direct method [6]. In this design, we need 6 multipliers and 9 adders.

Since chip area is always our main concern, we tried to find every possible way of further customizing our pipelined design to fit in our prototyping board. Some of these customizations are describe here.

As mentioned in Section 5.5, the multiplier takes 12 clock cycles to produce the result. However, by pipelining the design, we can get a result every 6 cycles, which is a significant improvement in terms of

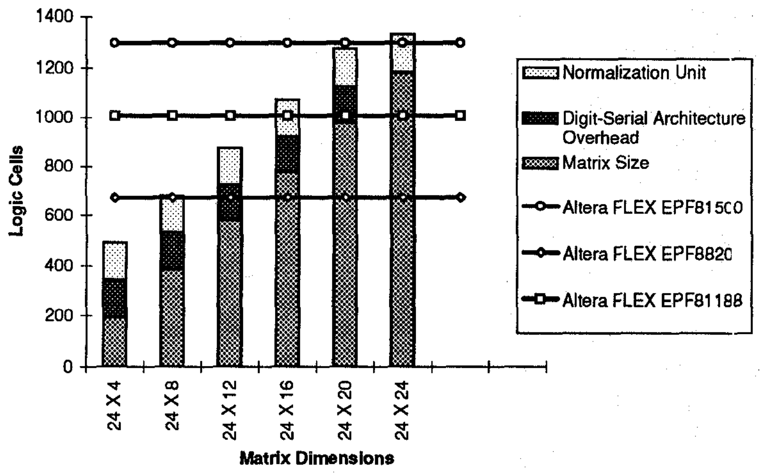


Figure 7: Multiplier size for different matrix sizes. The total number of logic cells is composed of the matrix size, the overhead hardware required by the digit-serial architecture and the hardware for the normalization unit. The capacity of three FPGAs are also indicated for comparison.

| Multiplier Matrix Size | Matrix Size in Logic Cells | Percentage of Altera Flex 81188 | Estimated Multiplier Size in Logic Cells | Percentage of Altera Flex 81188 |
|---|---|---|---|---|
| **24 X 4** | **193** | **18%** | **490** | **49%** |
| 24 X 8 | **383** | **37%** | 680 | 67% |
| 24 X 12 | **578** | **57%** | 875 | 87% |
| 24 X 16 | **776** | **76%** | 1073 | 106% |
| 24 X 20 | **977** | **96%** | 1274 | 126% |
| 24 X 24 | **1181** | **119%** | 1331 | 132% |

Figure 8: Matrix and Multiplier Sizes as a percentage of the Altera FLEX 81188. Bold values indicate actual designs, whereas, the others are estimated.

speed. Also, this 6 clock cycles period means that the normalization unit is used only once every 6 cycles, and sits idle for the other 5. By carefully examining the pipeline table, we found that we can easily share one normalization unit between two multipliers, since they don't need it at the same time. Since the normalization unit is about 15% of an Altera FLEX 81188 chip, this brings the effective size of each of the multipliers down to approximately 42% of the chip. Under different conditions, we could have as many as 6 multipliers sharing the same normalization unit, thus further minimizing the area requirements.

Another customization performed after examining our pipeline was to modify the multiplier into a square unit. Three of the six multipliers used in our design are actually calculating the square of a variable; thus, we saved some hardware by using fewer input pins and registers.

## 9 Future Improvements

The main objectives throughout our work were to minimize the number of logic cells required for the adder and the multiplier, while at the same time keeping the speed of the operations at a reasonable level and maintaining IEEE 32-bit accuracy. The results presented above show that these requirements have been satisfied to a great extent; however, this does not mean that further improvements are not possible. In the rest of this section, we present some of the ideas we have for making our designs faster, smaller, and more accurate.

### 9.1 Faster Multiplication

The digit-serial multiplier presented in Section 5.3.3 uses a digit-serial adder to produce the upper half of the result in order to save chip area. Using this approach, the multiplier takes 12 clock cycles for the complete result to be available (see Section 5.5). An alternative approach would be to use a parallel adder to compute the upper half of the result. In this case, one clock cycle is all that is needed for the addition of the saved carry and sum words; thus, it significantly speeds up the operation. Instead of the 12 cycles, only 7 would be required, 6 for the lower half computed in the array of bit-multipliers and one for the upper half computed in the parallel adder. The parallel adder requires more logic cells but partially compensates by replacing parallel-to-serial latches with ordinary latches.

### 9.2 Improved Accuracy

As mentioned in Section 4.1, the adder does not implement rounding of the result, as defined in the IEEE 754 standard. The main reason for not implementing it is the area limitations of current FPGAs. As part of our plans, proper rounding will be added in the implementations. Thus, depending on the availability of bigger FPGA chips in the future, the guard, round, and dirty bits as well as the shift registers to perform the rounding will be added.

## 10 Conclusions

We have shown that IEEE single precision floating-point arithmetic can be successfully implemented on FPGAs. Our implementations give respectable performance, though not at the level of custom implementations. Peak performances of 7MFlops for addition and 2.3MFlops for multiplication were obtained on our Altera FLEX8000 prototype.

## 11 Acknowledgments

This work has been partially supported by the New Jersey Space Grand Consortium/Associated Institutions for Materials Science under subcontract #NJSGC-94-05. The RIPP-10 FPGA board was provided by courtesy of Altera Corporation. We would like to thank Arvind Mandhani for his help in preparing this paper, and George Schmitt for his valuable comments in revising it.

## References

[1] IEEE Standards Board. "IEEE Standard for Binary Floating-Point Arithmetic". Technical Report ANSI/IEEE Std 754-1985, The Institute of Electrical and Electronics Engineers, New York, 1985.

[2] T. Ebisuzaki et al. "GRAPE Project: An Overview". *Publications of the Astronomical Society of Japan*, 45:361–375, 1993.

[3] Richard I. Hartley and Keshab K. Parhi. *Digit-Serial Computation.* Kluwer Academic Publishers, Boston, MA, 1995.

[4] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach, Second Edition.* Morgan Kaufmann, 1996.

[5] Nabeel Shirazi, Al Walters, and Peter Athanas. "Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines". In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 155–162, April 1995.

[6] Hong-Ryul Kim Todd A. Cook and Loucas Louca. "Hardware Acceleration of N-Body Simulations for Galactic Dynamics". In *SPIE Photonics East Conferences on Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing*, pages 115–126, 1995.