High Performance Applications on Reconfigurable Clusters

by

Zahi Samir Nakad

Thesis Submitted to the Faculty of
Virginia Polytechnic Institute and State University
In partial fulfillment of the requirements of the degree of

Masters of Science

In

Computer Engineering

Dr. Mark T. Jones, Chairman

Dr. Peter M. Athanas

Dr. James R. Armstrong

November 13, 2000
Blacksburg, Virginia

Keywords: Reconfigurable Computing, FIR Filters, Constant Coefficient Multipliers, FPGA

High Performance Applications on Reconfigurable Clusters

Zahi Samir Nakad

Committee Chairman: Dr. Mark Jones
The Bradley Department of Electrical Engineering

Abstract

Many problems faced in the engineering world are computationally intensive. Filtering using FIR (Finite Impulse Response) filters is an example to that. This thesis discusses the implementation of a fast, reconfigurable, and scalable FIR (Finite Impulse Response) digital filter. Constant coefficient multipliers and a Fast FIFO implementation are also discussed in connection with the FIR filter. This filter is used in two of its structures: the direct-form and the lattice structure. The thesis describes several configurations that can be created with the different components available and reports the testing results of these configurations.

To (Samir, Layla, Youssef, Wassim) Nakad

Acknowledgements

This thesis work would have never been accomplished without the valuable guidance and support of Dr. Mark Jones. I am extremely thankful for his patience and guidance throughout my work in the CCM lab, in his classes, and especially in the last phase of editing, reediting, and re-reediting.

I am extremely thankful to Dr. Peter Athanas for his help and guidance. My gratitude goes to him for his technical support and suggestions that made my work a success.

My thanks go out to Dr. James Armstrong for serving on my defense committee and in taking the time to read my thesis.

I would also like to thank all the people in the CCM lab who have taken from their time to offer pointers and support throughout my work. My thanks go out to: Louis Pochet, Luke Scharf , Dennis Collins, Santiago Leon, Ryan Fong, and all the friends I have in the lab.

I am also grateful to Claudia Rente who had the daunting task of editing my grammar in this work.

Last and not least, I want to extremely thank my parents for their invaluable support throughout my academic life that is still going on. Their moral support was immeasurable but living away from them has proved to be extremely hard.

Table Of Contents

List Of Figures

List Of Tables

Chapter 1: Introduction

Many problems faced in the engineering world are computationally intensive. These problems can be solved on general-purpose computers, Application Specific Integrated Circuits (ASICs), or Configurable Computing Machines (CCMs). In order to allow flexibility, a significant amount of the capabilities of the hardware in general-purpose machines is lost, preventing the use of certain implementation optimizations. ASICs can implement all optimizations needed, but implementation is the final and irreversible step; no changes to the design can be made after implementation. CCMs offer a compromise, where all required optimizations can be implemented and the flexibility of change or reconfiguration is possible [1], [2].

1.1 Thesis Contributions

The main contributions of this thesis are based on developing a fast and reconfigurable FIR filter on a commercial configurable computing accelerator. A lack of I/O speed in data entry and acquisition from the WILDFORCE board (discussed in Chapter 2) hinders running designs at higher speeds. A Fast FIFO was created that took advantage of the fast DMA accesses to the memory of the Processing Elements (PE) on the board. By providing new data paths this Fast FIFO allows implemented designs to run at higher speeds. The other difficulty in developing fast FIR filters is the area cost of the multipliers; constant multipliers were introduced in this phase. Because optimization of one of the coefficients helps in optimizing the multipliers, they can offer higher speeds and smaller area consumption.

This thesis compares and contrasts different hardware implementations of a FIR filter. The thesis studies an implementation that allows coefficients of the filter to be changed through the data path, using JHDL multipliers. This contrasts with an implementation that uses constant multipliers but requires full reconfiguration of the filter. These two implementations will also be provided in two structures of the FIR filter: the direct form and the lattice structures. Another comparison point is passing data through the filter using either the regular FIFOs or the Fast FIFO.

1.2 FIR Filters

Filtering using FIR (Finite Impulse Response) digital filters is a computationally intensive problem. If general-purpose computers are used to solve this problem, many of the hardware's capabilities will not be used. However, designing a new ASIC for each response needed is expensive and time consuming. Having a filter implementation where the filter response can be reconfigured on the fly is very useful. FIR filters can be used in many applications such as wireless in-door channel modeling [2]. Another example is the use of the lattice structure of the FIR filter in digital speech processing and in implementing adaptive filters [3].

This thesis studies the implementation of FIR filters on CCMs. The FIR filters discussed are implemented in the direct and lattice form. Two methods of reconfiguring the response of the filters are studied. The first is runtime, using information sent through the data path. The second makes use of constant coefficient multipliers; the response is changed with full synthesis of the design.

1.3 FPGA Boards

FPGAs (Field Programmable Gate Arrays) placed on boards that interface them to PCs, is one type of architecture where the above-mentioned characteristics of CCMs are found. The WILDFORCE and SLAAC1 boards are two such boards. The WILDFORCE board contains five FPGAs, referred to as PEs (Processing Elements). A systolic bus connects four of these PEs (PE1 – PE4). Using the systolic bus, these four PEs can be used in a pipeline-like structure. The SLAAC1 board contains three PEs, which can also be used in a pipeline-like structure. Communication with the outside world, on both boards, is through FIFOs or through PE memory access [4], [5].

The hardware WILDFORCE FIFO speed in communicating data is slow, especially when compared with the DMA mode of accessing memory. This thesis studies a method called the Fast FIFO which makes use of the DMA mode and provides an interface close to that of a FIFO. The Fast FIFO provides a significant increase in the speed of accessing data, at the price of using computational resources (PE0 and PE4) [4].

1.4 Multipliers and Area Consumption

Limited space on FPGAs is a problem that should be addressed in most designs to be implemented. Multipliers consume a large amount of space. Multiplying the input data with coefficients in each tap is required in the FIR filter implementation. These coefficients do not change unless the response of the filter is to be modified. The infrequent change in the constants makes constant multipliers attractive for consideration. Much space can be saved by the use of constant multipliers, because knowing the

coefficients introduces many optimizations to the multiplier. The use of Booth's algorithm and inverting the constant are two optimization techniques examined[6]. The reported results manifest a significant increase in speed and decrease in area consumption.

1.5 Thesis Organization

Chapter 2 of this thesis provides the background material on which the following work is based. Chapter 3 introduces the constant coefficient multipliers that were created and then used in the implementation of the FIR filter. Chapter 4 explains the direct form and lattice implementations of the FIR filter and it also discusses data-path and re-synthesis reconfiguration of the filter. Chapter 5 describes the Fast FIFO implementation and its uses. Chapter 6 describes how all the pieces integrate and reports the results of running the filter under various implementations. Chapter 7 concludes the work and provides future insight on further developments.

Chapter 2: Background

Speech processing, adaptive filtering, and channel modeling are examples of applications where FIR filters are used [2], [3]. Filtering using a FIR filter is a computationally intensive task and typically requires floating point multiplication and addition.

The following section provides the background used in the work to implement a fast reconfigurable FIR filter. Background material on the theory of the FIR filter, the constant coefficient multipliers, the WILDFORCE and the SLAAC-1 boards, and the ACS_API is provided.

2.1 FIR Filters

"Finite impulse response (FIR) filters have been referred to in the literature as moving average filter, transversal filters, and nonrecursive filters. [7]" The main attractive characteristics of this kind of filter are that it is stable and can be made to have an exact linear phase [7], [8]. Coefficient quantization does not affect the response of a filter with linear phase as severely as in other cases; the effect is only in magnitude as opposed to having the effect of the quantization in phase also [3].

Direct-form, cascade-form, frequency-sampling and lattice structures are different forms of implementing the FIR filter in hardware, and are discussed in [3]. Direct-form and lattice are the two structures that are implemented in this thesis. The Direct-form was chosen for its simplicity. This form follows directly from the nonrecursive difference equation (discussed below). This simplicity renders debugging easier because all the intermediate values are easily followed, and the hardware needed to

implement the filter is straightforward. The lattice structure was chosen for its cascading capability; this structure can be viewed as basic units that can be “strung” together without any regard to the number of stages or the position of a certain unit in the whole structure.

As its name implies, the Direct-Form structure follows exactly from the nonrecursive difference equation:

$$y(n)=\sum_{k=0}^{M-1}h(k)x(n-k) \qquad (1)$$

The structure is shown in Figure 1, which shows that there are *M* multiplications and *M-1* additions in this implementation. M is the number of stages; we see that the output of the filter is based on the weighted sum of the last M-1 values in addition to the current value (hence the M multiplications) [3].

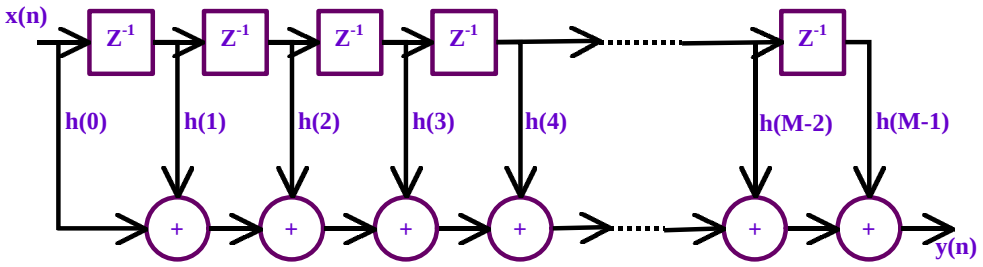

Figure 1 Direct-Form structure of an FIR filter (adapted from Figure7.1 [3])

The lattice filter is also derived from the nonrecursive difference formula, after some manipulation. In this case the response of the m^{th} term, $h_m(0) = 1$, $h_m(k) = \alpha_m(k)$, and k = 1,2,3,…,m. $\alpha_m(0)$ was taken to be 1 for mathematical simplicity. With these changes, the formula for the filter becomes:

$$y(n)=x(n)+\sum_{k=1}^{m}a_m(k)x(n-k) \qquad (2)$$

The coefficient m, that denotes the number of stages in the filter is called the degree of the polynomial $A_m(z)$ representing the filter response; in this case

$$A_m(z) = 1 + \sum_{k=1}^{m} a_m(k) z^{-k} \quad (3)$$

If we consider an FIR filter where m = 1, then

$$y(n) = x(n) + a_1(n)x(n-1) \quad (4)$$

If we call the coefficients of the lattice structure K_m, then the lattice structure filter with the same response will have $K_1 = \alpha_1(1)$. The structure of such a lattice filter is shown in Figure 2 [3].

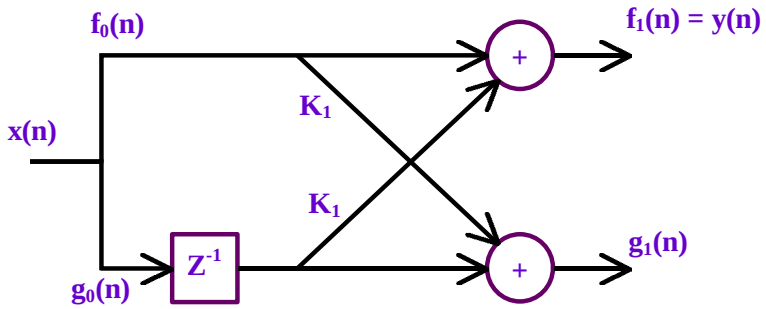

Figure 2 Lattice structure FIR filter with one degree (adapted from Figure7.9 [3])

The above figure shows that $f_1(n)$ corresponds to equation (4) of the FIR filter of degree 1. Considering an FIR filter with m = 2; the response is:

$$y(n) = x(n) + a_2(1)x(n-1) + a_2(2)x(n-2) \quad (5)$$

To get a similar result we cascade two lattice stages to get the structure shown in Figure 3 [3].

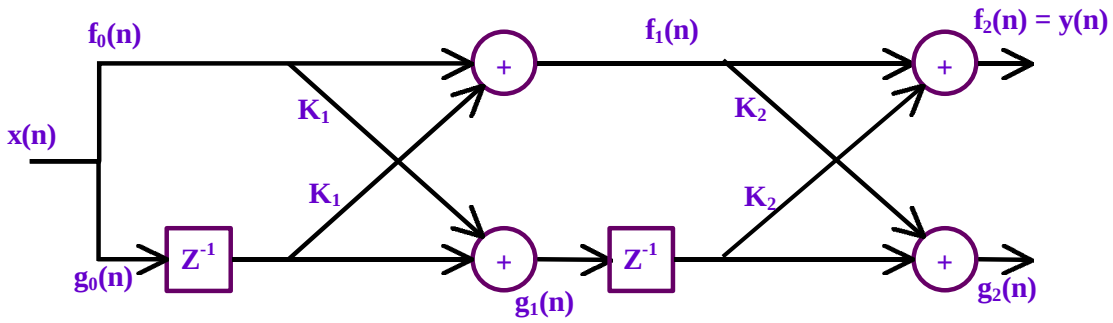

Figure 3 Lattice structure FIR filter with two stages (adapted from Figure 7.10 [3])

From the figure above we can deduce the following formulas:

$$f_1(n) = x(n) + K_1 x(n-1) \tag{6}$$

$$g_1(n) = x(n-1) + K_1 x(n) \tag{7}$$

$$y(n) = f_2(n) = f_1(n) + K_2 g_1(n-1) \tag{8}$$

$$g_2(n) = g_1(n-1) + K_2 f_1(n) \tag{9}$$

Putting equations (6) through (9) together, we end up with the following result:

$$y(n) = x(n) + K_1(1+K_2)x(n-1) + K_2 x(n-2) \tag{10}$$

This equation is the same as (5) with the following relationships between the α and K constants:

$$a_2(2) = K_2 \qquad a_2(1) = K_1(1+K_2) \tag{11}$$

The stages in the lattice structure can just be added until the required degree is established. Equations (8) and (9) can be generalized by substituting m for 2 and m-1 for 1 in the subscripts [3].

.

2.2 Constant Coefficient Multipliers

Implementing multipliers in hardware requires a significant amount of area; an example is the array multiplier (fixed point multiplier) used in the JHDL modgen. The area needed for the unpipelined version depends on the widths of the inputs, x and y, in the following formula: $area = y*(\lfloor x/2 \rfloor+1)$ [9]. This space can be an obstacle in creating certain designs if they do not fit on the chip or silicon that is to be used. If one coefficient that is to be multiplied is known or is going to be fixed for a long time, designing constant multipliers can help in creating hardware that saves in space and time, due to the optimizations gained from knowing one of the inputs [6].

The more 1's present in the binary representation of the coefficient, the greater the number of additions that are needed to get the product. Booth's algorithm reduces the number of additions by replacing any group of consecutive 1's with a subtraction of the term corresponding to the least significant position of the 1's from the term corresponding to the bit position that is one higher than the most significant 1. Figure 4 displays the multiplication of 5 (0101) by 7 (0111), both with and without the use of Booth's algorithm.

Without Booth's algorithm:

```
    0101
    0111
    0101    Term1: corresponds to the 1 in bit position 0 in (0111)
   01010    Term2: corresponds to the 1 in bit position 1 in (0111)
  010100    Term3: corresponds to the 1 in bit position 2 in (0111)
  100011
```

With Booth's algorithm:

```
    0101
    0111
 1111011    Term1: 2's compliment of (0101) corresponding to position 0
 0101000    Term2: corresponds one bit higher than position 2
 0100011
```

Figure 4 Multiplication using Booth's Algorithm

In the multiplication in Figure 4, Booth's algorithm canceled one of the terms that was to be added. The number of ones in the sequence has no significance to Booth's algorithm; more savings are attained with this method if there are more 1's in consecutive positions [6].

Another technique can be derived from Booth's algorithm, based on observing the results of the following example: if we have a series of three consecutive 1's followed by another series of three consecutive 1's, and separated by a 0 (1110111), then using usual multiplication will result in six terms to add. Using Booth's algorithm on each of the series will result in four terms, two terms being created out of each series of consecutive 1's. Of these four terms, two are positive and two are negative. An important observation here is that the term constituting seven 1's (1111111) is really the term (1110111) as mentioned above, with (1000) subtracted from it. Thus, the multiplication with (1110111) can be substituted with a multiplication with (1111111 – 1000). The multiplication with the (1111111) can be solved by Booth's with two terms. Following the discussion above

the whole multiplication can be resolved with the two terms from Booth's and then subtracting (1000). Figure 5 gives an example of multiplying 85 (1010101) with 119 (1110111).

Regular Multiplication	Booth's	New Technique
1010101	1010101	1010101
1110111	1110111	1110111
1010101	11111110101011	11111110101011
10101010	1010101000	10101010000000
101010100	11101010110000	11110101011000
10101010000	10101010000000	10011110000011
101010100000	10011110000011	
1010101000000		
10011110000011		
6 Terms	4 Terms	3 Terms

Figure 5 Multiplication using Booth's Algorithm and the new technique

2.3 Floating Point Format

The floating-point representation used in this thesis is based on the IEEE 754 standard [9]. The 32-bit format, shown in Figure 6, has one bit for the sign (s), eight bits for the exponent (e), and 23 bits for the mantissa (m).

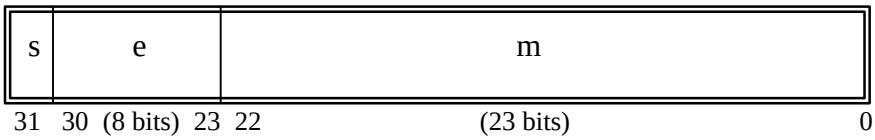

Figure 6 Floating Point Format (32 bits)

The value (v) of the floating-point is:

$$v = -1^{s} 2^{(e-127)} (1.m) \quad (12)$$

There is a bias of 127 for the exponent in this 32-bit format. There is always a 1 at the most significant position of the mantissa. This 1 does not show in the representation; it is implicit. The floating-point number 0x3F800000 will be used as an example. Bit 31 or s is zero in this number so we know that it is positive ($-1^0 = 1$). The exponent e = 0111 1111 or 127, thus the $2^{(e-127)}$ term evaluates to $2^0 = 1$. The term m is all zeroes, so v = 1*1*(1.0) = 1 [1], [2], [10].

2.3.1 Floating Point Multiplication

Multiplication for the floating-point representation is done in separate steps. The first step is to isolate each of the three parts of the representation alone. The resulting sign is the "xoring" of the signs of the two numbers to be multiplied. The exponents are added. The mantissas are multiplied, and the decimal point position is then modified to normalize the result; the value of the exponent sum is fixed to represent the change needed in the normalization [1], [11].

2.3.2 Constant Coefficient Floating Point Multiplication

Floating-point multipliers require a large amount of space when implemented in hardware. The fixed-point multiplier used to multiply the two mantissas is the most expensive part of the floating-point multiplier. The same discussion used for constant coefficient multipliers applies here, if one of the terms is fixed. Thus, using a constant

coefficient multiplier instead of a regular multiplier to multiply the mantissas can attain significant savings in the size of the floating-point multiplier.

2.3 WILDFORCE Board

The WILDFORCE is a product of Annapolis Micro Systems, Inc. This board holds five FPGAs (Field Programmable Gate Arrays) called Processing Elements (PE) and offers data communication between these five FPGAs and a controlling PC, usually referred to as the host [4].

2.3.1 The WILDFORCE Board Architecture

The WILDFORCE board constitutes five PEs, the first one is the control PE called CPE0. The other four are called PE1 up to PE4. Figure 7 shows how these PEs are placed on the board and the communication channels between them.

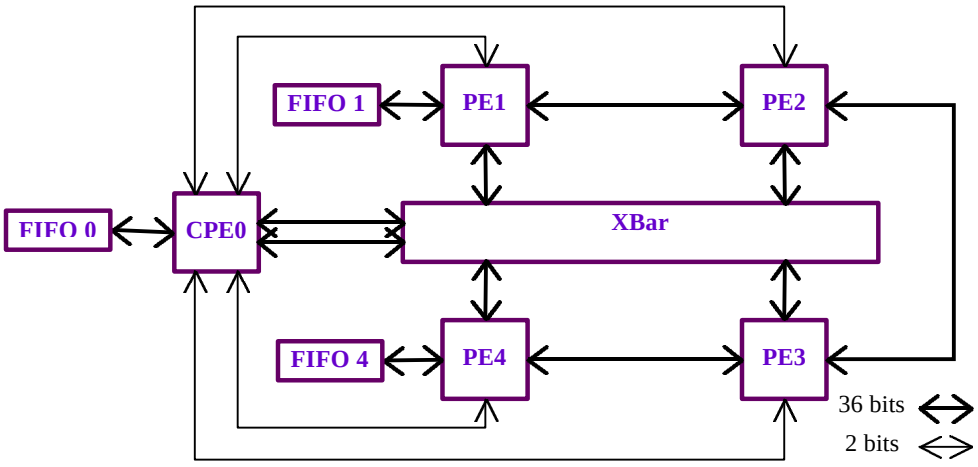

Figure 7 WILDFORCE Board Layout

For CPE0 to carry out its controlling tasks, it is provided with more connections to the communication channels on the board with respect to the other PEs. It has two

interfaces with the crossbar, as opposed to one on the other PEs, and it is connected to each PE with a two-bit control bus. Figure 7 shows the connections that were mentioned with respect to CPE0; it also shows that PE1 to PE4 are connected to each other through a 36 bits wide bus [4], [9].

FIFOs are used to push data to the board and to pull them out. There are 3 FIFOs on the WILDFORCE board and they are connected to CPE0 (FIFO 0), PE1(FIFO 1), and PE4 (FIFO 4). FIFO 1 and FIFO 4 can be used as the end points of a computational path through the systolic bus and through four PEs. The FIFOs offer an easier way of pushing data to the board as compared with using memory. Using the FIFOs also saves on hardware needed to control addressing and capturing the data from the memory.

Each PE has its own separate memory that can be accessed by it and by the host program (discussed in the following subsection). DMA access is also available, this access provides very fast data transfer from the host program to the boards, as opposed to the FIFOs [4].

The crossbar offers connections between all PEs. It can be configured to any setting that is needed. A configuration file sets the configuration of the crossbar, the can also be changed during execution. From Figure 7, it is seen that using this crossbar, any PE can be connected directly to any other PE. The delay inside the crossbar is two clock cycles.

2.3.2 The Host Program

The host program runs on the local machine that controls the WILDFORCE board. This program makes use of the Application Programming Interface (API), which

is provided with the board. Controlling the FIFOs, clock speed, memories, and PEs are examples of the host program's capability through the use of the API [4].

2.4 SLAAC1 Board

The SLAAC1 board was developed by ISI-East [5]. There are several versions of this board, the difference is mainly in the type of the FPGA chips that are found on the board. The SLAAC1-A holds a XC4085XL and two XC40150XL Xilinx chips, while the SLAAC1-V holds three Virtex 1000 Xilinx Chips. The following sections will discuss the board irrespective of which version is being used [5].

2.4.1 SLAAC1 Board Architecture

The SLAAC1 board holds 3 PEs: the control PE (PE 0), PE 1, and PE 2. Figure 8 shows how the PEs are placed on the board and the communication channels between them.

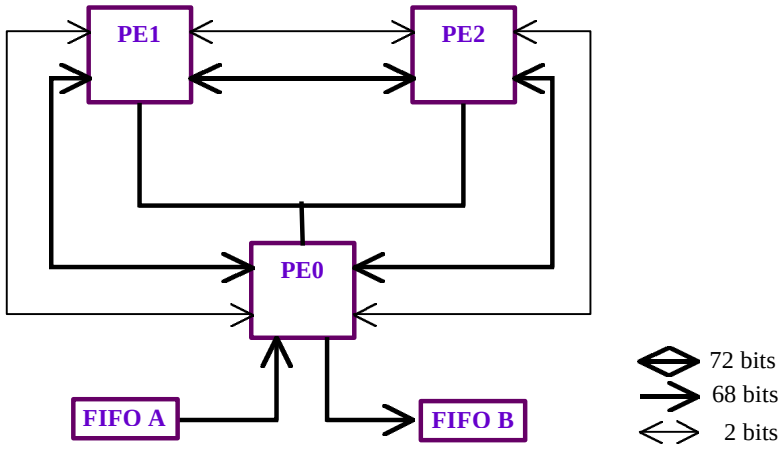

Figure 8 SLAAC1 Board Architecture

PE0 is the control Processing Element connected to the two other PEs through a two bit bus and a 72 bit data bus. This PE also takes care of data communication with the outside world through the FIFOs. FIFO A inputs data to the PE0 and through that to the other PEs, then PE0 takes care of writing to FIFO B, which pushes data to the host. PE0 can also communicate with the PEs through the crossbar. PE1 and PE2 are identical; they are connected to each other through the 72-bit bus, the two-bit bus, and the crossbar.

Each PE in the SLAAC1 has its own memory modules. The host program and the PE access these modules. The crossbar in the SLAAC1 has no configuration file, therefore the user specifies the connections to the crossbar and has to take care of contentions [5], [9].

2.4.2 The Host Program

The host program runs on the local machine, and controls the SLAAC1 board. Using the API provided with the board, the host program controls the FIFOs, clock speed, memories. Thus the API provides the host program and the user with all control needed over the SLAAC1 board [5].

2.5 ACS-API

The Tower of Power (TOP) in Virginia Tech is a good example of a parallel Adaptive Computing System (ACS). This system constitutes sixteen Pentium II machines, each equipped with the WILDFORCE Board. The sixteen machines are connected to each other through an Ethernet and a Myrinet [12] network. Developing designs on systems like the TOP requires more work than implementing on only one

board. After creating the bit files that are to be downloaded on the PEs, the developer has to write network code to connect the boards together. This increase in work can prove to be tedious and time-consuming [13], [14].

The ACS-API offers a way out of creating the network application code. This API offers a high level environment where the user does not have to worry about the network communication code. The user only has to specify a channel between two nodes (a node refers to a board in the case of the TOP). The API also offers calls that will configure the boards, create the connections between them, as well as send and receive data between the boards without intervention by the user. This environment proves to be very helpful in developing designs that need more than one board because the developer will create the hardware needed on one board and then use the API to control as many boards as needed [13].

Another advantage of the ACS-API is that it is independent of the hardware it controls. This is a real help when using different types of boards together, the WILDFORCE and the SLAAC1, for example. The channel object in the environment can act as a buffer for data that is moving from one board to the other; this way the user does not have to worry about creating flow control mechanisms between the boards in the case of a fast sender and a slow receiver.

Chapter 3: Constant Coefficient Multipliers

The following chapter discusses the implementation of the constant coefficient multipliers on both the WILDFORCE [4] and the SLAAC1-V [5] boards. The theory behind this type of multipliers was discussed in Section 2.2. There are two types of multipliers implemented; both use Booth's algorithm [6]. The first type, called the regular constant multiplier, uses Booth's algorithm directly on the 1's in the binary representation of the constant. The second type implemented is based on the New Technique that was introduced in Section 2.2, along with Booth's algorithm. The details of both implementations will be provided along with area and speed results collected, comparing both types with the modgen multiplier of JHDL [9]. The modgen multiplier provided with JHDL, is a hand-crafted solution targeted to Xilinx FPGAs.

3.1 Regular Constant Multiplier

The implementation created for this multiplier is able to support different sizes for the exponent and the mantissa. The area and speed testing that will be shown is based on the IEEE format [10] where the exponent is 8 bits wide and the mantissa is 24 bits wide; only 23 bits of the mantissa are recorded because there is an implicit '1' in the most significant position. Thus, the whole data word is 32 bits wide, counting the bit for the sign.

The floating-point multiplication that is used with IEEE format consists of several steps as discussed in Section 2.3. The most time and area consuming step is the multiplication of the mantissas. This multiplication is the primary focus of this section and also where the regular constant multiplier is used. After determining the binary

representation of the constant to be used as the coefficient in the multiplier, the process begins by checking the positions of the 1's in the representation. When there is a binary 1 surrounded by two 0's (010), it is counted as one term to be used in the overall addition operation. When there are two consecutive 1's, they are counted as two separate terms. A string of more than two consecutive 1's is dealt with using Booth's algorithm resulting in two terms, one of which is positive and the other negative. With this implementation, any group of 1's will end up being at most two terms in the overall addition that provides the product.

The position of any bit that maps to a term determines the number of shifts that are needed so that the terms line up correctly in the addition step [6]. This is depicted in Figure 5. The terms that are derived from the binary representation are added with an adder tree.

The terms that can exploit the benefits of Booth's algorithm are aligned together. Given the nature of Booth's algorithm, the number of subtractors required is half the number of terms that use Booth's algorithm. The other terms are grouped in pairs and added together. The terms that are produced from the Booth's algorithm terms are always positive because the positive term is always larger than the negative term. Thus, there will not be any subtractors in the adder tree at a level above the leaves. After acquiring the product of the mantissas from the adder tree, normalization takes place and the sum of the exponents is fixed to reflect the effect of the normalization. This process will end in the product of the two floating-point numbers.

The structure of the adder tree offers a very good pipelining mechanism. Registers installed between the outputs of a stage and the inputs of the following stage provide one

clock cycle delay of the whole result. The implemented multipliers give the user the option of pipelining, as well as a choice in the number of stages in the pipeline. The fully pipelined case will occur when all the adders have a register at their output. The number of stages of the fully pipelined stage depends on the number of terms to be added. The terms form the leaves of a binary tree; therefore, the depth of the tree will be $\lceil \log_2(NumberOfTerms) \rceil$.

Figure 9 shows an example adder tree of a regular constant multiplier with the constant coefficient set to 183, the binary representation of this number is (10110111). Approaching this representation right to left, the first three 1's make use of Booth's algorithm, the next three 1's end up in three extra terms to be added. The terms are shifted versions of the number that is to be multiplied with the constant coefficient. Figure 9 shows the needed shifts.

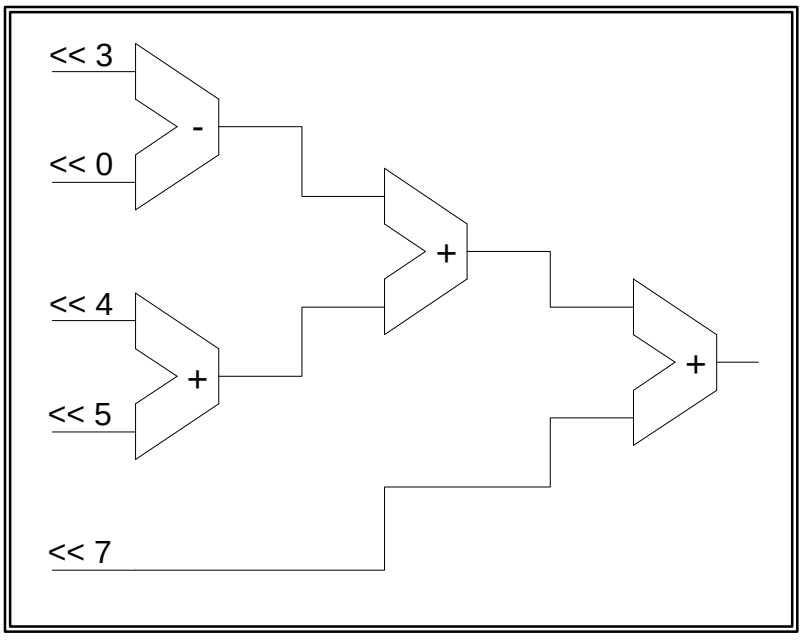

Figure 9 Regular constant multiplier (const = 10110111)

3.2 Conversion Constant Multiplier

The implementation of the conversion constant multiplier offers the capability of specifying the number of bits for the mantissa and the exponent for the floating-point representation. Using the same approach as with the regular constant multipliers, the results of area consumption and speed are reported for the multiplier. This multiplier has 8 bits for the exponent and 24 bits for the mantissa (23 bits are actually saved because of the implicit 1 in the most significant bit), the same as the IEEE format [10].

As discussed above, the most time consuming step in the floating-point multiplication is the multiplication of the mantissas. The following subsection will focus on this multiplication as it is done through the use of the conversion constant multiplier. The discussion to follow is based on the "new technique" which was explained in Section 2.2.

The multiplication is based on the binary representation of the constant coefficient. If the coefficient is being represented with n bits, it can be observed that this coefficient can be recreated by subtracting its 1's complement from the largest number that can be represented with n bits, which is n 1's. As an example, consider the coefficient 183 (10110111) used in the previous section. This coefficient requires at least eight bits to be represented, so consider n to be nine bits. The largest number that can be represented is 511 (111111111), and the 1's complement of 183 (010110111) is 328 (101001000). Then multiplication by the coefficient 183 can be substituted with a multiplication by (511 – 328). Because 511 is all 1's, the use of Booth's algorithm will result in two terms. The same approach used with 183 in the regular constant multiplier is used with 328.

The same rules used in the preceding section regarding the use of Booth's Algorithm and the bit-wise shifts are applied here. The first two terms are the two terms created from applying Booth's on the coefficient 511. A tree of adders is now created based on the 1's in the binary representation of 328 (101001000). There are three terms created from this representation based on the four 1's, since there are no three or more consecutive 1's. The result from the tree of adders will be subtracted from the result obtained from the first subtractor that represents the coefficient 511. The overall multiplier will have two subtractors outside the tree of adders, which will be created of two adders that will take care of summing the three terms. Figure 10 shows the structure of this multiplier with the bit-wise shifts needed at the different inputs.

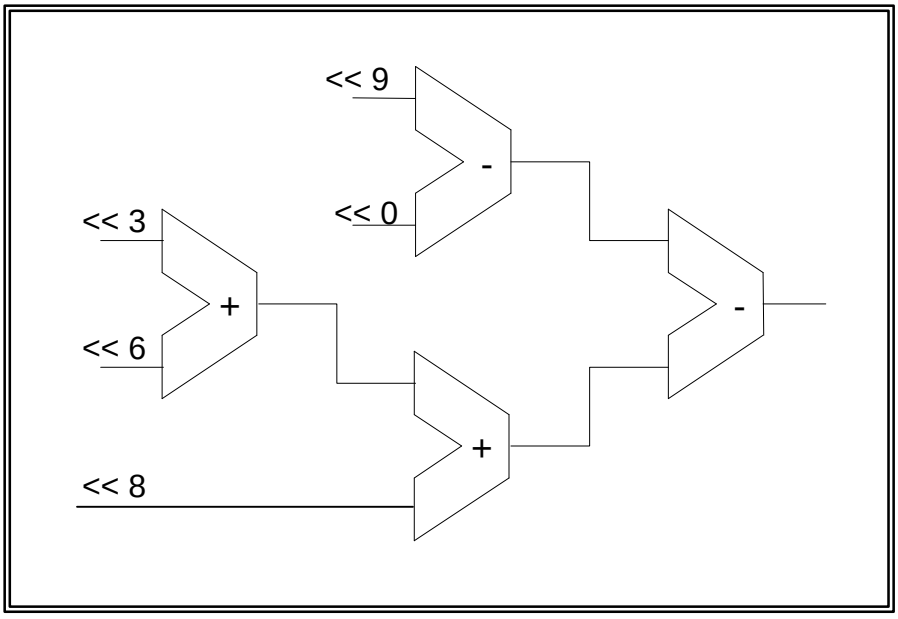

Figure 10 Conversion constant multiplier (const = 10110111)

The structure of the multiplier in this case, as in the regular constant multiplier case, offers a good pipelining mechanism by installing registers between the separate

stages. The implemented multiplier in this case offers same pipelining options as the regular multiplier. The fully pipelined version will be where all the outputs of the adders and subtractors have a register. The number of stages in the fully pipelined case depends on the number of terms to be added. With the conversion constant multiplier there is an extra stage added by the two subtractors needed to fulfill the conversion technique. The two subtractors in Figure 10 are the ones created by first implementing Booth's algorithm on the all 1's number and the second subtractor subtracts the result of the adder tree (based on the 1's complement of the coefficient) from the result of the first subtractor. Thus, in this case, the depth of the stages will be $\lceil \log_2(NumberOfTerms) \rceil + 1$, the NumberOfTerms in this case in the number of terms from the adder tree; that is, the number of terms from the 1's complement of the constant coefficient.

3.3 Implementation Statistics

The results collected after synthesizing and running the multipliers on the WILDFORCE and on the SLAAC1-V boards are provided in the following section. Each set of results provided shows the fastest clock speed at which the multiplier runs along with the hardware resources used, as reported by the synthesis tools. Each set compares the different multipliers (regular constant coefficient, conversion constant coefficient, and the JHDL modgen multiplier) to each other, considering different constants. The three sets considered are: unpipelined and pipelined multipliers implemented on the WILDFORCE Board, and unpipelined and pipelined multipliers implemented on the SLAAC1-V board.

3.3.1 Unpipelined Multipliers Implemented on the WILDFORCE Board

Table1 shows the sizes and the maximum clock speed for the unpipelined multipliers using the constants shown.

The terms in the table will be explained in the following section. The XC4000 signifies the Xilinx parts [15] used on the WILDFORCE board on which these multipliers have been implemented. The sign, exponent, and the mantissa show these values of the constants. The mantissa (binary) shows the binary format of the mantissa along with the implicit one in the most significant bit position.

The highest speed column gives the highest clock cycle frequency of the WILDFORCE board for which the multipliers still give no errors in computing the product. The testing mechanism will be discussed later. The next four columns: CLB usage, FLOPS usage, 4-input LUT, and 3-input LUT are reported directly from the Xilinx synthesis tools. The last column, fixed multiplier # of terms, shows the number of terms that the constant coefficient multipliers have to add; this number is important since it reflects the depth of the tree of adders, discussed in Sections 3.1 and 3.2.

To pick the constants to use in these tables, a function was created to estimate the size that would be needed in the implementation of the multiplier. The constant coefficient multipliers of the same type, regular or conversion, follow the same structure; the difference is in the number and the size of the adders needed to create the adder tree. Accounting for these two factors gives an estimate of the size of each multiplier in comparison with the same type of multiplier, but with a different constant. The function created was used on all the constants that can be created from the 23 bits of the mantissa. These numbers were sorted using Matlab [16], the constant creating the maximum size of

the regular constant coefficient multipler was picked. Then, from the same sorted array the constants whose sizes were in the ¼, ½, and ¾ positions of the array were picked.

The conversion constant coefficient multipliers require more logic to be created, on average, than the regular constant coefficient multipliers. This extra requirement is due to the use of two extra terms, as discussed in Section 3.3. Due to this observation, the tables were created based on the sizes of the regular constant coefficient multipliers. The same constants are used to create the conversion constant coefficient multipliers. For each constant, the multiplier that uses the least area is utilized. To complete the tables, two extra values were added. The first value shows the maximum area that can be used after choosing the smaller of the two types. This constant is the value whose binary representation has no consecutive ones or zeros, and the representation starts and ends with ones, in the case of an odd number of bits in the mantissa. When the conversion technique is used on this value, the result is a representation of no consecutive ones or zeroes, with zeroes at the boundary. Even that the number of ones is less in the use of the conversion case, but the use of the conversion technique requires extra logic that balances out the decrease in the number of ones. The second value has no ones in the mantissa representation; the value used has 128 as the exponent so that the multiplier multiplies by two.

The JHDL modgen multiplier is a multiplier that has two variable inputs, thus the size of the multiplier should be independent of the values that are multiplied. This is true in the case when the two inputs are left as variables, and the reported values of that case are shown at the bottom of the rows assigned for the JHDL modgen multiplier. In the case where one of the inputs to this multiplier is declared as a constant value, the

synthesis tools were able to introduce certain optimizations that reduced the size of the multiplier and increased the speed at which this multiplier could operate.

A function was developed that estimates the size of the constant multipliers for every different constant, using this function the specific coefficients were picked to report the results that are shown below.

The mantissa value of 6141659 is the constant that creates the greatest area consumption, when implemented in hardware, the CLB usage is 397 CLBs, the conversion constant multiplier for the same constant uses 250 CLBs. The JHDL modgen multiplier with two variable inputs uses 368 CLBs, while in the case of a constant input with a mantissa of 6141659, the CLB usage is 352. For these specific numbers, using the conversion constant multiplier is the best choice in saving area. For each constant the smaller between the regular and the conversion constant multipliers should be used to get good saving results.

The maximum operating clock frequency is another important aspect to consider. For the mantissa value of 6141659, the regular constant multiplier can run at 18 MHz, while the conversion constant multiplier can operate at 25 MHz; both these numbers are a significant gain over the operating frequency of the JHDL multiplier which is 9 MHz using two variable inputs and also with 6141659 as the mantissa of a constant input.

The number of terms that are to be added in the constant multipliers can give a good idea about the performance of the regular and the conversion versions of the constant multipliers with respect to each other. The regular constant multiplier has sixteen terms to add while the conversion multiplier has nine terms to add.

Another important constant to consider is the maximum constant multiplier that can be created while choosing the smaller of these two versions. In Table1, the constant that shows this case is 5592405. The area consumption of both versions of the constant multiplier is 330 CLBs (both add 13 terms), while that of the JHDL modgen with the same constant input is 344 CLBs. The savings in area consumption in this case are not significant. The highest speed at which this multiplier can run is 23 MHz using the regular version, while it is 22 MHz using the conversion type. The JHDL modgen runs at a maximum of 10 MHz.

The numbers 6321838, 4255289, and 2097828 map respectively to the values that were picked to represent the ¾, ½, and ¼ positions in the sorted array of the estimated mulitplier sizes. Using the smaller of the two versions in each case still saves on the area used by the JHDL modgen multiplier; there is also a significant increase in the speed at which these multipliers can run.

The constant that maps to multiplying by two, shows that the JHDL modgen multiplier uses the least area, 44 CLBs, while the area used is 54 CLBs for the regular constant multiplier and 149 CLBs for the conversion constant multiplier version. The WILDFORCE board can run the clock at a maximum of 50 MHz; both the regular and the JHDL modgen multipliers run at that speed, while the conversion type runs at 33 MHz.

3.3.2 Pipelined Multipliers Implemented on the WILDFORCE Board

An important feature of these constant coefficient multipliers and of the JHDL modgen multiplier is the ability to be pipelined. Pipelining helps by increasing the value

of the highest rate at which the clock can be run [17]. Introducing delays between the stages of the adder tree pipelines the constant coefficient multipliers, thus useful pipelining is limited by the depth of the adder tree. The modgen JHDL multiplier offers a greater number of pipelining stages. To compare the different multipliers, the regular and the conversion constant multipliers are fully pipelined, and the number of stages created is reported. The JHDL multiplier is then pipelined the number of stages that is typically used in both the previous multipliers. The results for the pipelined multipliers are reported in Table 2.

To create the pipelined version of the constant multipliers, the modgen adders of JHDL were used. The outputs of these adders can be registered without extra area cost. The limitation of these adders is that they only support 32-bit inputs and output. This problem was fixed by truncating every result in the adder tree to 32 bits; the lost accuracy is insignificant as will be discussed later.

The extra column in Table 2 shows the number of pipelined stages. As can be seen from this table, the number of stages typically used is 4, this is the number of stages used in the JHDL modgen multiplier.

In this implementation, the regular constant coefficient multiplier did not break at the speeds that can run on the WILDFORCE board. The conversion constant coefficient multiplier ran at 36 MHz in its worst case. When picking the smaller type for each constant the constant multipliers do not break on the WILDFORCE board. The JHDL multiplier with four pipelined stages did not break, only in the case when multiplying with the constant 2. The other cases operated between 32 and 40 MHz. When the JHDL multiplier was implemented with the two inputs variable, it operated at a maximum clock

rate of 27 MHz. In the reported cases above, the constant multipliers performed better for all case.

X 4000	Sign	Exponent	Mantissa	Mantissa (Binary)	Highest Speed (MHz)	CLB Usage	FLOPS Usage	4-input LUT	3-input LUT	Fixed Multiplier # Of Terms
Regular	0	127	6141659	1.10111011011011011011011	18	397	2	647	25	16
	0	127	5592405	1.10101010101010101010101	23	330	2	521	25	13
	0	127	6321838	1.11000000111011010101110	20	269	2	404	25	10
	0	127	4255289	1.10000001110111000111001	25	252	2	372	25	9
	0	127	2097828	1.01000000000001010100100	35	178	2	231	25	6
	0	128	0	0	>= 50	54	2	54	3	1
Conversion	0	127	6141659	1.10111011011011011011011	25	250	2	367	25	9
	0	127	5592405	1.10101010101010101010101	22	330	2	518	25	13
	0	127	6321838	1.11000000111011010101110	27	250	2	365	25	9
	0	127	4255289	1.10000001110111000111001	26	251	2	367	25	9
	0	127	2097828	1.01000000000001010100100	19	288	2	437	25	11
	0	128	0	0	33	149	2	176	25	4
Jhdl	0	127	6141659	1.10111011011011011011011	9	352	2	618	1	
	0	127	5592405	1.10101010101010101010101	10	344	2	610	3	
	0	127	6321838	1.11000000111011010101110	11	317	2	558	1	
	0	127	4255289	1.10000001110111000111001	9	323	2	570	1	
	0	127	2097828	1.01000000000001010100100	16	250	2	440	3	
	0	128	0	0	>= 50	44	2	80	3	
	Two variables, no constants				9	368	2	635	7	

Table 1 Results of the unpipelined multipliers implemented on the WILDFORCE board

X 4000 (Pipelined)	Sign	Exponent	Mantissa	Mantissa (Binary)	Highest Speed (MHz)	CLB Usage	FLOPS Usage	4-input LUT	3-input LUT	Pipeline Stages
Regular	0	127	6141659	1.10111011011011011011011	>= 50	344	407	475	26	4
	0	127	5592405	1.10101010101010101010101	>= 50	322	396	427	48	4
	0	127	6321838	1.11000000111011010101110	>= 50	261	267	296	40	4
	0	127	4255289	1.10000001110111000111001	>= 50	269	307	295	59	4
	0	127	2097828	1.01000000000001010100100	>= 50	195	180	211	41	3
	0	128	0	0	>= 50	54	2	54	3	0
Conversion	0	127	6141659	1.10111011011011011011011	>= 50	285	237	309	48	4
	0	127	5592405	1.10101010101010101010101	36	368	385	430	63	5
	0	127	6321838	1.11000000111011010101110	>= 50	285	228	299	48	4
	0	127	4255289	1.10000001110111000111001	44	282	226	296	49	4
	0	127	2097828	1.01000000000001010100100	>= 50	339	330	355	73	5
	0	128	0	0	>= 50	181	106	162	52	2
Jhdl	0	127	6141659	1.10111011011011011011011	32	376	216	645	43	4
	0	127	5592405	1.10101010101010101010101	39	372	213	652	43	4
	0	127	6321838	1.11000000111011010101110	39	362	212	621	42	4
	0	127	4255289	1.10000001110111000111001	39	371	210	635	42	4
	0	127	2097828	1.01000000000001010100100	40	342	207	589	42	4
	0	128	0	0	>= 50	239	181	365	40	4
	Two variables, no constants				27	386	223	657	51	4

Table 2 Results of the pipelined multipliers implemented on the WILDFORCE board

3.3.3 Unpipelined Multipliers Implemented on the SLAAC1-V Board

The multipliers implemented on the WILDFORCE board gave a very good result after being pipelined. The next inquiry is, how these multipliers will perform on a Virtex chip. The unpipelined version of both the constant coefficient multipliers and of the JHDL modgen multiplier were implemented and tested on the SLAAC1-V board. This board uses the Virtex 1000 Xilinx chips as its Processing Elements (PEs). [5]

Table 3 shows the results obtained; they are a bit different from the data reported for the WILDFORCE board. The area consumption in this case is measured in slices as opposed to CLBs in the case of the XC4000 PEs. The other hardware consumption data reported is the use of the 4-input LUTs, with the WILDFORCE there was also 3-input LUTs.

The relationship between the area consumption results with respect to the different types of the multipliers is quite close to those reported in the case of the WILDFORCE board. The maximum slice usage of the regular constant coefficient multiplier is 327 for the 6141659 constant; the corresponding conversion multiplier size is 198, while the JHDL multiplier used 305 slices. The largest multiplier that can be created while choosing between the two constant coefficient multipliers (constant = 5592405) uses 253 slices. The JHDL modgen multiplier uses 314 slices when both inputs are variable. When picking the smaller of the two types of constant multipliers, better area consumption in given in all reported cases.

The real significant performance enhancement is in the highest clock speed at which the constant multipliers can run. The lowest speed reported on the constant

multipliers, while choosing the multiplier with the lower area consumption, is 52 MHz. The JHDL multiplier did not provide such a good result; it ran at 14 MHz when both inputs were variable and between 15 and 18 MHz for the other cases, excluding the case when multiplying by two where this multiplier did not break.

3.3.4 Pipelined Multipliers Implemented on the SLAAC1-V Board

The last implementation that will be discussed is that of the pipelined multipliers on the SLAAC1-V board. The area consumption in this case of the constant multipliers is larger than that of the JHDL multipliers because the modgen adders were not supported for the Virtex chips and thus regular adders were used. The registers used for pipelining were mapped to their own slices (without the use of placement tools) and thus more area was consumed. Table 4 shows the results collected, and follows directly from Table 3 on format it also follows the same discussion in Section 3.3.2 on the number of pipelined stages to use with the modgen multiplier.

The results of the area consumption can be compared between the two types of constant multipliers. The largest regular constant multiplier (constant = 6141659) used 848 slices, while the same constant needed 629 slices in the conversion constant multiplier. The largest multiplier that can be created while picking the smaller of the two types (constant = 5592405), consumes 748 slices with the regular multiplier and 835 with the conversion multiplier. The size of the JHDL modgen when both the inputs are variable is 411 slices. It can be seen that the registers did require a lot of space.

The fastest clock speed results still show that the constant multipliers can work at higher speeds. All coefficients used with the regular constant multiplier resulted in logic

that did not break at the speeds at which the SLAAC1-V board can run. The conversion constant multiplier ran at the lowest speed of 80 MHz for the largest multiplier considered. The modgen multiplier had speeds between 57 and 61 MHz, excluding the case of multiplying by 2. The constant multipliers still provided operation at higher speeds even though they used more area on the chip.

The results reported in the four tables show the improvement in area consumption and in high clock speed operation that the constant multipliers offer. This improvement is of use in the case where the multipliers in a design have a fixed coefficient or when the coefficients are constant for a long time.

Virtex	Sign	Exponent	Mantissa	Mantissa (Binary)	Highest Speed (MHz)	Slice Usage	4-input LUT	Fixed Multiplier # Of Terms
Regular	0	127	6141659	1.10111011011011011011011	42	327	502	16
	0	127	5592405	1.10101010101010101010101	50	252	405	13
	0	127	6321838	1.11000000111011010101110	48	242	354	10
	0	127	4255289	1.10000001110111000111001	51	231	321	9
	0	127	2097828	1.01000000000001010100100	68	140	216	6
	0	128	0	0	>= 147	56	87	1
Conversion	0	127	6141659	1.10111011011011011011011	53	198	336	9
	0	127	5592405	1.10101010101010101010101	48	253	435	13
	0	127	6321838	1.11000000111011010101110	55	223	359	9
	0	127	4255289	1.10000001110111000111001	52	229	368	9
	0	127	2097828	1.01000000000001010100100	43	260	413	11
	0	128	0	0	87	140	253	4
Jhdl	0	127	6141659	1.10111011011011011011011	15	305	430	
	0	127	5592405	1.10101010101010101010101	15	305	334	
	0	127	6321838	1.11000000111011010101110	15	293	334	
	0	127	4255289	1.10000001110111000111001	16	305	310	
	0	127	2097828	1.01000000000001010100100	18	281	166	
	0	128	0	0	>=147	195	63	
	Two variables, no constants				14	314	618	

Table 3 Results of the unpipelined multipliers implemented on the SLAAC1-V board

Virtex (Pipelined)	Sign	Exponent	Mantissa	Mantissa (Binary)	Highest Speed (MHz)	Slice Usage	4-input LUT	Pipeline Stages	Fixed Multiplier # Of Terms
Regular	0	127	6141659	1.10111011011011011011011	>=	848	502	4	16
	0	127	5592405	1.10101010101010101010101	>=	748	405	4	13
	0	127	6321838	1.11000000111011010101110	>=	642	354	4	10
	0	127	4255289	1.10000001110111000111001	>=	615	321	4	9
	0	127	2097828	1.01000000000001010100100	>=	448	216	3	6
	0	128	0	0	>= 147	200	87	0	1
Conversion	0	127	6141659	1.10111011011011011011011	87	629	336	4	9
	0	127	5592405	1.10101010101010101010101	80	835	435	5	13
	0	127	6321838	1.11000000111011010101110	84	628	359	4	9
	0	127	4255289	1.10000001110111000111001	84	629	368	4	9
	0	127	2097828	1.01000000000001010100100	87	755	413	5	11
	0	128	0	0	>=	362	253	2	4
Jhdl	0	127	6141659	1.10111011011011011011011	57	394	430	4	
	0	127	5592405	1.10101010101010101010101	58	390	334	4	
	0	127	6321838	1.11000000111011010101110	62	375	334	4	
	0	127	4255289	1.10000001110111000111001	61	384	310	4	
	0	127	2097828	1.01000000000001010100100	61	355	166	4	
	0	128	0	0	>=	325	63	4	
	Two variables, no constants				58	411	618	4	

Table 4 Results of the pipelined multipliers implemented on the SLAAC1-V board

Chapter 4: FIR Filter Implementation

This section describes in detail the FIR filter implementations that were considered, namely the direct-form and the lattice structures [3]. The two implementations mentioned above are reconfigurable. The reconfiguration was accomplished in two ways, through the data path and through re-synthesis and full reconfiguration of the PEs. Reconfiguration using the data path is real-time, where the new coefficients that are to be used in the taps of the filter are introduced into the data path. The tap records the new value and uses it as the multiplying coefficient. The multipliers used are the regular floating-point multipliers provided in the modgen package with JHDL. The drawback to these multipliers is that they are generic, and thus cannot take advantage of the fact that the coefficient in each tap may be fixed for a long time. If the coefficients do not change frequently, then the constant coefficient multipliers discussed in the previous chapter are well suited for the job. These multipliers offer enhancements in computational speeds and in chip area consumption. These multipliers have to be re-synthesized for each new coefficient. Thus, there will be time loss with respect to the generic multipliers when changing the response of the filter. The following subsections will discuss the implementation of each FIR format, irrespective of the multipliers used, and the processes by which to reconfigure the response of the filter when using the different multipliers. A final subsection will discuss the relative merits of each of the four resulting configurations of the FIR filter.

4.1 Direct-Form FIR Filter

As discussed in Section 2.1, the direct-form FIR form follows directly from the non-recursive equation (1) governing the response of the filter. Figure1 shows the structure of this filter. The discussion in the following subsection deals with the implementation of the direct-form, irrespective of the multipliers used, after describing the mechanism by which the constants of the multipliers are set in the case of data-path reconfiguration.

The data path that traverses PE1 through PE4 through the systolic bus on the WILDFORCE board was chosen for the hardware implementation on the FIR filter. Each PE can hold one or two taps, which will relay data to each other through the systolic bus. PE1 will get the input data from FIFO1; after processing this data, it will pass it to PE2. PE4 will get the data after it is processed in the data path; and then outputs it to FIFO4. With this structure, the host code provides data to FIFO1 to be processed and then retrieves the processed data from FIFO4.

Each tap in this structure needs two inputs and two outputs as shown in Figure 11. Time multiplexing was used so that each path will use every other clock cycle on the systolic data bus because the PEs have only one input and one output to the bus.

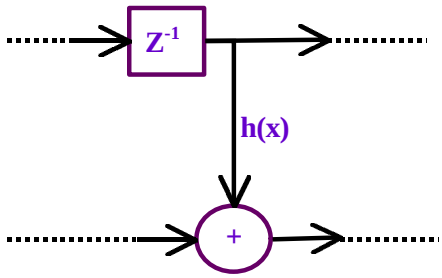

Figure 11 A single tap of the direct-form FIR filter [3]

Control bits were introduced at the top four bits of each word passing through the filter. These bits are used to differentiate between the different control and data words that will pass through the PEs. There are two types of valid data words which map to the two paths that need to be multiplexed into one path. The other control bits are used to either reset the different FSMs (Finite State Machines) used, declare that the words going in are valid or not, or to setup the coefficients in the case of data path reconfiguration. Table1 shows the different combinations of these control bits and what meaning they encode.

Control Bits	30	29	28	Description
	V	R	P	
	0	0	0	Invalid Data Mode
	0	0	1	Invalid Program Mode
	0	1	0	
	0	1	1	Reset Signal
	1	0	0	Valid Mode (Original Data)
	1	0	1	Valid Program Mode
	1	1	0	Valid Data Mode (Accumulation Data)
	1	1	1	Reset Signal

Table 5 Control bits used in the filter design

Bit 31 is tied directly to the *fifoempty* flag of PE1which is considered as another valid bit. Thus, whenever the FIFO is empty the words are considered invalid. This bit is also tied to the *fifoalmostfull* flag of PE4; this flag is sent back from PE4 to PE1 through the crossbar and gives the control needed so that no data is overwritten in PE4's FIFO. The bits that are shown in Table1 are V-valid, R-Reset, and P-Program.

Figure 12 shows the structure implemented on the PEs in the case of data-path reconfiguration. RegC takes care of saving the value of the coefficient for the multiplier and in resetting this value when needed. When the values inside the RegCs of the

different taps are to be changed, the Reset word is first sent through the board. This signal will reset the FSM (Finite State Machine) inside RegC to the initial case. The RegCs now start saving the new coefficients that are passed to them, each RegC uses the first valid coefficient it gets and then invalidates it. Thus, the first RegC will get the first coefficient and so on.

Figure 13 shows the same implementation using the constant coefficient multipliers, so there is no need for RegC to hold the coefficient.

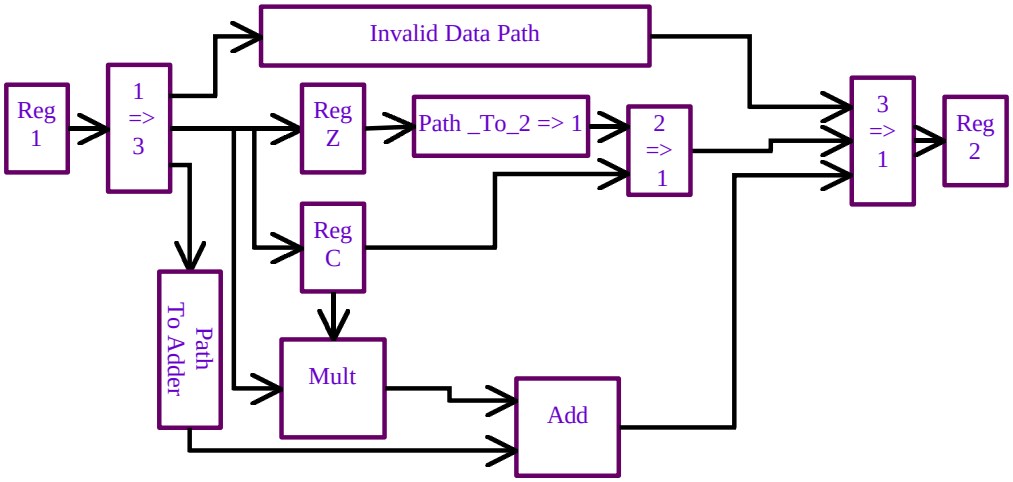

Figure 12 Implemented structure of the direct-form FIR filter with regular multipliers

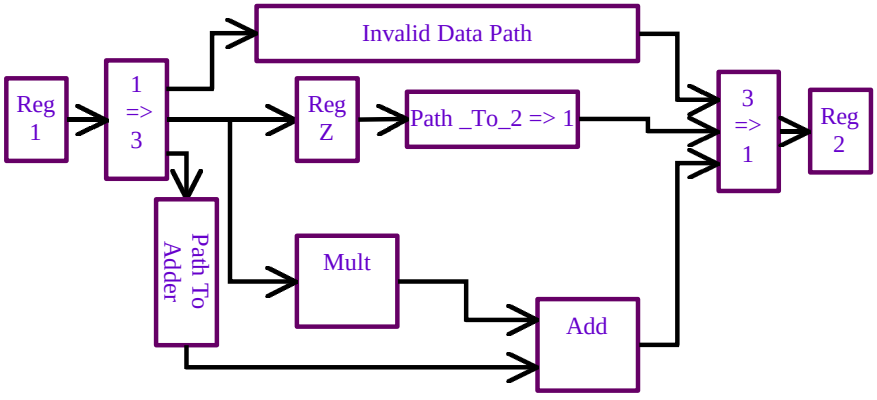

Figure 13 Implemented structure of the direct-form FIR Filter with constant coefficient multpiliers

The valid data mode words that go through the filter should always be passed as Original data (from Table1) first followed by Accumulation data. This order is important so that the intermediate accumulation result is attributed to the correct Original data. The "1 => 3" and the "3 => 1" modules are responsible for keeping the order of the two types of valid data.

4.2 Lattice FIR Filter Structure

The response of the lattice structure of the FIR filter was discussed in Chapter 2. Equations (6 & 7) describe the response of the first tap of the lattice structure, Equations (8 & 9) for the second stage follow in the same manner, with the following taps following a similar pattern. The hardware implementation of this type of filters is discussed next. Using the same approach as Section 4.1, the discussion will focus on the overall implementation, irrespective of the multipliers used, and then the differences between the two approaches will be pointed out.

The process used in setting the coefficients when using the regular multipliers is the same that was used with the direct-form implementation. The same data path used in the direct-form filter is used with the lattice structure. The input data to the filter comes through FIFO1 to PE1, then the data traverses the systolic bus through the four PEs and the data is sent to the host program through FIFO4. In this structure each PE can hold an arbitrary number of taps of the FIR filter, as long as the design still fits on the chip.

From Figures 2 and 3 it can be seen that there are two input and two output paths on each tap of the lattice structure; the same approach in the direct-form is used, where the two paths are multiplexed into one and the control bits are used to distinguish

between them. The two paths are the “f” data and the “g” data that are shown in Figures 2 and 3 [3], thus each of these paths will use every other clock cycle on the systolic bus.

The control bits are used to distinguish the different data types and indicate the validity of the data. These bits identify control words that deal with setting the coefficients when using the JHDL multipliers and resetting the filter, and the two types of data words that are used to represent the “f” and the “g” paths. The control bits are also used to declare if the words are valid or not. The following table shows the different combinations of these bits and their encoded meaning.

Control Bits	30	29	28	Description
	V	R	P	
	0	0	0	Invalid Data Mode
	0	0	1	Invalid Program Mode
	0	1	0	
	0	1	1	Reset Signal
	1	0	0	Valid Mode ("f" Data)
	1	0	1	Valid Program Mode
	1	1	0	Valid Data Mode ("g" Data)
	1	1	1	Reset Signal

Table 6 Control bits used in the lattice structure FIR filter

As in the direct-form implementation, bit 31 is used to also declare if the values are valid or not by being tied directly to the *fifoempty* flag of FIFO1. When the FIFO is empty, the data being read is invalid. This bit is also tied to the *fifoalmostfull* flag of PE4 through the crossbar; this signal is needed so that no data is overwritten inside FIFO4. Also following from the previous structure, the bits are V for the valid control signal, R for the Reset signal, and P for distinguishing between program data and regular data.

The forms of lattice structure implementation with the regular multipliers and the constant coefficient multipliers are shown in Figures 14 and 15 respectively. The

difference comes from the hardware that was used to store the coefficients for the JHDL multipliers.

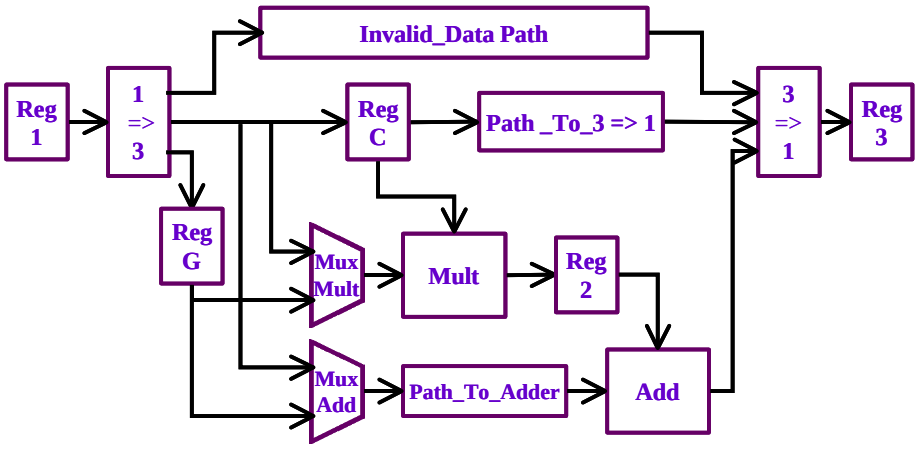

Figure 14 Implemented structure for the lattice structure with regular multipliers

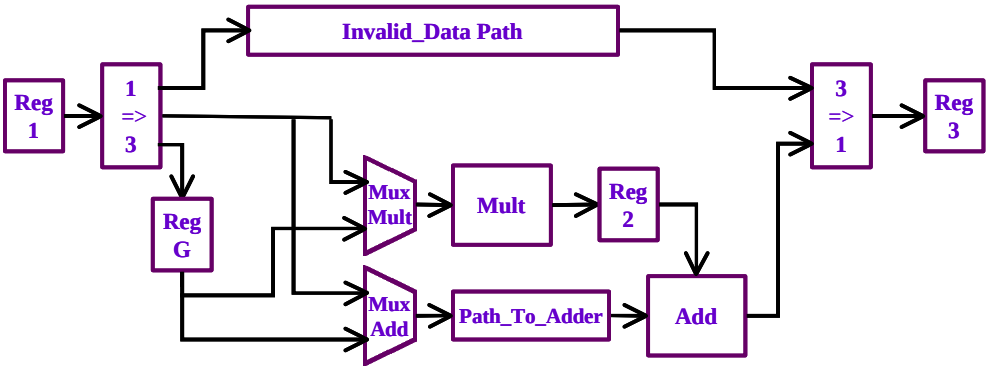

Figure 15 Implemented structure for the lattice structure with constant coefficient multipliers

From the lattice structure of the FIR filter that is shown in Figures 2 and 3, we see that both the “f” and the “g” path are multiplied by the same coefficient. In the implementation that is used on the PEs, the two data paths “f” and “g” will be multiplexed. With the correct control of the data that is entering the PE, the “f” and “g” values can be multiplied and added using the same multiplier and adder, thus saving a significant amount of area on the chip.

The individual modules in this implementation follow exactly from their counterparts in the direct-form structure except for the RegG, Mux Mult, and Mux Add. The “Mux Mult” and the “Mux Add” are two multiplexers that take care of sending the “f” and “g” data at the right time to the multiplier and adder, so that the output is the same as the one expected of the lattice-structure, while using only one multiplier and one adder.

The “f” and the “g” data should be kept in the correct sequence, in the same manner that the Original and the Accumulated data had to be kept in sequence in the direct-form implementation.

4.3 Implementation Results

There are four different possible implementations that can be created of the FIR filter. These four implementations rise from first choosing which form to use, the direct-form or the lattice structure. The second step is picking either the JHDL modgen multipliers or the constant coefficient multipliers. Another parameter to consider is the number of filter taps in each PE.

When using the JHDL modgen multipliers, the two inputs to the multipliers are variable. This helps the FIR implementation (both the direct-form and the lattice structure) to be real-time reconfigurable. The process of this reconfiguration was discussed in Subsection 4.1 and 4.2, the reconfiguration process needs to send a reset signal followed by the coefficients that are to be used as the new constants of the filter. The time needed for that depends on the depth of the filter and on the number of pipelined stages used, thus it is the time needed for the values to pass through the logic of the filter. Another observation is that the new values to be filtered by the new filter

configuration need not be delayed after the coefficients are sent, so the delay is only the number of the values sent to reconfigure (each value needs one clock cycle to get to the input of the filter).

The constant multipliers offer faster operating clock rates and smaller area, but the run-time reconfigurability is out of the question. To change the response of the filter the whole multipliers should be changed, synthesis will be needed after the code for the multipliers is changed. The process of changing the coefficients and resynthesizing is time consuming; the synthesis of four PEs for one board can take up to an hour with the conventional methods. The use of JBits [18] can help in this time consumption, the reconfiguration of one PE on the WILDFORCE will approximately require 190 ms, changing the response for the three PEs would take 3 * 190 ms. Throughput for each period T that includes a response change will be (T – 3*190) * maximum clock speed. When using the JHDL multipliers for the same time period, setting the response only takes three clock cycles which is negligible, so the throughput will be T* maximum clock speed with the JHDL multipliers. The larger throughput for each case of time of operation and maximum clock speed determines which multipliers to use with the FIR filter needed.

The JHDL modgen multipliers used in the FIR filter have two variable inputs and thus the size of these multipliers is not dependent on the values at the inputs. The area consumption of these filters and the maximum operating clock speed are reported in Table 7.

	One Tap Per PE		Two Taps Per PE		Four Taps Per PE
	Area Consumption	Clock Speed	Area Consumption	Clock Speed	Area Consumption
Direct-Form JHDL Multipliers	1097	24	2231	20	4226
Direct-Form Constant Multipliers	1055	27	2119	20	4083
Lattice Structure JHDL Multipliers	1104	20	2207	17	4224
Lattice Structure Constant Multipliers	1037	24	2074	18	4096

Table 7 Implementation results for the different FIR filters

Reporting the area consumption of the FIR filters using the constant multipliers is more complex. The size of the multipliers is dependent on the constant to be used for the multiplication. Another issue is that for each constant there are two versions of the constant multiplier (regular and conversion); they give the same result, but one version needs less area for each different constant. The reported values will be based on the constant that creates the average size of the minimum of both types of constant multipliers. After determining the sizes needed for each constant for both types, one of the values that create the median of the minimum is 26810 (1.0000110100010111010). After using this value in the implementation of the FIR filters, the area consumption results are reported in Table 7.

In all implementations of the FIR filter, the JHDL multipliers were pipelined at 10 stages and the constant multipliers were fully pipelined. Then, an extra delay was added to get to the 10 stages of the JHDL multipliers (this delay was not added while testing for area). With this setup, the speed at which the filters break is not limited by the multipliers. Table 7 reports the maximum speeds at which each of the filter implementations was able to run.

The implementations using the constant multipliers have a better clock speed performance since there is less logic in the implementation. The performance of the

multipliers was of no effect because the multipliers were shown to work at higher speeds with no errors, while being pipelined with fewer stages, results of Subsection 3.3.

The constant multipliers proved to use less area and to run at higher speeds, reported results in Subsection 3.3. When these multipliers replace the JHDL modgen multipliers on the FIR filters, the area savings are not significant because of all the logic in the design.

Chapter 5: Fast FIFO

Data access on the WILDFORCE board can be accomplished by the use of the hardware FIFOs on PE0, PE1, and PE4 [4]. These FIFOs offer a method to get data on and off the board with flow-control. Consuming the CPU and having a slow data transfer rate are the two primary problems of the FIFOs on the WILDFORCE board. The WILDFORCE board offers the ability to access the memories of the PEs from the host program with and without DMA (Direct Memory Access). Using the memory as the main data interaction point adds to the complexity of the hardware, because addressing and memory bus handling must be integrated into the FPGA design. The Fast FIFO was developed to offer the same simple interface as the regular FIFOs along with the speed that can be attained by using DMA to the memories. The following sections will discuss the implementation and the results obtained from running the Fast FIFO [4].

5.1 Fast FIFO Concept

Data access to the memories of the PEs is faster than data access to the FIFOs, this fact is the main motivation behind creating the Fast FIFOs. To improve the speed that can be achieved on the data path from FIFO1 through the systolic bus to FIFO 4; the Fast FIFO uses the memory of PE0 as the primary temporary storage of data that is to be passed through the systolic bus. PE0's job is to keep track of the size of the data chunks written to its memory and their position. When PE1 asks for data by asserting a control bit, PE0 supplies data through the crossbar. PE1 gets data as if it were coming from a regular FIFO; it does not have to worry about addressing or memory access. The data is processed in PE1, PE2, and PE3. PE4 in this case has to temporarily store the data in the

area that it has (inside the PE and not in PE4's memory). When the storage space is full, PE4 stops PE0 from sending more data, and it moves the data stored to the memory and informs the host program of the size of the data in the memory.

5.2 The Host Program

The host program controls sending data to the board and retrieving it after it has been processed in the PEs. Passing data through the board using the Fast FIFO is done in blocks of 0 to 256 words. The host program moves the blocks of data through DMA access to the memory of PE0. The host code sends a "control" word to the regular FIFOs (FIFO 1) to notify PE0 of the new data block. The regular FIFOs were picked because the PE can check the *fifoempty* flag and determine if any new data is present. The "control" word that is written carries information about the size of the block and its position in PE0's memory. Each "control" word gives information about one block of data.

Retrieving the data is done in a similar manner. When PE4 has transferred a block to its memory, it notifies the host program by sending a regular FIFO word that contains the size of the block that has been written. PE4 writes the blocks to its memory consecutively, thus the host program should keep track of where it read last from PE4's memory. The number of words in FIFO 4 reflects the number of blocks in PE4's memory. With this information, the host program will be able to retrieve all processed data.

5.3 PE0 Implementation

The job of PE0 is to read the data written to its memory by the host program. The whole process to be done in PE0 is controlled by a Finite State Machine (FSM). PE0 checks the *fifoempty* flag of its regular FIFO, looking for new blocks written to its memory. Every word in the regular FIFO maps to a block that was placed in PE0's memory in a location and of the size shown in the FIFO word itself.

When PE0 finds that there is data for it to retrieve, it reads the amount of data written at the specified location. This data will be pushed to PE1 through the crossbar; PE1 has the capability to ask for data, thus if PE0 does not have an asserted flag that PE1 needs data, it will stop pushing data and stop reading from its memory.

The first job of the FSM is to read the "control" word from the regular FIFO. The size of the block is extracted from the control word and sent to PE4. This information is needed to control the temporary storage (to be discussed in the following subsection). PE0 can then start reading from the memory; there are two branches here, PE0 either has to read a single word or more than that. If a single word is to be read, it is read and pushed to PE1. PE0 waits to be "stalled" by PE4, announcing that the temporary storage in PE4 is full. PE0 waits again to be un-stalled, signaling that PE4 has finished writing from the temporary storage to its memory. The cycle starts again with PE0 checking the regular FIFO for new words. If PE0 is to read more than one word from memory, the basic process is the same, except that PE0 is waiting for multiple words instead of a single word and thus different clock cycles are to be controlled.

5.4 PE4 Implementation

PE4 is responsible for grabbing the data that is passed to it from PE3, and then passing this data to memory and informing the host program of the size of the data that was received.

The process that takes place in PE4 is controlled by two FSMs. The first step in the process is recording the size of the block to be sent. PE0 sends the size of the block to be read from its memory; PE4 stores this word. The implementation in PE4 contains a RAM that can hold up to 256 words for temporary storage. Using this RAM, PE4 starts storing the data that it receives through the data path (usually through PE3). This process is controlled by the first FSM (Write_1), which takes care of writing the correct amount of data into the temporary storage. After all data specified by PE0 are received by PE4, the FSM sends a Stall signal to PE0 that all data has been stored. This FSM signals the second FSM (Write_2) to start its work. Write_2 is responsible for sending the data in temporary storage to the memory of PE4. When all data is transferred, PE4 writes a word into the regular FIFO that contains information about the size of the block that is written in the memory. In this way, the host program knows that new data has been written, and can retrieve it. After the writing phase in PE4 is finished, it sends an “unstall” signal to PE0, so that the cycle can start again.

5.5 Testing Results

The Fast FIFO was compared to the regular FIFOs by sending the same amount of data through each and timing the process. In the case of the Fast FIFO, the host code dumped 256 blocks of 256 words each into the memory of PE0 and signaled PE0 by

sending control words to the regular FIFOs. The host code read the same values back from PE4's memory after reading the control words from FIFO4; this process was repeated 10,000 times to decrease the significance of the error caused by the granularity of the timers. The same amount of data was passed through the regular FIFOs; the host code dumped the data into FIFO1 in chunks of 512 words each and then retrieved the data from FIFO4.

Timing of these two transfers is done in two ways. The first method used was timing all the 10,000 runs so that the entire transfer was tested. The reading function is not called until FIFO 4 was almost full to save on time because reading and writing to the regular FIFOs were blocking calls.

Overall Timing	Time Required (sec)	Throuput (Mbytes/s)
Regular FIFOs	362.1673	6.90288
Fast FIFOs	118.5461	21.0888

Table 8 Timing results for data passing through the FIFOs

Processor Timing	Overall Time	Processor Time	Percentage
Regular FIFOs	394.7339	376.117	95.28%
Fast FIFOs	120.6576	47.71392	39.54%

Table 9 Processor time consumption by the FIFOs

Reading and writing to the FIFOs is done through the DMA FIFO read and write, so that less time is needed, as opposed to using the regular FIFO read and write. The results of the timings are reported in Table 8. the Fast FIFO has a throughput 3.055 times larger. The second timing method is aimed at testing the amount of work the processor is using in the actual reading and writing operations as opposed to the total time elapsed

during the process. To accomplish this timing, only the write and read functions are timed when they are called. The extra functions needed in timing slow the whole process by a fraction, thus the overall timing and the timing of the reading and writing will be reported in Table 9. The reported percentages show that using the regular FIFOs will render the processor unable to do other tasks, because most of the operating time is consumed by writing and reading. In the case of the Fast FIFO, the write and read functions require a much smaller fraction of the overall time and thus the processor will be able to have other tasks running while data is passed through the board.

Chapter 6: Integration and Testing Results

Developing a fast, scalable, and reconfigurable FIR filter implementation is the goal of this thesis. This implementation can be created through the integration of the various techniques mentioned in the previous chapters.

The different choices available for the FIR implementation create four different filter configurations. The filter can be either direct-form or lattice structure and it can use either the JHDL modgen multipliers or the constant coefficient multipliers. It can also have a tap or two on each PE of the WILDFORCE board (or more on new, larger FPGAs). The use of the constant multipliers creates a simpler logic implementation, as well as a higher operating frequency than the modgen multipliers.

The data input to the filter can be provided by the regular FIFOs that are provided by the WILDFORCE board, or through the use of the Fast FIFO. Operating the regular FIFOs in the DMA mode, and having only two registers in each PE to pass the data from the input to the output, had a transfer rate of 6.90288 Mbytes/sec as reported in subsection 5.5. The same implementation used with the Fast FIFO, had a transfer rate of 21.0888 Mbytes/sec. Thus, having the Fast FIFO provide the input to the filter can improve the speed of the entire operation.

The direct-form FIR filter was implemented with the constant coefficient multipliers and run under both the regular FIFOs and the Fast FIFO. The throughput through the use of the regular FIFOs was 5.89658 Mbytes/sec and through using the Fast FIFOs it was 19.287 Mbytes/sec. We can see that there is a loss of data rate from what

was reported before. This decrease is attributed to the pipeline stages in the adder and multiplier; these stages will use clock cycles to fill up before output data can be collected.

To create the best-case scenario, the Fast FIFO was used with the direct-form FIR filter with constant multipliers while implementing two taps in every PE. The throughput of this implementation is 17.865 Mbytes/s. Every two output words represent one output of the FIR filter (Original and Accumulation data). Each output of the FIR filter requires a multiplication and an addition, thus two floating-point operations in every tap. There are two floating-point operations every two clock cycles, or this can be reported as one floating-point operation every clock cycle. The throughput in words/sec is 17.865 Mbytes/s / 4 = 4.46625 Mwords/s. Each words goes through six taps, thus there are six floating point operations on each word every clock cycle; the overall output is 26.7975 MFLOPS.

The previous paragraphs showed how the throughput of the filter can be increased by using the constant multipliers and through the use of the Fast FIFO to input data and collect results at the output of the filter. The scalability issue is addressed by running the filters on multiple boards. The ACS_API is used to load the FIR filter implementation on several boards, and then to “tie up” these boards together to form a larger filter. The ACS_API can make use of the regular FIFOs and the Fast FIFO as means of input and output from the specific boards, then it relays the data between the boards.

	Throughput	
Number of Boards	**Mbytes/sec**	**MFLOPS**
1 (without ACS_API)	17.865	26.7975
1	6.838	10.257
4	0.2798	1.6792
7	0.113	1.1866

Table 10 Results of implementation on multiple boards

A decrease in the total throughput of the system is observed with Table 10. Sub-optimal performance as measured by the total throughput of the system is observed for implementations on multiple boards. The abstraction layers of the ACS_API and the communication channels between the boards are the main source of this overhead. The ACS_API is still under construction; proper tuning of the implementation of the communication channels would significantly reduce the communication overhead. The best-case projected results of such an implementation are shown in Table 11.

	Throughput	
Number of Boards	**Mbytes/sec**	**MFLOPS**
1 (without ACS_API)	17.865	26.7975
1	6.838	10.257
4	6.838	41.028
7	6.838	71.799

Table 11 Results of the optimistic implementation on multiple boards

The SLAAC1-V board with the three Virtex chips offers a greater amount of resources and speed of operation as opposed to the WILDFORCE board. An implementation of the FIR filter on such a board would result in a much larger MFLOP count because of the faster operation and the extra number of taps that can be implemented in each PE. There are 2304 CLBs on the X400 chips on the WILDFORCE board, while there are 12,288 slices on the Virtex 1000 chips on the SLAAC1-V;

approximately the Virtex chips can hold designs six times larger than the X4000 chips; then 12 taps of the FIR filter should fit on one chip. If the WILDFORCE boards in the Tower Of Power were to be substituted with SLAAC1-V boards will make implementing a 576 (16 boards * 3 chips * 12 taps) tap FIR filter possible.

In the best-case operation, the PCI bus can be considered the limiting factor to the clock speed that the board can be used at. The SLAAC1-V has a 72-bit bus that runs at 66 MHz, this bus takes care of input and output, then this bus can provide 32 bits of input data at every clock cycle. Every PE can hold 12 taps, so there are 12 floating-point operations every clock cycle in each PE; this amounts to 792M floating-point operations (66M*12) on every PE per second. The SLAAC1-V holds three PEs; one of these PEs (X0) holds the PCI core, thus the board will be considered to hold two and a half PEs to implement the FIR filter, then there will be 792 * 2.5 = 1980 MFLOPS on each SLAAC1-V. In a best-case situation the ACS_API is considered to have negligible cost, then the SLAAC1-V Tower Of Power will reach 16 boards * 1980 MFLOPS (31680 MFLOPS) with this implementation.

The SLAAC1-V offers a 72-bit bus interface to the PEs, this bus can be used to input the Original and Accumulated data at the same time, then there will be no need to multiplex these two paths into the same bus any more; thus, area consumption on the PEs can be lowered due to removing the logic that took care of multiplexing.

Chapter 7: Conclusions

The goal of this thesis is stated as developing a fast, scalable, and reconfigurable FIR filter implementation. Constant multipliers were created to increase the speed of operation and reduce the area used on the chip. The Fast FIFO was created to increase the speed of data communication between the board and the host program. These two implementation optimizations helped in reaching the 26.7579 MFLOPS that was recorded in running the optimized direct-form FIR filter. Reconfiguration of the filter response was implemented in two ways, either by sending the filter coefficients through the data path or through reconfiguration of the multipliers, when using the constant multipliers. The design is scalable in the sense that it can be implemented on multiple boards while using the ACS_API to connect the boards together.

The Fast FIFO increased the data rate between the Host Program and the board with a factor of three. The Fast FIFO was also able to reduce the processor time needed during a transfer, the regular FIFOs required 95.28% of the processors time while the Fast FIFO only needed 39.54% of that time. The constant multipliers operated at a two fold higher speed than their JHDL counterparts, and required less area on the chip.

One of the most primary limitations of the design is area consumption. The maximum number of taps that can be placed on a PE is two taps. The full implementation of the FIR filter was tested on the WILDFORCE board. The Virtex chips on the SLAAC1-V board can accommodate larger designs than the X4000 Xilinx chips on the WILDFORCE board. Implementing the FIR filters on the SLAAC1-V can give better results in terms of speed and number of computations.

In the operation of the Fast FIFO, PE0 sends the size of the block to PE4, the word that carries the size has to pass through the logic of PE1, PE2, and PE3. This requires that the designs on these three PEs recognize that this control word should be considered invalid. A better implementation would not require these PEs to worry about such control words.

The constant multipliers use the modgen adders when there is a need for pipelining. The modgen adders can register their output without losing area on the chip. These adders do not operate for number representations larger than 32 bits and they are not mapped to work on the Virtex chips. Through the use of placement tools the regular adders can have a register at their output without extra area cost, implementing the constant multipliers with these tools will be of great benefit for the use of these multipliers on the Virtex chips.

Bibliography

[1] K. Ramachandran, *Unstructured Finite Element Computations on Configurable Computers.* Masters Thesis, Virginia Polytechnic Institute and State University, 1998.

[2] A. L. Walters, *A Scalable FIR Filter Implementation Using 32-bit Floating-Point Complex Arithmetic on a FPGA Based Custom Computing Platform*. Masters Thesis, Virginia Polytechnic Institute and State University, 1998.

[3] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing.* Prentice Hall 1996.

[4] Annapolis Micro Systems, Inc., WILDFORCE Reference Manual, 1997, 1998.

[5] Information Sciences Institute – East, SLAAC Project Page, http://www.east.isi.edu/projects/SLAAC

[6] B. Parhami, *Computer Arithmetic: Algorithms And Hardware Designs*. Oxford University Press, 2000.

[7] N. K. Bose, *Digital Filters: Theory and Applications*. Elsevier Science Publishing, 1985.

[8] A. V. Oppenheim and R. W. Schafer, *Digital Signal Processing.* Prentice Hall, 1975.

[9] Brigham Young University's JHDL WWW Site, http://www.jhdl.org

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

[11] N. Shirazi, A. Walters, and P. Athanas, "*Quantative Analysis of Floating Point Arithmetic on FPGA Custom Computing Machines,"* IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, California, April 1995.

[12] Myricom High-Speed Computers and Communications, http://www.Myri.com

[13] K. Yao, *Implementing an Application Programming Interface for Distributed Adaptive Computing Systems*. Masters thesis, Virginia Polytechnic Institute and State University, 2000

[14] M. Jones, L. Scharf, J. Scott, C. Twaddle, M. Yaconis, K. Yao, P. Athanas, and B. Schott, *"Implementing an API for Distributed Adaptive Computing Systems,"* Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, CA, April, 1999.

[15] Xilinx, The Home Page For Programmable Logic, http://www.Xilinx.com

[16] The MathWorks home, Matlab 5.3.1 http://www.mathworks.com\products\matlab

[17] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, 1993.

[18] Steve Guccione, Delon Levi, and Prasama Sundararajan, "*JBits: Java based interface for reconfigurable computing."* Second Annual Military and Aerospace Applications of Programmable Devices and Technologies (MAPLD'99), The Johns Hopkins University, Laurel, Maryland, Sep 1999.

Vita

Zahi S. Nakad

Zahi Nakad was born in December, 1976 in a small town called Jdita in the Bekaa Valley, Lebanon. After graduating from his high school, he was accepted to the Computer and Communication Engineering program in the American University of Beirut. After graduating with distinction in 1998, Zahi joined the Electrical Engineering program in Virginia Tech as a graduate computer engineering student. In the spring of 1999, Zahi joined the Configurable Computing Lab under Dr Mark Jones. He received his Master's degree in 2000 and is now pursuing a PhD degree in Electrical Engineering at Virginia Tech.