NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

IMPLEMENTATION OF A HIGH-SPEED NUMERIC FUNCTION GENERATOR ON A COTS RECONFIGURABLE COMPUTER

by

Thomas J. Mack

March 2007

Thesis Co-Advisors: Jon T. Butler
Herschel H. Loomis

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2007	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE Implementation of a High-Speed Numeric Function Generator on a COTS Reconfigurable Computer		**5. FUNDING NUMBERS** N/A	
6. AUTHOR(S) Thomas J. Mack			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Security Agency Fort Meade, MD		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited		**12b. DISTRIBUTION CODE** A	

13. ABSTRACT (maximum 200 words)

Certain methods of realizing numeric functions, such as *sin(x)* or $\sqrt{x}$, in hardware involve a Taylor Series expansion or the CORDIC algorithm. These methods, while precise, are iterative and slow and may take on the order of hundreds to thousands of CPU clock cycles.

A faster method involves a piecewise approximation to the function. The function value is computed by reading pre-calculated coefficients (slope and intercept for first order approximations). And then, by multiplying the function argument by the proper slope and adding the proper intercept, a close approximation to the function solution is produced.

This thesis shows how this first order approximation technique was implemented on an FPGA-based COTS reconfigurable computer. MATLAB routines were developed to approximate the function as a set of consecutive, linear equations. The MATLAB approximation is combined with other modules designed in VHDL to construct an overall circuit.

A pipelined circuit was created on the SRC-6E computer that reduces the latency of the *sin(πx)* function by over 88% and produces a result on each clock cycle. The circuit easily implements other functions by simply exchanging the approximation and coefficients. Thus, a user-friendly environment was created for calculating functions at higher speeds than the more popular current methods.

14. SUBJECT TERMS Numerical Function Generator, Piecewise Linear Approximation, Field Programmable Gate Array (FPGA), Reconfigurable Computer			**15. NUMBER OF PAGES** 143
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

IMPLEMENTATION OF A HIGH-SPEED NUMERIC FUNCTION GENERATOR ON A COTS RECONFIGURABLE COMPUTER

Thomas J. Mack
Lieutenant Commander, United States Navy
B.S.E.E., University of San Diego, 1997

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 2007

Author: Thomas J. Mack

Approved by: Professor Jon T. Butler
Thesis Co-Advisor

Professor Herschel H. Loomis
Thesis Co-Advisor

Professor Jeffrey B. Knorr
Chairman, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Certain methods of realizing numeric functions, such as *sin(x)* or $\sqrt{x}$, in hardware involve a Taylor Series expansion or the CORDIC algorithm. These methods, while precise, are iterative and slow and may take on the order of hundreds to thousands of CPU clock cycles.

A faster method involves a piecewise approximation to the function. The function value is computed by reading pre-calculated coefficients (slope and intercept for first order approximations). And then, by multiplying the function argument by the proper slope and adding the proper intercept, a close approximation to the function solution is produced.

This thesis shows how this first order approximation technique was implemented on an FPGA-based COTS reconfigurable computer. MATLAB routines were developed to approximate the function as a set of consecutive, linear equations. The MATLAB approximation is combined with other modules designed in VHDL to construct an overall circuit.

A pipelined circuit was created on the SRC-6E computer that reduces the latency of the *sin(πx)* function by over 88% and produces a result on each clock cycle. The circuit easily implements other functions by simply exchanging the approximation and coefficients. Thus, a user-friendly environment was created for calculating functions at higher speeds than the more popular current methods.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION....1
A. CENTRAL PROBLEM AND PURPOSE....1
B. IMPLEMENTATION OVERVIEW....2
C. THESIS ORGANIZATION....4

II. FUNCTION APPROXIMATION....5
A. SEGMENTATION....5
1. Non-Uniform Approximation Algorithm....11
2. Uniform Approximation Algorithm....12
B. MATLAB RESULTS....15
C. SUMMARY....15

III. NFG CIRCUIT....17
A. CIRCUIT OVERVIEW....17
B. CIRCUIT COMPONENTS....18
1. Slope and Intercept Look-up....18
2. Multiplier....19
3. Adder....19
4. Number System....19
C. SUMMARY....20

IV. SRC IMPLEMENTATION....21
A. SOFTWARE CODE....21
1. main.c....21
2. <subroutine>.mc....22
3. makefile....22
4. Macros....22
a. info....23
b. blk.v....23
c. HDL Files....23
B. SUMMARY....24

V. IMPLEMENTATION RESULTS....25
A. PC C++ IMPLEMENTATION....25
B. SRC IMPLEMENTATION....25
1. SRC C Code Implementation....27
a. Use of C Library Functions....27
b. If, Then, Else Implementation (Floating Point)....28
c. If, Then, Else Implementation (Fixed Point)....29
2. SRC Macro VHDL....29
C. ASIC IMPLEMENTATION....33
D. FPGA RESOURCES....33
E. COMPUTATION RESULTS....34
F. SOURCES OF ERROR....35

1. Function Approximation 36
2. Conversion from Decimal to Binary in MATLAB and Excel 36
3. Absence of Rounding in the Multiplier and Adder 36
4. Insufficient Bits 37

VI. CONCLUSION 39
A. SUMMARY OF WORK 39
B. SUGGESTED FUTURE WORK 41
1. Multiple NFGs in Parallel 41
2. 16-bit and Higher Implementations 42
3. Memory Vice If/Then 42
4. Uniform Approximation Increases to Next Power of 2 43
5. Higher-Order Approximations 44
6. Different Architecture, $y = c_1(x-p) + c_0$ Circuit 44
7. Use of *Remez* Algorithm for Segmentation 44
8. Rounding Vice Truncation 45

APPENDIX A. MATLAB ALGORITHMS 47
A. LINEAR APPROXIMATION USING POLYFIT 47
B. MULTIPLE LINE APPROXIMATION 56
C. NON-UNIFORM LINEAR APPROXIMATION 57
D. UNIFORM LINEAR APPROXIMATION 58
E. UNIFORM LINEAR APPROXIMATION WITH ERROR BOUNDS 58
F. FIXED-POINT DECIMAL TO BINARY 60

APPENDIX B. VHDL SOFTWARE CODE 63
A. NFG TOP-LEVEL VHDL CODE 63
B. SLOPE AND INTERCEPT LOOK-UP CODE 70
C. MULTIPLIER CODE 71
D. ADDER CODE 71

APPENDIX C. SRC COMPUTER VHDL MACRO IMPLEMENTATION CODE 73
A. C IMPLEMENTATION CODE USING MATH LIBRARY 73
1. Main.c 73
2. Sin.mc 75
B. C IMPLEMENTATION CODE USING IF, THEN, ELSE 76
1. Floating Point 76
a. Main.c 76
b. Sin.mc 78
2. Fixed Point 79
a. Main.c 79
b. Sin.mc 81
C. SRC VHDL MACRO IMPLEMENTATION CODE 83
1. SRC C Coding 83
a. Main.c 83
b. Sine.mc 84
c. Makefile 85
2. SRC Macro Files 88

a. Info *88*
b. Blk.v *89*

APPENDIX D. PC C++ CODE **91**

APPENDIX E. LESSONS LEARNED **95**
A. FILE NAMING PROBLEMS **95**
B. USING THE `CONST` CONSTRUCT IN C **95**
C. INCORRECT ARGUMENTS IN SYSTEM SUPPLIED MACROS **96**
D. IF / THEN / ELSE LIMITATION **97**
E. MULTIPLE FILES USED IN A MACRO **97**
F. XILINX / SYNPLIFY INCONSISTENCIES **97**
G. MODELSIM AND MULTIPLE HDL'S **98**
H. INITIALIZING MEMORY FROM A SEPARATE FILE **98**
I. MACRO LATENCY AND SRC OVERHEAD **101**

APPENDIX F. SRC OUTPUT **103**

APPENDIX G. SYNPLICITY AREA REPORT **107**

LIST OF REFERENCES **119**

INITIAL DISTRIBUTION LIST **121**

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

Figure 1. Numeric Function Generator (NFG) Architecture (After Ref. 1). 3
Figure 2. Linear Approximation Function User-Interface. 6
Figure 3. MATLAB Segmentation Output. 8
Figure 4. Graphical Representation of Function and Approximation. 9
Figure 5. Close-up of Approximation. 10
Figure 6. Error Across Approximation 11
Figure 7. MATLAB Output for Uniform Approximation. 13
Figure 8. NFG Top-Level Schematic. 17
Figure 9. Types of Macros and their Characteristics (Ref. 2). 31
Figure 10. Multiple NFGs Working in Parallel. 41

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1. Functions of Interest and Their Domains....4
Table 2. Number of Segments for Non-Uniform and Uniform Segmentation for 8 and 16-bit Precision. 16
Table 3. Summary of Implementations. 32
Table 4. Summary of FPGA Resources Used. 33
Table 5. Difference vs Bit Equivalency 34
Table 6. Examples of Output Results....35

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

ASIC	Application Specific Integrated Circuit
BDD	Binary Decision Diagram
BRAM	Block Random Access Memory
BUA	Basic Unit of Accuracy
CORDIC	Coordinate Rotation Digital Computer
CPU	Central Processing Unit
DSP	Digital Signal Processing
ECS	Engineering Capture System
EVBDD	Edge-Valued Binary Decision Diagram
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IEEE	Institute of Electrical and Electronics Engineers
ISE	Integrated Software Environment
ITE	If, Then, Else
LSB	Least Significant Bit
LUT	Look-Up Table
MAP	Multi-adaptive Processor
MHz	Megahertz
MSB	Most Significant Bit
MS	Microsoft
NFG	Numeric Function Generator
NPS	Naval Postgraduate School
OBM	On-Board Memory
PC	Personal Computer
RAM	Random Access Memory
RMS	Root Mean Square
ROM	Read Only Memory
SRC	Seymour R Cray
USN	United States Navy
Verilog	A C-Based HDL

VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

ACKNOWLEDGMENTS

I would like to thank my wife, Kary and my boys, Joey and Billy for their endless support and love. They are my motivation.

I would also like to thank my advisors, Prof. Jon Butler and Prof Herschel Loomis. Their courses sparked my interest and increased my knowledge in the field of digital engineering. Their support on this thesis was invaluable.

Also, thank you to the National Security Agency (NSA) for their financial support; SRC Computers, Inc. for their technical support and the U.S. Navy for giving me this great educational opportunity.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

The need for high-speed numeric computation is greater now than it has ever been. Applications such as digital signal processing, graphics rendering and scientific calculations require the evaluation to numeric functions (e.g. *sin(x)* or *log(x)*) to be done quickly and repeatedly. Current methods of producing these solutions may involve a Taylor Series expansion or the CORDIC algorithm. These methods, while precise, are iterative and slow and may take on the order of hundreds to thousands of clock cycles.

This thesis shows that the evaluation of numeric functions can be done much quicker by using a unique architecture. This architecture realizes *f(x)* as a piecewise approximation, $f(x) \approx c_1x+c_0$. The function solution is realized by storing pre-calculated coefficients, (slope, c_1 and intercept, c_0) and then accessing these coefficients from memory, when needed. By multiplying the function argument, x, by the proper slope, c_1, and adding the proper intercept, c_0, a close approximation to the function solution can be produced. Additionally, this process can go beyond just simple functions to more complicated calculations, as needed.

It is shown how this first order approximation architecture was implemented on an FPGA-based COTS reconfigurable computer. A pipelined circuit was created that reduces the latency of the $sin(\pi x)$ function by over 88% and produces a result on each clock cycle. The circuit easily implements other function by simply exchanging the approximation and coefficients. Thus, a user-friendly environment was created for calculating functions at higher speeds than the more popular current methods.

The process includes several steps using three software packages and two pieces of hardware. Initially, the desired function approximation is generated on a PC using MATLAB. MATLAB determines the piecewise linear approximation, and generates the VHDL code to be used by the circuit for coefficient look-up. The overall general circuit was initially built using Schematic Capture on the Xilinx ISE software package. The Xilinx and MATLAB generated VHDL code is transferred to the SRC-6E reconfigurable computer where it is combined with SRC-specific C code to create a macro that is

capable of performing the function calculations. From this point, different functions can be implemented by simply replacing the coefficient look-up VHDL code.

I. INTRODUCTION

A. CENTRAL PROBLEM AND PURPOSE

The need for high-speed numeric computation is greater now than it has ever been. Applications such as digital signal processing, graphics rendering and scientific calculations require the evaluation of numeric functions (e.g. *sin(x)* or *log(x)*) to be done quickly and repeatedly. Certain methods of computing these functions involve a Taylor Series expansion or the CORDIC algorithm [1]. These methods, while precise, are iterative and slow and may take on the order of hundreds to thousands of clock cycles.

Sasao, Butler and Riedel [2] have shown that numeric functions can be produced much more quickly by using a unique architecture. This architecture realizes $f(x)$ as a piecewise approximation, $f(x) \approx c_1x+c_0$. The function solution is realized by storing pre-calculated coefficients, (slope, c_1 and intercept, c_0) and then accessing these coefficients from memory, when needed. By multiplying the function argument, x, by the proper slope, c_1, and adding the proper intercept, c_0, a close approximation to the function solution can be produced.

Ref. [2] also discusses a method of using a look-up table (LUT) cascade in order to determine which segment a particular input value, x, corresponds. This is known as segment indexing. Nagayama, Sasao and Butler [3] show a recursive segmentation algorithm for dividing a function over its range and an alternate LUT cascade method based upon the edge-valued binary decision diagram (EVBDD).

Frenzen, Sasao and Butler [4] discuss the relationship between the amount of memory needed and desired error constraints for three approximation methods. Ref. [5] shows how to design a LUT tree circuit for segment indexing, without the need for the designer to refer to a binary decision diagram (BDD). The characteristics of NFGs based-upon second-order approximations are analyzed in Ref. [6]. Cao, Wei and Cheng [7] discuss three different hardware algorithms that use second-order approximation. Their method uses floating point numbers, vice fixed-point numbers.

The purpose of this thesis is to put theory into practice by demonstrating how the first order approximation architecture can be produced with a user-friendly interface and implemented on an FPGA-based COTS reconfigurable computer. It will then be shown that this architecture produces results with less latency and a shorter average time (for long blocks of calculations) than current methods.

B. IMPLEMENTATION OVERVIEW

The implementation process includes several steps using three software packages and two pieces of hardware. Initially, the desired function approximation is generated on a PC using MATLAB [8]. MATLAB generates the VHDL code to be used by the circuit for coefficient look-up. The general circuit is built using Schematic Capture on the Xilinx Integrated Software Environment (ISE) software package. The subsequent Xilinx and MATLAB generated VHDL code is transferred to the SRC-6E reconfigurable computer where it is combined with SRC-specific C code to create a macro that is capable of performing the function calculations. The SRC synthesizer will physically configure the circuit on the FPGA. From this point, different functions can be implemented by simply replacing the coefficient look-up VHDL code and re-synthesizing the circuit.

In this thesis, all approximations are first order, and so the segments take the form: $f(x) = c_1 x + c_0$ where c_1 is the slope of the line and c_0 is the intercept of the line. The architecture of the numeric function generator (NFG) is shown in Figure 1. The independent variable, *x,* is shown at the top.

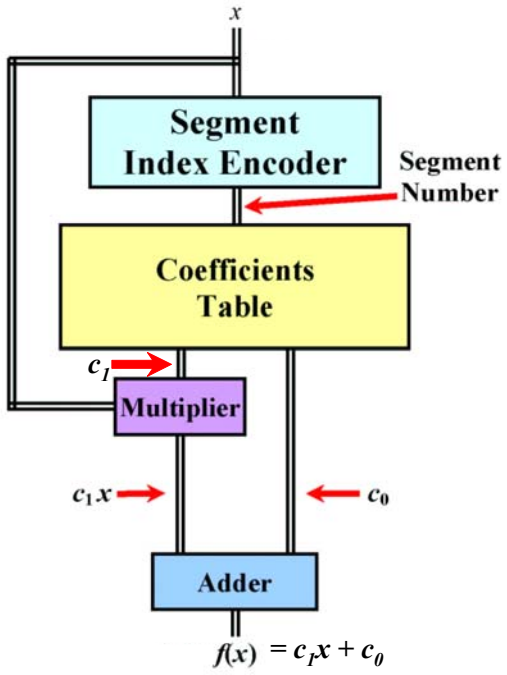

Figure 1. Numeric Function Generator (NFG) Architecture (After Ref. 1).

Since the function is typically approximated by many adjoining straight-line segments, it is necessary to determine what segment is being used for the given value of x. This is done by the Segment Index Encoder, which produces a segment number associated with the value of x. After determining the appropriate segment, the coefficients, c_1 and c_0 are looked up in memory. These coefficient values are then used by the rest of the circuit to compute an approximation to $f(x)$. The resultant value is the solution of the function.

Function	**Interval**	
f(x)	***x***	***f(x)***
2^x	[0,1)	[1,2)
$1/x$	[1,2)	(½,1]
$\sqrt{x}$	[0,2)	$[0, \sqrt{2})$

$1/\sqrt{x}$	[1,2)	$(1/\sqrt{2}, 1]$
$log_2(x)$	[1,2)	[0,1)
$ln\ x$	[1,2)	[0,ln2)
$sin(\pi x)$	[0,½)	[0,1)
$cos(\pi x)$	[0,½)	[1,0)
$tan(\pi x)$	[0,¼)	[0,1)
$\sqrt{-\ln(x)}$	$[1/256, ¼)$	$(\sqrt{-\ln(1/4)}, \sqrt{-\ln(1/256)}]$
$tan^2(\pi x)+1$	[0,¼)	[1,2)
$-(x\ log_2 x + (1-x)\ log_2(1-x))$	(0,1)	
$\frac{1}{1+e^{-x}}$	[0,1)	$[½, \frac{1}{1+e^{-1}})$
$\frac{1}{\sqrt{2\pi}} e^{\frac{-x^2}{2}}$	$[0, \sqrt{2}]$	$[\frac{1}{\sqrt{2\pi}}, \frac{1}{\sqrt{2\pi e^1}}]$

Table 1. Functions of Interest and Their Domains.

As identified in Ref [2], the specific functions of interest and the associated domains of their input values are above in Table 1.

The rest of this thesis will discuss the process that is followed to implement the NFG on the FPGA. In addition, results will be discussed and compared.

C. THESIS ORGANIZATION

Chapter II discusses the approximation of the function and the calculation of the coefficients by MATLAB. Chapter III discusses the coding of the NFG in VHDL. Chapter IV explains the construction of the NFG circuit on the SRC's FPGA. Chapter V discusses implementation results. Chapter VI provides a brief summary and suggestions for future work.

II. FUNCTION APPROXIMATION

The key to the whole NFG algorithm is the *function approximation*. The NFG uses the approximation to calculate the value of the function at any given point (in actuality, the output of the NFG is the value of the approximation, and not that of the actual function, but within error limits). The approximation must be pre-calculated; by dividing up the function into multiple, adjacent linear segments. The segment characteristics (e.g. endpoints, slope and intercept) are then used by the NFG for its calculations. Therefore, the NFG's accuracy depends upon the accuracy of the approximation. The error can be made as small as desired by reducing the segment size. The tradeoff is memory size; as segment size decreases, more segments and more memory is needed.

A. SEGMENTATION

The approximation is generated with the aid of user-specified parameters by MATLAB (see Appendix A for the MATLAB M-files).

The M-file, *LinAppxPfit.m* (short-hand notation for Linear Approximation using MATLAB's routine "Polyfit") is the "master" routine that calls the subordinate subroutines (the other M-files in Appendix A) as needed to perform its approximation. Routine *LinAppxPfit.m* shows the user a set of options that allows MATLAB to perform the approximation to fit the user's needs. The question and answer format is designed to make the process as user-friendly as possible, little knowledge of MATLAB is required. The MATLAB interface looks like this:

```
********************************************************************************

     LINEAR APPROXIMATION OF A FUNCTION USING POLYFIT with INTERCEPT SHIFTING
                              [DEFAULT in BRACKETS]

Input the Function, func[sin(pi*x)]:  sin(pi*x)
Input the Range of x - LOW value, x(low)[0]:  >> 0
Input the Range of x - HIGH value, x(high)[0.5]:  0.5
(1)Non-uniform or (2)Uniform Segmentation or (3)Both [1]:  1
Input the Desired Error, epsilon[2^-9]:  2^-9
Input the no. of pts the fct is to be evaluated (per unit), N[10000]:  1000
Input the equation to use: (1)F(x)=mx+b or (2)F(x)=m(x-p)+b, [1]:  1
```

Figure 2. Linear Approximation Function User-Interface.

First, the user is asked:

*1. Input the Function, func[sin(pi*x)]:*

The user then designates which function to approximate and implement. The default value for all the prompts are in the brackets []. In this case, the default value is $\sin(\pi x)$ which has been the 'test' function throughout this project because of its simplicity and relevance. If the user simply presses 'Enter' at this point, the default value is used.

Next MATLAB prints out:

2. Input the Range of x - LOW value, x(low)[0]:

The user then specifies the lower-end of the range of the independent variable of the function. Zero is the default value. Next, MATLAB types:

3. Input the Range of x - HIGH value, x(high)[0.5]:

The user then specifies the upper-end of the range of the independent variable of the function. One-half is the default value. Next, MATLAB types:

4. (1)Non-uniform or (2)Uniform Segmentation or (3)Both [1]:

The user then specifies whether the approximation will be conducted where all the segments may have different lengths (non-uniform), the same length (uniform) or both. Non-uniform is the default. Further discussion on the advantages and disadvantages of non-uniform and uniform will follow. Next, MATLAB types:

5. Input the Desired Error, epsilon[2^-9]:

This user then specifies the maximum acceptable error (the absolute value of the difference between the approximation and the actual value of the function over all points). The default value is 2^{-9} for 8-bit accuracy. Next, MATLAB types:

6. Input the no. of pts the fct is to be evaluated (per unit), N[10000]:

Since MATLAB performs calculations discretely (at fixed points), this option allows the user to determine the desired resolution when performing these calculations. If the function has radical changes throughout its range, a higher resolution may be desired. Of course, the higher the resolution, the more calculations there are, and thus more time is required to perform the approximation. Next MATLAB types:

7. Input the equation to use: (1)F(x)=mx+b or (2)F(x)=m(x-p)+b, [1]:

This feature is not used, but would allow the user to have the output produced with an extra coefficient to be used on a different architecture. Default is to use the standard equation of a line: $y = mx + b$. At some point in the future, this option may be eliminated and the second option chosen automatically.

If uniform segmentation (2) is chosen in question 4, then the user is prompted with:

8. Would you like to constrain (1)Number of Segments or (2)Error [1]:

This allows the user to determine how an approximation using uniform segmentation is conducted. The default is to constrain the number of segments, usually a power of 2. If the user chooses to constrain error, he/she will be prompted with question 5 above. If *[1] Number of Segments* is used, then the following appears:

9. Input the number of Desired Segments[16]:

This prompts the user to designate the number of desired segments for a uniform approximation. The default is 16, which is easily represented by a 4-bit number.

Upon completion of the user inputs, MATLAB computes the approximation and returns to the user the following:

1. Segment endpoints in both decimal and binary fixed point representation for each segment

2. Segment slope and intercept in both decimal and binary fixed point representation for each segment

3. A graphical representation of the function with the approximation overlaid

4. A graphical representation of the error throughout the range of the approximation, with maximum error highlighted

5. VHDL code that describes the segment index and coefficient look-up to be used by the SRC (behavioral code)

```
********************************************************************************

NON-UNIFORM Segmentation
Segment   End Point   End Point          c_1        c_1                c_0        c_0
Number    (Decimal)   (Binary)           (Decimal)  (Binary)           (Decimal)  (Binary)
     0     0.121224    0000000.000111110   3.07373    0000011.000100101   0.00105    0000000.000000000
     1     0.200940    0000000.001100110   2.74354    0000010.101111100   0.04085    0000000.000010100
     2     0.269054    0000000.010001001   2.32099    0000010.010100100   0.12564    0000000.001000000
     3     0.331366    0000000.010101001   1.84314    0000001.110101111   0.25413    0000000.010000010
     4     0.390378    0000000.011000111   1.32869    0000001.010101000   0.42456    0000000.011011001
     5     0.447590    0000000.011100101   0.79036    0000000.110010100   0.63469    0000000.101000100
     6     0.499900    0000000.011111111   0.25817    0000000.010000100   0.87262    0000000.110111110

********************************************************************************
```

Figure 3. MATLAB Segmentation Output.

Figure 3 shows a typical MATLAB output after conducting the approximation and segmentation of the user-specified function using the LinAppxPfit routine. In this case, our test function *sin(πx)* is defined from $0 \leq x < 0.5$, using non-uniform segmentation with a maximum error of at 2^{-9}. The routine produced a 7 segment approximation. All fixed-point binary numbers are in a signed, twos-complement, 7.9 format (7 bits to the left and 9 bits to the right of the implicit binary point). This format allows a number between $-64 \leq x < 64$ to be represented.

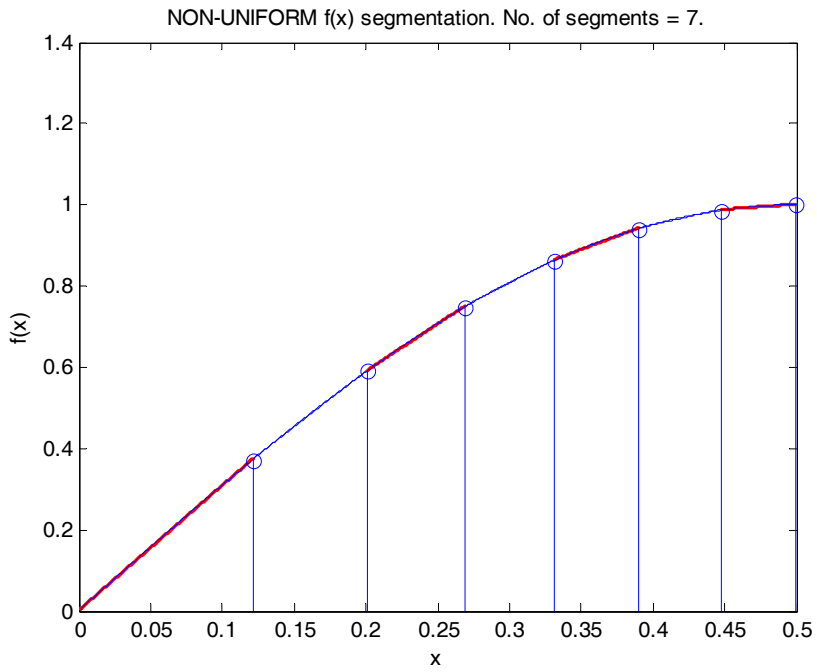

Figure 4. Graphical Representation of Function and Approximation.

Figure 4 above shows the graph of the approximation produced by MATLAB. In this figure, the approximation is superimposed on the actual function. Segments in the approximation are shown as straight lines colored red and blue alternating. Figure 5 below is a "close-up view" on the same figure where the 3rd and 4th segments meet. This shows that the segmentation truly is an approximation. The red and blue lines at this point are slightly higher than the actual function. Also, the two segments do not overlap and due to discreteness there is a slight gap between the two segments.

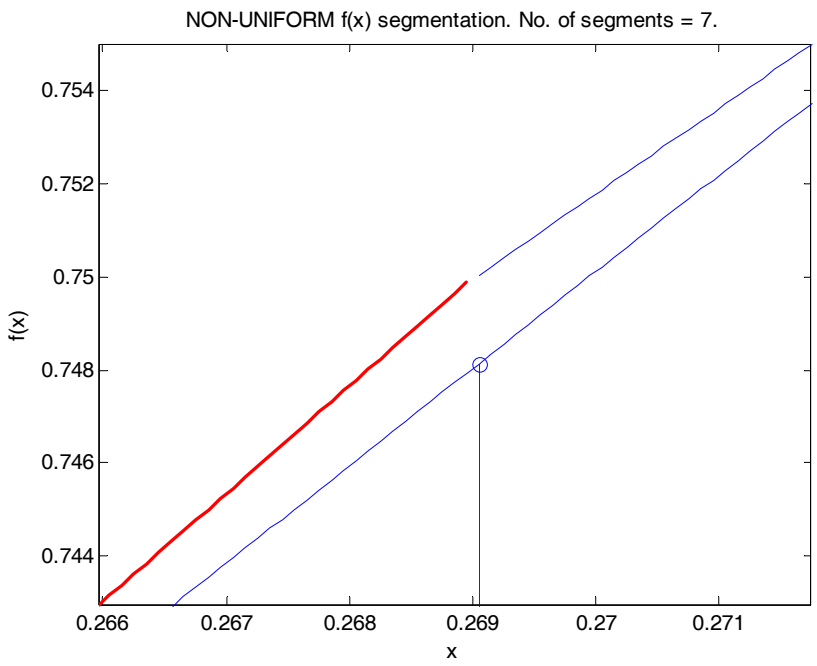

Figure 5. Close-up of Approximation.

Figure 6 shows how the error between the approximation and the actual function changes across the segmentation. Alternating curves of red and blue correspond to each of the segments. As can be seen, the error does not exceed that specified by the user. The numerical value of maximum error is displayed on the x-axis. The magnitude of the maximum positive and negative error in each segment is of equal magnitude. In the above case, the error starts negative; the difference between the actual function and the approximation is negative, therefore the approximation is greater than (above) the function. When the error curve crosses the zero axis, this is where the straight-line approximation segment intersects the actual function. The segment (in order from left to right) starts above the function, intersects the function, goes below the function, intersects the function again and then finishes up above the function, for this example.

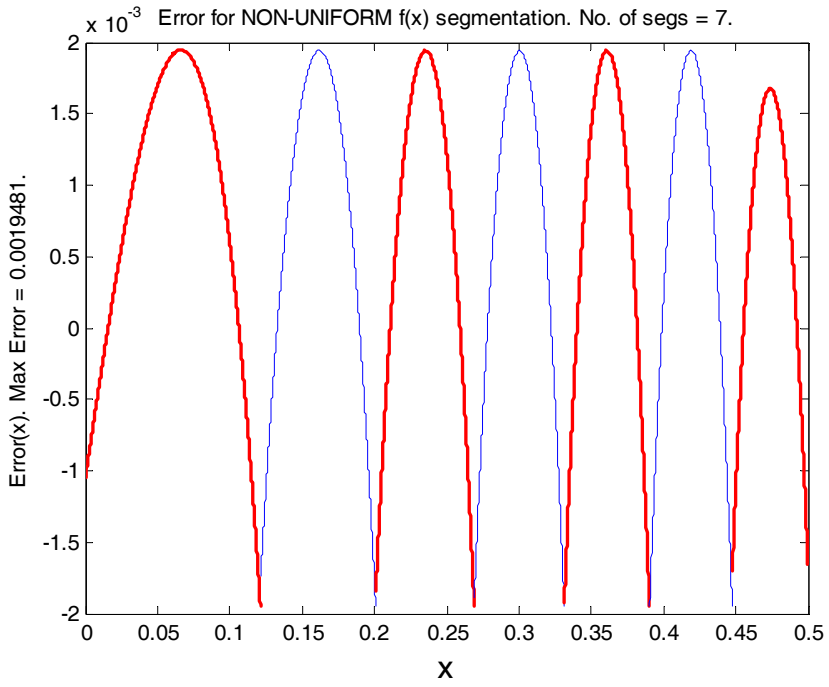

Figure 6. Error Across Approximation.

As previously discussed, there are two methods to approximating a function; non-uniform and uniform segmentation. Each method has its own advantages and disadvantages.

1. Non-Uniform Approximation Algorithm

With non-uniform approximation the width of the segment is chosen to be as large as possible so that the given error is not exceeded. In regions of the function where it is linear, the segments will be large in comparison to regions where the function's curve changes rapidly. This is where the second derivative (the rate of change of the slope) is changing the most. Non-uniform approximation results in fewer segments than uniform segmentation. But, the disadvantage is that the determination of which segment a particular input value belongs to is more complex and requires the use of a segment index encoder which must examine all of the input bits of x.

The MATLAB function *multiplelinapprox.m* (short for multiple line approximation) performs the calculations for non-uniform approximation of the function.

Multiplelinapprox.m does this with the aid of *varlinapprox.m* (short for variable line approximation) which it calls during its execution. Both of these M-files are shown in Appendix A. The algorithm also depends on the MATLAB function *polyfit*.

The non-uniform approximation algorithm uses the following procedure, starting from left to right through the range of the input values to the function.

1. Beginning with two points, *polyfit* calculates the two coefficients (slope and intercept) for a first order approximation of a given set of points.
2. Maximum error is determined between the approximation and the actual function.
3. The approximation is shifted vertically by half the distance of the maximum error, towards the actual function, so that it intersects the function somewhere besides its endpoints.
4. Error between the newly-shifted approximation and the actual function is recalculated.
5. Error is checked against the user-specified maximum error to ensure it has not been exceeded.
6. If maximum error has not been exceeded, the next adjacent point of the function is added and the process repeats with step 1.
7. If maximum error has been exceeded, after many iterations, then the previous segment is used where the maximum error requirement is not exceeded.
8. The endpoint of the given segment is recorded and the process restarts at the next given point.

2. Uniform Approximation Algorithm

With uniform approximation, each segment is the same length. Therefore, each segment is restricted to the size of the shortest segment. The algorithm must determine what the largest size can be of the shortest segment which still meets the user's error requirements. The function is then divided into segments of this length. The advantage of uniform approximation is that the process of indexing which segment a particular input belongs to becomes much easier. If the number of segments is a power of 2, then the circuit only needs to examine the $\log_2$ number of bits of the input *'x'* in order to determine what segment the input indexes. Therefore, when initially conducting the

approximation, after determining the number of segments to meet the error requirement, it is best to increase to the next power of 2. The circuit must examine the same number of bits, but the approximation will have a smaller error due to more segments. This will be achieved with only a slightly higher memory requirement. For a first order approximation with 16-bit coefficients, the difference in memory is:

$$\Delta \text{ memory} = 2 \text{ x } (\text{segs}_{\text{power of 2}} - \text{segs}_{\text{uniform}}) \text{ x 16-bits} \qquad (1)$$

Figure 7 shows the MATLAB output for the test function, *sin(πx)*, defined from $0 \leq x < 0.5$, using uniform segmentation with 16 segments. As shown, the segments are neatly distributed as defined by the 4 bits from bit position 4 – 7 (rectangle with dotted line).

```
UNIFORM Segmentation
Segment    End Point    End Point
Number     (Decimal)    (Binary)
     0      0.031009     0000000.0000 1111
     1      0.062218     0000000.0001 1111
     2      0.093328     0000000.0010 1111
     3      0.124437     0000000.0011 1111
     4      0.155546     0000000.0100 1111
     5      0.186755     0000000.0101 1111
     6      0.217864     0000000.0110 1111
     7      0.248973     0000000.0111 1111
     8      0.280083     0000000.1000 1111
     9      0.311292     0000000.1001 1111
    10      0.342401     0000000.1010 1111
    11      0.373510     0000000.1011 1111
    12      0.404619     0000000.1100 1111
    13      0.435829     0000000.1101 1111
    14      0.466938     0000000.1110 1111
    15      0.498047     0000000.1111 1111
```

Figure 7. MATLAB Output for Uniform Approximation.

The MATLAB functions *constantlinapprox.m* and *constantlinappxwerr.m* perform the calculations for the uniform approximation of a function. The first function *constantlinapprox.m* (short for constant linear approximation) performs the calculations using a user-specified number of segments. The second function *constantlinappxwerr.m* (short for constant linear approximation with error) uses the maximum allowable error as the parameter to determine the segmentation. Both of these M-files are shown in Appendix A and call upon *polyfit*.

The uniform approximation algorithm using *constantlinapprox.m* performs the following:

1. Divide the length of the domain of the function by the number of user-defined segments.
2. Using *polyfit*, determine the slope and intercept of each segment.
3. For each segment, determine the maximum error between the segment and the function.
4. Shift the approximation segment vertically one-half the distance of the maximum error, towards the actual function, so that it intersects the function somewhere besides its endpoints.

The uniform approximation algorithm, using *constantlinappxwerr.m*, performs a different procedure that is a hybrid between the two previously discussed procedures. The procedure is as follows:

1. Determine at what point on the function the second derivative is the greatest [4].
2. From this point, move outward to the left and right by one point on the function.
3. Using *polyfit*, determine the slope and intercept of a linear approximation of the points.
4. Determine the maximum error between the linear segment and the actual function.
5. Shift the approximation segment one-half the distance of the maximum error, towards the actual function, so that it intersects the function somewhere besides its endpoints.
6. The error between the newly-shifted approximation and the actual function is recalculated.
7. The error is checked against the user-specified maximum error to ensure it has not been exceeded.
8. If the maximum error has not been exceeded, the process repeats with step 2.
9. If the maximum error has been exceeded, then the previous segment is used where the maximum error requirement was not exceeded.
10. Determine the number of segments by dividing the entire length of the function by the size of the segment created.
11. Using *polyfit*, determine the slope and intercept of each segment.
12. For each segment, determine the maximum error between the segment and the function.

13. Shift the approximation segment vertically one-half the distance of the maximum error, towards the actual function, so that it intersects the function somewhere besides its endpoints.

It is likely that a user would run both uniform approximation algorithms. First, the *constantlinappxwerr.m* would be used to provide the user an idea of how many segments are needed in order to meet the error requirements for a particular function. After obtaining this number, the user would run the *constantlinapprox.m* algorithm where the next power of 2 is provided as the number of segments desired for the approximation. In this way, both the error requirements are met and the indexing is simplified by only using a few bits to determine which segment to use.

B. MATLAB RESULTS

Table 2 below shows the number of segments required for 8 and 16-bit precision, of given functions, as generated by MATLAB. It is interesting to note how many segments are needed for different functions. Some functions, such as 2^x are more suited to a uniform segmentation implementation since the difference between uniform and non-uniform is small. Non-uniform requires five segments versus uniform segmentation which requires six. Other functions, such as $\sqrt{-\ln x}$ are much more suited for non-uniform approximation. In this case, non-uniform approximation requires twelve segments where uniform approximation requires 145.

C. SUMMARY

As shown, an important step in the NFG implementation process is the approximation. This is done by dividing the function into multiple, consecutive segments. The segments can either be of uniform or non-uniform length. Usually the approximation is developed to meet certain error requirements.

After the function approximation is complete, the circuit can be accurately described using the information gathered and incorporating it into an HDL. This process will be described in detail in the next chapter.

Function	**Interval**	**Non-Uniform**		**Uniform**	
f(x)	***x***	**8-bit**	**16-bit**	**8-bit**	**16-bit**
2^x	**[0,1)**	5	75	6	91
1/x	**[1,2)**	5	75	8	130
$\sqrt{x}$	**[0,2)**	10*	216*	8,206*	5.38 x 10^8 *
$1/\sqrt{x}$	**[1,2)**	4	50	5	79
$\log_2(x)$	**[1,2)**	5	76	7	110
ln x	**[1,2)**	4	63	6	91
$\sin(\pi x)$	**[0,½)**	7	109	9	144
$\cos(\pi x)$	**[0,½)**	7	108	9	148
$\tan(\pi x)$	**[0,¼)**	5	73	9	144
$\sqrt{-\ln(x)}$	$[\frac{1}{256}, \frac{1}{4})$	12	216*	145	2,507*
$\tan^2(\pi x)+1$	**[0,¼)**	10	152	18	291
$-(x \log_2 x + (1-x) \log_2(1-x))$	**(0,1)**	16*	342*	136*	34,787*
$\frac{1}{1+e^{-x}}$	**[0,1)**	2	21	2	28
$\frac{1}{\sqrt{2\pi}} e^{\frac{-x^2}{2}}$	$[0, \sqrt{2})$	4	43	5	81

* from Ref[5].

Table 2. Number of Segments for Non-Uniform and Uniform Segmentation for 8 and 16-bit Precision.

III. NFG CIRCUIT

A. CIRCUIT OVERVIEW

The next step in the process is to design the circuit using software design tools. This is done using a combination of software tools and techniques, including Xilinx's *Integrated Software Environment (ISE)* [9,10], Mentor Graphic's *ModelSim* [11], Synplicity's *Synplify* [12,13] and standard IEEE Behavioral VHDL [14,15,16]. The end product is a behavioral description of a pipelined circuit written in VHDL which is used by the SRC computer for implementation on a FPGA.

The top-level circuit is built with Xilinx ISE's *Engineering Capture System (ECS)* tool. The ECS is a schematic editor tool in which a circuit can be built visually by simply connecting parts together. Some of the parts are already available as Xilinx primitives, such as flip-flops, grounds and power (VCC). Other parts were custom built.

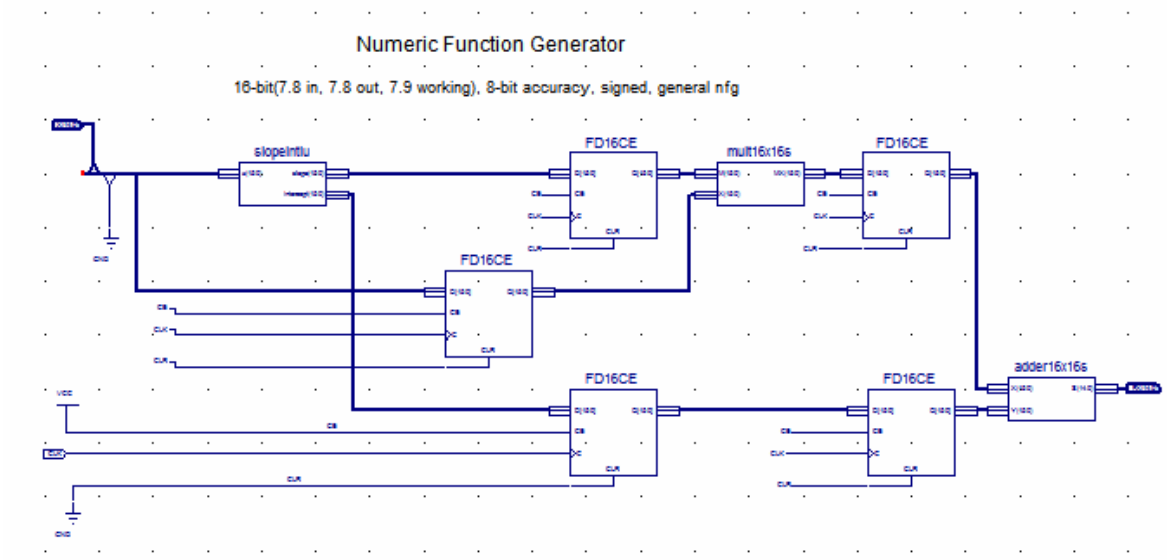

Figure 8. NFG Top-Level Schematic.

Figure 8 is the schematic of the NFG as built in Xilinx ECS. The schematic shows the input, *x*, coming into the circuit at the top left. The input passes through the *slopeintlu* (short for slope & intercept look-up) module that produces the slope and intercept coefficients to be used by the rest of the circuit. The input, slope and intercept

are all clocked into registers. On the next clock, the slope and input, *x*, pass through a signed multiplier. The product is then clocked into a register. The intercept simply passes it on to the following register to be saved until it is needed. On the last clock, the product and the slope are added together to produce a signed sum. This is the value of the function.

After constructing the circuit schematically, the VHDL code that describes the circuit can be extracted within the Xilinx ISE. This code will be used by the SRC reconfigurable computer. All VHDL code is shown in Appendix B.

B. CIRCUIT COMPONENTS

1. Slope and Intercept Look-up

The heart of the NFG is the Slope and Intercept Look-up module. In this implementation, the computation of the slope index and subsequent output of the slope and intercept coefficients is conducted all in one module. This varies from the block diagram shown in Figure 1 which shows separate process for indexing and coefficient output. This module is described behaviorally by VHDL code, which is automatically generated by MATLAB during its function approximation algorithm (item number 5 on page 8). The VHDL code uses a set of If/Then/Else statements to describe the module. For example, if the input value *x* is greater than 0.121224 but less than 0.200940, then the *slope* equals 2.74354 and the *intercept* equals 0.04085 (see Figure 3). By using behavioral VHDL, the FPGA synthesizer has maximum flexibility to construct the actual circuit on the FPGA. The user can review the construction of the circuit through various output files and reports created during synthesis to see how the circuit was developed.

By simply replacing the Slope and Intercept Look-up module with another, a different function can be generated.

2. Multiplier

The 16 by 16 signed-arithmetic multiplier is also developed in behavioral VHDL. By using VHDL, the characteristics of the multiplier, in terms of bits used for input and output, can be explicitly specified. In this case, two 16-bit numbers are multiplied. Although this could potentially produce a 32-bit output, in this circuit, a 16-bit output is produced by simply extracting the middle 16-bits of the product (bits 9 through 24). No rounding algorithm is used. The synthesizer will build the multiplier on the FPGA from this behavioral description using the resources available on the target chip.

3. Adder

In a similar way, the 16 by 16, signed-arithmetic adder module is designed in behavioral VHDL. Two 16-bit numbers are added together. The sum is the output of the circuit which is also the value of the function. The value of the function is expressed in a 15-bit, signed, two's complement, fixed-point format. The 15-bit representation is produced by removing the LSB of the 16-bit sum.

The additional the components in the circuit, specifically the FD16CE (16-Bit Data Registers with Clock Enable and Asynchronous Clear), ground and VCC (power) are standard IEEE components.

4. Number System

The circuit was designed so it could be used for as many of the target functions as shown in Table 1 as possible. Therefore, the design incorporates 15-bits, fixed-point for input and output. The 15-bits are distributed with 7 bits to the left of the decimal point and 8 bits to the right. This provides an 8-bit accuracy (2^{-8} is the lowest resolution which can be represented). In order to improve the accuracy of the circuit, 16-bits are used for the working calculations in a 7.9 format (2^{-9} resolution). A signed 16-bit number in 7.8 format can represent values between $-64 \leq x \leq 63.998$ (64 minus 2^{-8}).

C. SUMMARY

As shown, the design of the circuit is done using mainly the Xilinx ISE software. The top-level circuit is constructed in the Xilinx ECS by laying out the circuit schematically. Individual components are either readily available as part of the standard Xilinx primitives or are custom designed in VHDL.

The ModelSim software package allows the designer to verify the operation of the circuit. ModelSim integrates with Xilinx ISE. The designer can create test bench waveforms to input into the circuit. Various outputs can be placed strategically throughout the circuit to verify that the proper signals are being passed. When the circuit is not operating properly, this is a great troubleshooting tool. The designer can pinpoint down to specific components in order to determine where the fault exists.

After verifying the proper operation of the circuit, the VHDL code is ready to be transferred to the SRC computer for circuit implementation. The SRC uses Synplify [12,13] for its synthesis tool. A good engineering practice is to first synthesize the VHDL code in a stand-alone version of Synplify. This is because, when designing a circuit in the Xilinx ISE, the Xilinx XST synthesizer is used to verify the circuit construction. There have been some differences noted between the two synthesizers. Software code which works in one may not work in the other. This anomaly is further discussed in Appendix E, Lessons Learned. Therefore, it is a good design check to ensure that the code works with the synthesizer that the SRC uses.

Chapter IV will go into greater detail on how to finish implementing the circuit on the SRC and its FPGA.

IV. SRC IMPLEMENTATION

SRC Computers of Colorado Springs, CO was founded by the legendary Seymour R. Cray, who lends his initials to the company's name. SRC developed the SRC-6E reconfigurable computer that is the target architecture for the construction of the NFG.

The SRC-6E computer at NPS provides a unique architecture where an Intel-based PC is interfaced with SRC's proprietary MAP processing boards. The MAP is the heart of the system where the FPGA resides. On the MAP are three Xilinx XC2V6000 FPGAs and dual-ported memory. Only two of the FPGAs can be programmed; the other FPGA performs control functions. It is through this unique architecture and its associated interface that the NFG is implemented on the FPGA. More information on the SRC computer is available in Ref [17]. More information on the Xilinx XC2V6000 FPGA is available in Ref [18].

A. SOFTWARE CODE

The SRC system develops a pipelined circuit on the FPGA. It has the ability to implement the design from software code written in C, FORTRAN, VHDL or Verilog. Along with this code, there are a few SRC specific files which must be included in order to synthesize the design on the FPGA.

Specific files which must be provided (see Appendix C) in the project directory include:

1. main.c

This is the main routine, written in C, which runs on the SRC's Intel processor. This routine is used to interact with the user and the *.mc* subroutine which runs on the MAP processor board.

2. <subroutine>.mc

Called by *main.c* this subroutine, also written in C, executes on the MAP's FPGAs to perform a certain function. In order to take advantage of the benefits of an FPGA, this is where the programmer would place computation intensive portions of an algorithm. Results of the computations are returned back to *main.c*.

3. makefile

Commonly used in C/C++ programming, the *makefile* tells the compiler which files to use when compiling. A standard *makefile* is provided by SRC and simply modified to accommodate the programmer's unique code.

4. Macros

Macros allow a programmer to more explicitly design a function on the FPGA. Macros are called by the *.mc* files and are typically written in an HDL, such as VHDL or Verilog. By writing in one of these languages, the programmer can manipulate the circuit down to the individual bit level. In this way, operations can be done on any combination of bits. Also, the bits can be combined or split-up as necessary. Unlike C programs, macros must be manually pipelined.

In order to pipeline a circuit, the programmer must place registers in-between functional modules. As shown in Figure 8, 16-bit registers are inserted in between the slope & intercept look-up, multiplier and adder modules. Pipelining a circuit requires the programmer to determine how much work can be done in one clock cycle and then place the register to store the results until the next clock cycle. Pipelining allows for subsequent calculations to occur simultaneously, as needed results are held until they are used. Therefore, even though a calculation may take several clock cycles, once the pipeline is full, a result will be produced each clock cycle. Although a result may be

produced every clock cycle, it usually takes more than one clock cycle to do a complete computation- beginning to end. This is known as latency.

In Figure 8, the pipeline is three clocks deep. Therefore, its latency is 3. A result is produced at the output every clock cycle.

The following files are needed when implementing a macro:

a. info

The *info* file specifies the characteristics of the macro. It tells the compiler whether the macro is pipelined and specifies its latency, among other parameters.

b. blk.v

The *blk.v* file commonly known as the "black box" file specifies the macro interface. It describes the inputs and outputs (to include bit width) to and from the macro.

c. HDL Files

Written either in VHDL or Verilog, the HDL files describe the operation of the circuit. In VHDL, the circuit can be described with behavioral, dataflow or structural modeling. All the files in use must be listed in the *makefile*. They will have a suffix appropriate to the language so that the synthesizer knows how to interpret them, such as '*.v'* for Verilog, '*.vhd'* for VHDL and '*.c'* for C programming.

For the NFG circuit, the VHDL files discussed in Chapter III are provided to the SRC macro. These files behaviorally describe how the circuit should operate. The SRC synthesis tool, Synplify by Synplicity, is the tool which translates the behavioral VHDL code into the actual implementation on the FPGA.

B. SUMMARY

As shown, in order to implement the NFG on the SRC's FPGA, the user must provide all of the necessary information to the SRC system. This information is in the form of various files that provide the details of the circuit. Once of all the information is provided, the SRC is able to construct a pipelined circuit on the FPGAs resident on the MAP processing board.

The relatively easy user-interface of the SRC allowed for various design implementations to be created on the system. These different design yielded some interesting results to be discussed in the next chapter.

V. IMPLEMENTATION RESULTS

Various implementations of the NFG were constructed and compared. Speed and latency were the dominant characteristics used for comparison.

A. PC C++ IMPLEMENTATION

As a basis of comparison, an NFG was constructed using simple C++ code on an INTEL PC. Specifically, $f(x)=sin(\pi x)$ was realized. Such computations usually are in the form of a Taylor Series expansion or the CORDIC algorithm [1].

Because of the coarse timing information provided by the PC, it was necessary to calculate the $sin(\pi x)$ 100 million times (10^8). The time to perform these calculations is determined and is then divided by the total number of calculations to compute a time per calculation. In this way, an average time per calculation for the $sin(\pi x)$ was determined to be approximately 130 nanoseconds.[1]

This will serve as the baseline to compare the NFG's performance as implemented on the SRC. Of course, CPUs are always increasing in performance. But, FPGAs are also improving as well. Also, the FPGA has several key features of which the programmer can take advantage. Namely, the FPGA is virtually a blank slate, which can implement any architecture up to the limitations of the resources on the FPGA chip. As we will see, parallelism and pipelining can be exploited to increase an architecture's performance.

B. SRC IMPLEMENTATION

The SRC system allows the programmer to choose a number of different methods to implement a function. The system recognizes functions written in C and FORTRAN[2],

[1] C++ routine performed using Microsoft Visual Studio 2005 [19] on a Toshiba Satellite M30X laptop with an Intel Celeron M processor, 1.30 GHz, 768 Mb RAM.

[2] The FORTRAN compiler is not available on the NPS SRC-6E.

as well as macros written in HDLs, such as VHDL and Verilog. The system will take the software code written in any of these languages and translate it into a pipelined circuit on the MAP hardware's FPGA processor. During compilation, the SRC system provides the user with several reports.

Among the most useful are the *Inner Loop Summary* and the *Place and Route Summary*. These provide the programmer with considerable insight into the workings of the circuit.

The *Inner Loop Summary* has only 3 output lines, listed below (not in order):

1. *Pipeline depth* indicates how deep the pipeline is for a particular loop. Therefore, it is also an indication of latency for that loop. If an input is applied at time *t* to the loop. The result of that loop will be available at time *t* + *pipeline depth*.

2. *Clocks per iteration* indicates how many clocks between each successive output. In most cases, once the pipeline is full, each successive iteration will come out one clock later with a delay of *pipeline depth* from when its input was applied. Nevertheless, there are cases where there may be two or more *clocks per iteration*. In these cases, the programmer will most likely want to adjust the program to prevent this.

3. *Loop on line* 'n' indicates to which loop in the program the report applies. If the program has multiple loops, each loop will have its own *Inner Loop Summary*.

The *Inner Loop Summary* is typically an accurate indication of how fast a program will run.

The other very useful report produced by the SRC compiler is the *Place and Route Summary*. This report provides the following information:

1. *Number of Slice Flip Flops* used, usually expressed as a number 'n' out of 'm' and a percentage. These are 1-bit flip-flops that are resident on the FPGA within the slices. There are two flip-flops per slice. For the Xilinx Virtex-2 XC2V6000, m equals 67,584.

2. *Number of 4 input LUTs* used, also expressed as a number 'n' out of 'm' and a percentage. Each LUT can realize any 4-variable logic function. There are two LUTs per slice for a total of 67,584 (=m).

3. *Number of occupied Slices* used, expressed as a number 'n' out of 'm' and a percentage. A slice is the basic unit in the FPGA. Each slice has two of flip-flops and two LUTs, in addition to other logic. For the Xilinx Virtex-2 XC2V6000, m equals 33,792.

4. *Number of MULT18X18s* used, expressed as a number 'n' out of 'm' and a percentage. These are high-speed multipliers resident on the chip. The XC2V6000 has 144, 18 by 18, signed multipliers.

5. *Freq* (short for frequency) indicates at what speed the FPGA will operate. Frequency is determined by the synthesizer and varies depending upon the structure of the circuit, the target speed is 100 MHz.

The *Place and Route Summary* thus indicates how much of the FPGA is occupied by the circuit and how fast the circuit will operate.

1. SRC C Code Implementation

a. Use of C Library Functions

The FPGAs on the SRC can be programmed using C, FORTRAN or an HDL, either Verilog or VHDL. To compare with an ordinary C program running on a PC as described above, we also implemented the *sin(πx)* function in the SRC's FPGAs using the 'sin' function in the SRC standard library (libmap.h). To test this program, random values of 'x' were generated and sent to the MAP processor where the test function *sin(πx)* was computed. The *Inner Loop Summary* report for this implementation specified a *pipeline depth* of 104 clocks with one *clock per iteration*.

To further understand the circuit, this implementation was also modified where the π in *sin(πx)* was removed so that only the *sin(x)* was computed. In this case,

the *pipeline depth* reduced down to 89 clock cycles. This suggests that 15 out of the 104 clock cycles for *sin(πx)* are solely for the floating point multiplication of π times *x*.

All of the SRC source code is in Appendix C.

b. If, Then, Else Implementation (Floating Point)

The next NFG implementation on the SRC computer attempts to approximate the hardware as depicted in Figure 1. But, instead of describing the hardware with VHDL, it is described with C code. The segment index encoder and coefficient look-up is realized using "if, then, else" statements where all the values are floating point numbers. The slope coefficient is multiplied by 'x' and the intercept coefficient is added to this product. The multiplication and addition operations are C standard library functions. The *Inner Loop Summary* report for this implementation displayed a *pipeline depth* of 40 clocks with one *clock per iteration*. The domain of the input was from $0 \leq x \leq 0.5$ and a 7 segment approximation.

Therefore, by simply shifting from a library C function which is based upon the CORDIC algorithm [20] to an 'if, then, else' implementation, a 61% savings in clock cycles is realized. Of course, this time savings comes at the cost of a restricted input domain and some error in the approximation.

For this implementation, a follow-on test was performed where slope and intercept coefficients were simply set to static values, thus removing the segment index and coefficient look-up from the circuit. In this case, the *pipeline depth* is reduced from 40 clocks to 36 clocks. If the circuit is further reduced to just a multiplication of slope and input, the *pipeline depth* is reduced from 36 down to 22 clock cycles. And, if the circuit is reduced to just an addition of input and intercept without any multiplication, the *pipeline depth* is reduced from 36 to 24 clock cycles.

It is interesting to note that multiplication takes fewer clock cycles (22 clocks) than the addition process (24 clocks). This is most likely due to the fact that the FPGA has dedicated 18 by 18 multipliers on the chip while it does not have dedicated

adders. Therefore, any adders needed in an implementation must be constructed with logic and carry chains. Also, it is interesting to see that the time needed to perform the one segment implementation (36 clock cycles) does not equal the sum of its parts (the multiplier with 22 clocks and the adder with 24 clocks). This seems to suggest that the SRC synthesizer is building the circuit with some of the multiplying and adding in parallel, thus saving some clock cycles.

c. If, Then, Else Implementation (Fixed Point)

The next NFG implementation takes the approximation of the hardware in Figure 1 one-step further. The VHDL implementation discussed in Chapter 3 uses fixed-point binary numbers. Therefore, an implementation in C code using fixed-point numbers was the next appropriate step. The SRC C code in Appendix C for the segment index encoder and coefficient lookup uses fixed-point numbers represented in hexadecimal. These numbers are essentially integers, since fixed-point numbers are simply integers that have been scaled. It is up to the user to interpret the numbers correctly by applying the correct scaling factor. In this case, a scaling factor of 2^{-8} is used.

The *Inner Loop Summary* report for this implementation displays a *pipeline depth* of 18 clocks with one *clock per iteration*. This was for the same domain of the input, $0 \leq x \leq 0.5$, with a 7 segment approximation.

Thus, by shifting from floating point to fixed-point numbers in the 'if, then, else' implementation, a 55% savings in clock cycles is realized.

2. SRC Macro VHDL

The final NFG implementation is an SRC macro implementation using the VHDL source code as described in Chapter III. In this case, the programmer has more control

over how the SRC synthesizer constructs the NFG circuit. The VHDL code provides a more explicit, behavioral description of how the circuit should work.

The following is an excerpt from Ref [17] which explains the types of macros which are available for use on the SRC.

Macros can be categorized by various criteria, and the compiler treats them in different ways based on their characteristics. In the MAP compiler, five characteristics are particularly relevant:

•A macro is "stateful" if the results it computes are dependent upon previous things it has computed or seen. A simple example is a macro that sums the values that are arriving in sequence at its input. In contrast, a "non-stateful" macro computes values using only its current inputs; it has no "memory" of its past computations.

•A macro is "external" if it interacts with parts of the system beyond the code block in which it lives. For example, a macro referencing a bank of OBM, or a macro that reads/writes a control processor register. The distinction is important in determining what things can happen in parallel. Since the effects of executing two external macros may be affected by the order in which they are executed, any call to an external macro is isolated into a unique block of code. Code blocks are executed sequentially; thus, the two external macros are executed in a deterministic order.

•"Latency" is the number of clock cycles required between the time that a macro is activated with data until valid results appear. Some macros may not have a fixed latency. For example, a macro that waits for a flag register to go high will have an unpredictable wait. Since the pipelined inner loops generated by the MAP C compiler use fixed delay queues to balance the paths through the loop, all macros for inner loops must have a fixed latency.

A "pipelined" macro is able to accept new data values on its inputs while it is still internally processing the results from previous input values. A "fully pipelined" macro can accept new inputs on every clock.

A "periodic-input" macro is one that cannot take new inputs on every clock, but rather can take inputs at regular intervals. For example, a macro might be able to take new inputs every three clocks. In that case, its "period" is three.

Five types of user macros can be used by the MAP compiler: Pure Functional, Pure Functional Periodic, Stateful, Stateful Periodic, and External. The chart below shows their characteristics:

	Stateful	External	Latency	Fully Pipelined	Periodic
Pure Functional	No	No	Fixed	Yes	No
Pure Functional Periodic	No	No	Fixed	No	Yes
Stateful	Yes	No	Fixed	Yes	No
Stateful Periodic	Yes	No	Fixed	No	Yes
External	Yes or No	Yes	Variable	N/A	N/A

Figure 9. Types of Macros and their Characteristics (Ref. 2).

The macro characteristics are designated in the *info* file. For the NFG VHDL macro implementation, the following were used:

```
STATEFUL = NO
EXTERNAL = NO
PIPELINED = YES
LATENCY = 2
```

This makes the NFG a *Pure Functional* macro.

Upon synthesizing the NFG VHDL macro implementation, the *Inner Loop Summary* report for the NFG displayed a *pipeline depth* reduction to 12 clock cycles with still only one *clock per iteration*. For macros, it is up to the programmer to manually pipeline the circuit in the circuit design and then to set the latency for use by the SRC compiler in the *info* file. When synthesizing this circuit, the SRC synthesizer will automatically pipeline the rest of the circuit.

During synthesis, the SRC system creates a pipeline that feeds into the macro as well as a pipeline that collects the return values from the macro. This accounts for the difference between the *Inner Loop Summary* report which gives a value of 12 for the

pipeline depth and Figure 8 which shows a pipeline depth of 3. The SRC system has added a 10-deep pipeline overhead to the macro circuit. For this configuration, the macro has a 5-deep pipeline input and a 5-deep pipeline output. The reason the *pipeline depth* is not 13 (circuit pipeline of 3 plus 10 clock pipeline overhead) is because the programmer must indicate a latency of 2 in the *info* file for the inputs and outputs of this circuit to match up. The last stage of this 3-deep pipeline circuit is part of output pipeline created by the SRC.

If the programmer erroneously specifies the latency as 6, for example, in the *info* file, the *Inner Loop Summary* will output a *pipeline depth* of 16. In fact, the *Inner Loop Summary* will always output the value for latency in the *info* file plus 10. The way a programmer might know that they have entered an incorrect value for latency is that when a number of successive inputs are run through the macro, the outputs will not match, but may be one or two outputs off. For example, in a long string of computations, the output for the current input may not be correct, but may match up quite nicely with an input before, after or nearby. When entering a latency in the *info* file, the designer should use a value that is one less than the pipeline depth that is shown in the circuit.

A summary of these implementations is in Table 3 below.

sin (πx)

Code	Method	Segmentation	Num of Segs	Pipeline Depth	Clks per Iteration	Type
C	MAP - C	Non-Uniform	N/A	104*	1	floating pt
C	ITE	Non-Uniform	7	40	1	floating pt
C	Mult & Add	One Segment	1	36	1	floating pt
C	Mult Only	One Segment	1	22	1	floating pt
C	Add Only	One Segment	1	24	1	floating pt
C	ITE	Non-Uniform	7	18	1	fixed-pt
VHDL	ITE	Non-Uniform	7	12	1	fixed-pt

*89 when π is omitted

Table 3. Summary of Implementations.

When comparing these implementations, it is important to note that the first implementation works on floating point numbers over a large domain. The rest of the implementations work only over the limited input domain of $0 \leq x < 0.5$. And in the last two cases, the implementations use only fixed-point numbers. Therefore, the savings in *pipeline depth* and its associated latency are not realized without drawbacks.

In all of the SRC implementations, once the pipeline is full, a new result is produced each clock cycle. Running at a nominal 100 MHz, the SRC produces an output every 10 nanoseconds. Comparing to the PC C++ implementation, where the average time per calculation was approximately 130 nanoseconds, this represents a 13 times increase in performance.

C. ASIC IMPLEMENTATION

If the circuit described in Chapter 3 was implemented and placed on a free-standing FPGA or Application-Specific Integrated Circuit (ASIC), then the 10-clock cycle overhead which comes from the nature of SRCs computer system would be eliminated. A binary value could be applied to the input pins of the circuit, along with a clock signal, and an output could be received three clock cycles later. Of course, without the SRC architecture, the ability to build and re-build the circuit for alternate functions becomes more difficult. Placing the circuit on an FPGA or ASIC would be best once development is complete and for a user that does not need the added flexibility and ease of use of the SRC.

D. FPGA RESOURCES

As previously discussed, the *Place and Route Summary* is also a very useful report when compiling software on the SRC. This gives the developer an idea of how much of the FPGA is being used, and how much is still available. Table 4 below summarizes the amount of the FPGA chip used by the final implementation.

Code	Method	Segmentation	Slice FF	LUT's	Occ Slices	Mult18x18	Freq (MHz)
VHDL	ITE	Non-Uniform	4058 (6%)	1530 (2%)	2425 (7%)	1 (1%)	90.7
FPGA Total			67584	67584	33792	144	

Table 4. Summary of FPGA Resources Used.

Table 4 shows that for the final implementation only approximately 7% of the FPGA chip is used. This leaves a lot more space to add other components, possibly other NFGs, to the overall circuit.

E. COMPUTATION RESULTS

Random values were generated and fed into the SRC macro by the *main.c* and *sine.mc* routines. The results of the computations performed by the NFG were compared with values computed by Microsoft (MS) Excel. Table 6 provides a summary of these results. The table shows the value of x and the resulting value of $f(x)$ as computed on the MAP. This value is then compared to a value computed by Excel (algorithm unknown). The last two columns show the difference between Excel and the NFG and quantifies that difference in terms of the Basic Unit of Accuracy (BUA). One BUA is equivalent to the value of the LSB (2^{-8}) for the output number.

BUAs	**Difference, Δ**
0	$\Delta <$ BUA
1	BUA $\leq \Delta <$ 2 BUAs
2	2 BUAs $\leq \Delta <$ 3 BUAs
3	3 BUAs $\leq \Delta <$ 4 BUAs
4	4 BUAs $\leq \Delta <$ 5 BUAs
5	5 BUAs $\leq \Delta <$ 6 BUAs

Table 5. Difference vs Bit Equivalency

x[14:0]		sin (πx)		f(x)[14:0]		Difference	
Hex	Decimal	Hex	Decimal	Hex	Decimal	BUA	Decimal
0 ≤ x ≤ 0.5				2^-8 =	0.003906		
10	0.0625	31	0.1950903	**32**	0.195313	0	0.000222
20	0.125	61	0.3826834	**62**	0.382813	0	0.000129
30	0.1875	8E	0.5555702	**8F**	0.558594	0	0.003024
40	0.25	B5	0.7071068	**B6**	0.710938	0	0.003831
50	0.3125	D4	0.8314696	**D6**	0.835938	1	0.004468
60	0.375	EC	0.9238795	**EE**	0.929688	1	0.005808
70	0.4375	FB	0.9807853	**FD**	0.988281	1	0.007496
14	0.078125	3E	0.2429802	**3D**	0.238281	1	0.004699
54	0.328125	DB	0.8577286	**DB**	0.855469	0	0.00226
15	0.082031	41	0.2548657	**40**	0.25	1	0.004866
35	0.207031	9B	0.605511	**9A**	0.601563	1	0.003949
31	0.191406	90	0.5657318	**90**	0.5625	0	0.003232
50	0.3125	D4	0.8314696	**D4**	0.828125	0	0.003345
1	0.003906	3	0.0122715	**3**	0.011719	0	0.000553
5B	0.355469	E6	0.8986745	**E5**	0.894531	1	0.004143
7E	0.492188	FF	0.9996988	**FF**	0.996094	0	0.003605
19	0.097656	4D	0.3020059	**4C**	0.296875	1	0.005131
0 ≤ x ≤ 2				2^-8 =	0.003906		
194	1.578125	7F07	-0.970031	**7F06**	-0.976563	1	0.006531
1D4	1.828125	7F7C	-0.5141027	**7F7B**	-0.519531	1	0.005429
15	0.082031	41	0.2548657	**40**	0.25	1	0.004866
1B5	1.707031	7F34	-0.7958369	**7F33**	-0.800781	1	0.004944
1B1	1.691406	7F2C	-0.8245893	**7F2B**	-0.832031	1	0.007442
D0	0.8125	8E	0.5555702	**8D**	0.550781	1	0.004789
1	0.003906	3	0.0122715	**3**	0.011719	0	0.000553
1DB	1.855469	7F8F	-0.4386162	**7F8F**	-0.441406	0	0.00279
FE	0.992188	6	0.0245412	**5**	0.019531	1	0.00501
99	0.597656	F4	0.953306	**F3**	0.949219	1	0.004087
		sqrt(-ln(x))					
1/256 ≤ x ≤ 1/4				2^-8 =	0.003906		
14	0.078125	198	1.5966982	**198**	1.59375	0	0.002948
15	0.082031	194	1.5813459	**194**	1.578125	0	0.003221
35	0.207031	141	1.2549444	**140**	1.25	1	0.004944
31	0.191406	149	1.2858294	**148**	1.28125	1	0.004579
1	0.003906	25A	2.35482	**255**	2.332031	5	0.022789
19	0.097656	186	1.5252218	**186**	1.523438	0	0.001784

Table 6. Examples of Output Results.

F. SOURCES OF ERROR

As shown in Table 6 above, the value of the function as computed by the NFG and as computed by MS Excel does not always match. At most, these two values differ by one BUA (2^{-8} which is equivalent to 0.003906), in all of the above cases with the

exception of one case ($\sqrt{-\ln(x)}$ with an input of 1). This is proof that the NFG is computing *f(x)* properly. The difference between the NFG and Excel outputs can be attributed by several possible sources of error.

1. Function Approximation

As shown in Chapter 2, the NFG uses an approximation of the actual function to perform its calculations. Figure 5 shows that the approximation and the function do differ by up to some maximum error. This difference between the approximation and the actual function is a source of error in the circuit.

2. Conversion from Decimal to Binary in MATLAB and Excel

Both MATLAB and Excel are taking floating-point numbers and converting them into binary, fixed-point numbers. Some error is going to be introduced, since most floating point numbers do not exactly convert, given the chosen accuracy. In this case, the conversion algorithm will need to round the floating-point number. This rounding is a source of error.

3. Absence of Rounding in the Multiplier and Adder

In the NFG circuit, the complete result of the multiplier and the adder are not used. In the VHDL code that describes the multiplier, two 7.9 numbers (7 bits to the left and 9 bits to the right of the decimal point) are multiplied. This product will be in the form of a 32-bit (14.18) product. In order to convert the product back to a 7.9 number, only the middle 16-bits (bits 9 through 24) are used. The other bits are simply truncated. A more sophisticated circuit would use a rounding algorithm when removing the lower order bits. The rounding algorithm would look at the 10^{th} bit to the right of the decimal point. If this bit were 0, then the rest of the bits would be truncated. If this bit were 1, then 1 would be added to the 7.9 number. A rounding algorithm should be one of the

items for future work, thus eliminating this source of error. This same argument holds for the adder where there is a 16-bit output which is converted into a 15-bit output by truncating the LSB.

4. Insufficient Bits

As more bits are used to the right of the decimal place, numbers and arithmetic which is being performed in binary will be able to be closer to their real values. This is why the NFG uses an LSB of 2^{-9} in the *working* portion of the circuit. Even more bits would reduce this source of error by greater amounts.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSION

A. SUMMARY OF WORK

In this thesis research, a high-speed NFG circuit was developed on a COTS, reconfigurable computer.

First, MATLAB routines were developed which will approximate a given function with successive first-order, linear segments. The MATLAB code will divide the function into either uniform (same length) or non-uniform (variable length) segments. In the uniform case, the user can specify whether to use a certain number of segments or to meet a desired maximum error constraint. In the non-uniform case, only the latter choice is available. MATLAB generates the approximation along with VHDL code that has the segment endpoint, slope and intercept information.

It was decided to use MATLAB to generate VHDL to make the overall process easier on the end-user. MATLAB has the capability to write to a file. In the NFG circuit, the only object that distinguishes one function from another is the segment index encoder and coefficient look-up. Therefore, if the user wants to implement a different function, they just need to run the MATLAB routine and retrieve the MATLAB-generated VHDL file. This is much easier than having to manually inserted the MATLAB calculations into pre-formed VHDL code.

VHDL code was developed for the rest of the circuit and combined with the MATLAB-generated code to form the complete description of the circuit.

The VHDL code was transferred to the SRC-6E computer system where it was placed into a macro. The SRC used the macro VHDL code along with all of the other required information files in order to implement the NFG on the FPGA.

Several versions of the NFG were generated, both on and off the SRC, in C and VHDL code for comparison.

The NFG implementations were compared in Chapter IV as summarized in Table 3. The table provides the parameters used in comparing the various NFG

implementations. These parameters were *pipeline depth* and *clocks per iteration*. In the end, it is really up to the customer to decide what features are the most important to them.

Pipelining is what makes the SRC so powerful. All of the SRC implementations provide an output once every clock cycle. Albeit, on the SRC, the clock speed is what may be considered a relatively slow 100 MHz. When millions of calculations are needed to be performed in succession, this pipelined approach will probably be the quickest. If the latency between input and output is important to the user, then one of the implementations where latency is minimized should be considered. As with all things, each advantage has a disadvantage. The fastest circuit (smallest latency) has the most limitations. The circuit works only with fixed-point numbers and over a limited range of domain for the input values. Larger domains may be realized at the expense of more segments and thus more memory.

On the SRC, the slowest implementation had a *pipeline depth* of 104 clock cycles. The architecture of the NFG implementation reduced this down to 12 clock cycles, representing more than an 88% performance increase for this parameter. Additionally, when comparing the NFG to a function implementation using C++ on a personal computer, a performance increase of 92% was shown.

The NFG circuit can easily be reconfigured to generate alternate functions, simply by replacing the segment index and coefficient look-up portion of the circuit. Not only can elementary functions be approximated, but the complexity can easily increase with no adverse effects. For example, some of the functions in Table 1 such as, the entropy function ($x\log_2 x - (1-x)\log_2(1-x)$), would require several computations on a general purpose CPU. That is, parts such as $\log_2 x$ and $\log_2(1-x)$ are computed separately, multiplied by x and $1-x$, respectively and then summed, whereas, in the NFG circuit, all of the computations are done in the same circuit as for any other function.

B. SUGGESTED FUTURE WORK

1. Multiple NFGs in Parallel

One of the advantages about the FPGA implementations is the relatively small amount of the FPGA-chip that is consumed. The circuit can be made even more complex up to the limitations of the resources on the chip. One such implementation may be that several NFGs are implemented in parallel on the chip. Each one could separately, in parallel be calculating the result of a function based from an applied input. Another input could be used to decide which of the outputs is desired, by means of a selector. In this way, several functions are implemented and the chip does not have to be reprogrammed each time a different function was needed. This is very reasonable as evident in Table 4 where the sample function, *sin(πx)*, only takes up 7% of the FPGA.

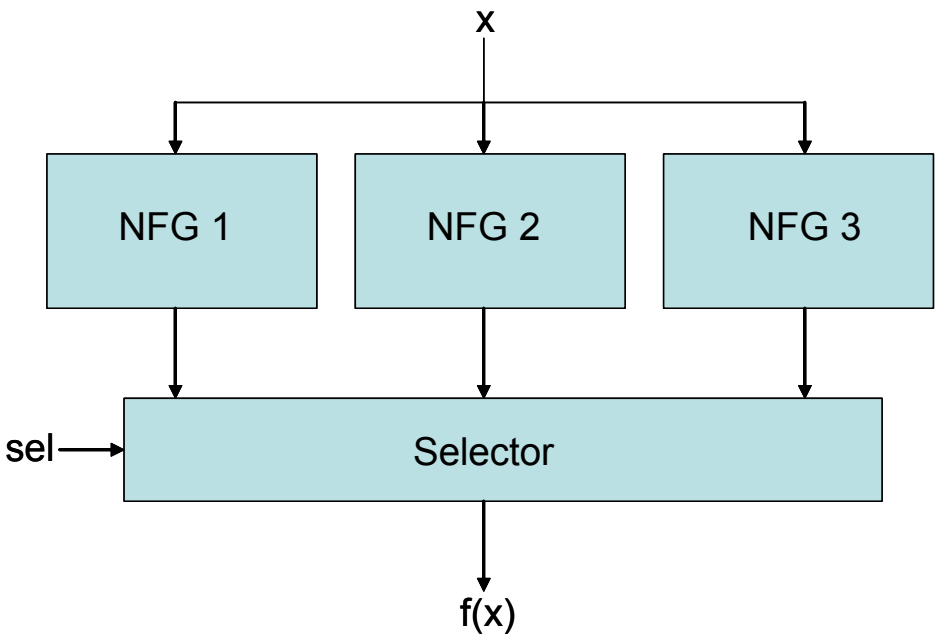

Figure 10. Multiple NFGs Working in Parallel.

Another variation on this theme would be where multiple segment index and coefficient look-up modules exist in the circuit, but the multiplier and adder are not duplicated. In this circuit, a selector would choose which segment index and coefficient look-up module to use based upon which function was needed to be calculated.

2. 16-bit and Higher Implementations

A similar circuit should be built which implements 16, 32 and 64-bit versions of the circuit. These higher accuracy circuits may be more in line with the needs of potential users. Higher accuracy will mean smaller errors, more segments and more resources of the FPGA being consumed. This will most likely require a shift in architecture from the current segment index and coefficient look-up module, which uses "if/then" statements, to one which is based on memory (see #2 below).

3. Memory Vice If/Then

Currently, the segment index and coefficient look-up occur in the same module, which is built from a behavioral VHDL description using if, then, else statements. As shown by synthesizer reports produced by the SRC and Synplicity (see Appendices F and G), this module is created using a chain of logic blocks (LUTs). This architecture has its limitations. As the complexity of the circuit increases, due to more segments and higher accuracy, the logic chains will get longer and the delay of the longest path will get longer. This in turn, will lead to slower clock frequencies in order to accommodate the longer delay path. The solution to the above problem is to split up the segment index and coefficient look-up portions of the circuit.

The coefficient look-up module is achieved by programming memory that is pre-loaded with the values as calculated by MATLAB. There are Xilinx primitives such as *RAMB16_S18_S18*, which is a 16 kilobyte block RAM with two 18-bit outputs. The RAM can be described in VHDL or Verilog, to include the initial values of the memory. MATLAB could write the VHDL code, similar to how it is done now. The memory would only have to be re-programmed when implementing a different function. Thus, the RAM is really just a ROM during NFG execution. One other option investigated, but currently does not work, would be to have MATLAB only generate the initialization

values for the RAM. These values would be placed in a separate file that would be opened by the VHDL code that describes the RAM. This was tried unsuccessfully but may be possible.

One drawback from this architecture is that the *RAMB16_S18_S18* is a 16 kilobyte RAM. It is not certain what happens when the entire 16k is not used. Does the Synthesizer only tie up the FPGA resources that are needed? Or does it use the entire 16k? Obviously, the former would be desired. In this way, a memory is created which can expand or contract based upon the characteristics of the function.

Another drawback is that when using memory for coefficient look-up, only have the problem is solved. The current architecture uses one module for both segment index encoding and coefficient look-up. When using memory, the segment index encoding portion of the module would have to occur in another, separate module. The memory simply holds the values of the coefficients. The segment index encoder would provide the address to memory for which location to read.

As previously discussed, there are two ways to approximate a function; uniform and non-uniform segmentation. In uniform segmentation, all segments are the same length, and the number of segments should be a power of two. The segment indexing for uniform approximation is simpler, since a certain number of bits ($\log_2$ of the number of segments) is simply used as the read address for the memory.

Segment indexing for non-uniform approximation is more difficult. Each segment may be a different size, so there is no set pattern to follow to determine which segment the input belongs. References [2] and [5] describe how to use a LUT cascade to encode an input into a particular segment. What is not known, is how much of a delay to the overall circuit the LUT cascade will cause.

4. Uniform Approximation Increases to Next Power of 2

It has already been shown that when using uniform segmentation, it makes the most sense to divide the function up into a number of segments equivalent to the power

of two. The MATLAB software code should be modified to automatically do this, rather than having the user run the software routine twice. Once to determine the number of segments for a given error and again to increase the segmentation to the next power of two.

5. Higher-Order Approximations

The use of higher order approximations, such as 2nd order ($y = (c_2x + c_1)x + c_0$) may be able to better approximate a function with fewer segments. This will result in a memory savings and less complex segment indexer, although, the circuit will be more complex due to the addition of a multiplier (for the c_2x) and an adder. Additionally, the memory will have to provide three coefficients vice two. Ref [21] is currently investigating higher order approximations. Most likely, the benefits of higher order approximations will only be realized on specific (less linear) functions.

6. Different Architecture, $y = c_1(x-p) + c_0$ Circuit

Another architecture which has been considered is one of the form where the characteristic equation is: $f(x) = c_1(x-p) + c_0$. In this case, the input x has a pivot point, p which is subtracted from it prior to multiplication. This architecture may lead to a smaller multiplier in the circuit, thus saving time and resources. Although, since the Xilinx Virtex-2 has 18 by 18 multipliers resident on the chip, the synthesizer might just use the 18 by 18 multiplier anyhow. The circuit will have to have three coefficients provided from memory and an additional subtractor. The advantages of this architecture need to be investigated.

7. Use of *Remez* Algorithm for Segmentation

Upon examination of Figure 6, it is evident that error at one endpoint of a segment may not necessarily match that at the other endpoint of the segment nor the endpoint of

the adjacent segment. The error may be close, but not exactly the same. This anomaly is most likely due to the use of the *polyfit* function in the MATLAB segmentation routines. The *polyfit* function performs an approximation of the given points where the error is minimized in a least squares sense. This is different than when the error to be minimized is the maximum error. The *Remez* algorithm is being investigated in Ref [21] which may solve this anomaly.

8. Rounding Vice Truncation

As discussed in Chapter V, the current circuit does not use any rounding algorithms. Not using rounding is a potential contributor to the overall error of the circuit. Future circuits should use a rounding algorithm.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. MATLAB ALGORITHMS

The following MATLAB Code generates the segmentation for any given function.

A. LINEAR APPROXIMATION USING POLYFIT

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  LinAppxPfit.m
%
%  This program produces VHDL code for the SRC-6 that realizes a numeric
%  function generator (NFG).  The user specifies a function to realize, a
%  domain over which the function is realized, a desired error, and a type
%  segmentation (uniform or non-uniform).  The program computes a piecewise
%  linear approximation using the specified segmentation-type that is
%  accurate to the specified error of the specified function in the
%  specified interval.
%
%    Created: March 16, 2005 (adapted from Arbitrary_Slope_Piecewise_Linear.m
%                                                    written by Jon Butler)
%    Last modified:  January 16, 2007
%    Produced by: Tom Mack
%
% This program applies an algorithm that produces
%               1. Uniform piecewise linear approximation
%               2. Non-uniform piecewise linear approximation
% For a description of the algorithm see  C. L. Frenzen, T. Sasao, and J.T.
%        Butler,  "The tradeoff between memory size and approximation error
%        in numeric function generators based on table lookup," preprint.
%
% For non-uniform segmentation, it uses MATLABs polyfit algorithm.
% For uniform segmentation, the program determines the minimum segment
% length needed at the point of greatest curvature.
%
%  Inputs
%      1.  N       - number of elements on which function is expressed
%      2.  f(x)    - function to be evaluated
```

```
%      3.  x_low   - low end of interval over which f(x) is evaluated
%      4.  x_high  - high end of interval over which f(x) is evaluated
%      5.  epsilon - precision of approximation (for non-uniform only)
%      6.  consegs - number of segments to use to approximate (for constant
only)
%
%  Outputs
%      1.  Segment Info - Segment No, Beginning Pt, End Pt, Slope,
Intercept, and Error
%      2.  Plot showing the approximation
%      3.  VHDL code for the SRC Computer
%
%%%%%%%%%%%%%%%%%%%% INPUT OF USER-SPECIFIED PARAMETERS %%%%%%%%%%%%%%%%%
clear
close all
format long
fprintf('\n')
fprintf('\n*******************************************************************************')
fprintf('\n')
fprintf('\n     LINEAR APPROXIMATION OF A FUNCTION USING POLYFIT with
INTERCEPT SHIFTING')
fprintf('\n                                  [DEFAULT in BRACKETS]')
fprintf('\n\n')
func = input( 'Input the Function, func[sin(pi*x)]:  ','s');
if isempty(func)
    func = 'sin(pi*x)';     %Default
end
x_low = input( 'Input the Range of x - LOW value, x(low)[0]:  ');
if isempty(x_low)
    x_low = 0;              %Default
end
x_high = input( 'Input the Range of x - HIGH value, x(high)[0.5]:  ');
if isempty(x_high)
    x_high = 0.5;           %Default
end
vari_or_const = 0;
while vari_or_const ~= 1 & vari_or_const ~= 2 & vari_or_const ~= 3  %Check
for erroneous input
    vari_or_const = input( '(1)Non-uniform or (2)Uniform Segmentation or
(3)Both [1]:  ');
        if isempty(vari_or_const)
```

```
            vari_or_const =1;       %Default
        end
end
if vari_or_const ~= 2
    epsilon = input( 'Input the Desired Error, epsilon[2^-9]:  ');
        if isempty(epsilon)
            epsilon = 2^-9;
        end
end
if vari_or_const == 2
    err_or_segs = input( 'Would you like to constrain (1)Number of Segments
or (2)Error [1]:   ');
    if isempty(err_or_segs)
        err_or_segs = 1;
    end
    if err_or_segs == 1
        consegs = input( 'Input the number of Desired Segments[16]:  ');
        if isempty(consegs)
            consegs = 16;
        end
    end
    if err_or_segs == 2
        epsilon = input( 'Input the Desired Error, epsilon[2^-9]:  ');
        if isempty(epsilon)
            epsilon = 2^-9;
        end
    end
end
N = input( 'Input the no. of pts the fct is to be evaluated (per unit),
N[10000]:  ');
if isempty(N)
    N = 10000;
end
eqn = input( 'Input the equation to use: (1)F(x)=mx+b or (2)F(x)=m(x-p)+b,
[1]:  ');
if isempty(eqn)
    eqn = 1;
end
N = N * (x_high - x_low);
x = linspace(x_low, x_high, N);
%
```

```
% Some sample functions
%
% func = '-x.*log2(x) - (1-x).*log2(1-x)';
% func  = '(1/sqrt(2*pi))*exp(-sqrt(x.^2)/2)';
% func = 'sin(x)';
% func = '2.^x';
% func = '1./((x - 0.3).^2 + 0.01) + 1./((x - 0.9).^2 + 0.04) - 6;';
% func = 'humps(x)';
% func = 'sin(x)./x';
% func = 'sin(1./x))';   x = 0.1:.001:1.0;
%
%%%%%%%%%%%%%%%%%%%%%%%%% NOTES %%%%%%%%%%%%%%%%%%%%%%
%
%  The segments in this program do NOT overlap (i.e.  the first element of
the NEXT segment
%  is NOT the last element of the PREVIOUS segment.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
eval(['F = ', func, ';'])
% Print demarcation line
fprintf('\n*******************************************************************************')
fprintf('\n')
%%%%%%%%%%%%%%%%%%%%%%%% Segmentation Algorithm %%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%% REPEAT FOR EACH i %%%%%%%%%%%%%%%%%%%%%%%%%%%%
repeat = 1;
while repeat == 1
    if (mod(vari_or_const,2) == 1)
        [endpt,seg_end_point,c_1,c_0] = multiplelinapprox(x,F,epsilon);
    end
    if (vari_or_const == 2) & (err_or_segs == 1)
        [endpt,seg_end_point,c_1,c_0] = constantlinapprox(x,F,consegs);
    end
    if ((vari_or_const == 2) & (err_or_segs == 2)) | (vari_or_const == 4)
        [endpt,seg_end_point,c_1,c_0] = constlinappxwerr(x,F,epsilon);
    end
%%%%%%%%% Compute and plot function, approximate function and error
    ind = 1;
    for i = 1:length(seg_end_point);
        m = 1;
        XP = [];
        FP = [];
```

```
        Error = [];
        while (ind < seg_end_point(i))
            XP(m) = x(ind);
            FNC(m) = F(ind);
            FP(m) = c_1(i)*x(ind) + c_0(i); % FP is fct piecewise
            Error(m) = FNC(m) - FP(m);
            ind = ind + 1;
            m = m + 1;
        end % while
        MaxError(i) = max(abs(Error));
        if (mod(i,2) == 0)  % Plot every other segment a different color
            figure(mod(vari_or_const,2)+1)
            plot(XP,FP)
            figure(mod(vari_or_const,2)+3)
            plot(XP,Error)
        else
            figure(mod(vari_or_const,2)+1)
            plot(XP,FP,'r','LineWidth',2)
            figure(mod(vari_or_const,2)+3)
            plot(XP,Error,'r','LineWidth',2)
        end % if (mod(i,2) == 0)
        figure(mod(vari_or_const,2)+1)
        hold on
        xlabel('x','FontSize',10)
        ylabel('f(x)','FontSize',10)
        if (mod(vari_or_const,2) == 1)
            title(['NON-UNIFORM f(x) segmentation. No. of segments =
',num2str(length(seg_end_point)),'.'],'FontSize',10)
        elseif (mod(vari_or_const,2) == 0)
            title(['UNIFORM f(x) segmentation. No. of segments =
',num2str(length(seg_end_point)),'.'],'FontSize',10)
        end
        figure(mod(vari_or_const,2)+3)
        hold on
        xlabel('x','FontSize',14)
        ylabel(['Error(x). Max Error =
',num2str(max(MaxError)),'.'],'FontSize',10)
        if (mod(vari_or_const,2) == 1)
            title(['Error for NON-UNIFORM f(x) segmentation. No. of segs =
',num2str(length(seg_end_point)),'.'],'FontSize',10)
        elseif (mod(vari_or_const,2) == 0)
```

```
            title(['Error for UNIFORM f(x) segmentation. No. of segs =
',num2str(length(seg_end_point)),'.'],'FontSize',10)
        end
    end % for i = 1:length(seg_endpt)
figure(mod(vari_or_const,2)+1)
plot(x,F)   % Plot function on same figure as piecewise approximation
stem(x(seg_end_point),F(seg_end_point))
hold off

%%%%%%%%%%%%%%%%%%%%%%%% Decimal to Binary Conversion Algorithm
% Convert string end points, c_1 and c_0 into a binary string with
% 8 fraction bits and print results table.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if (mod(vari_or_const,2) == 1)
    fprintf('\n NON-UNIFORM Segmentation')
elseif (mod(vari_or_const,2) == 0)
    fprintf('\n UNIFORM Segmentation')
end
if eqn == 1
fprintf('\n Segment   End Point   End Point           c_1        c_1
         c_0        c_0')
fprintf('\n Number    (Decimal)   (Binary)            (Decimal)  (Binary)
         (Decimal)  (Binary)')
end
for i = 1:length(seg_end_point)
xbin(i) = dec2binfp(x(seg_end_point(i)));
segment(i+1) = x(seg_end_point(i));   % Used in next program
c_1bin(i) = dec2binfp(c_1(i));
c_0bin(i) = dec2binfp(c_0(i));
if eqn == 1
% Print Remaining Results Table
fprintf('\n    %3d     %8.6f    %017.9f %10.5f    %017.9f %10.5f
%017.9f', i-1,  x(seg_end_point(i)), xbin(i), c_1(i), c_1bin(i), c_0(i),
c_0bin(i))
end % if eqn == 1
end %for i = 1:length(seg_end_point)

% Create text file to initialize memory
% mem = [c_1bin .* 10^9 ; c_0bin .* 10^9];
% fid = fopen('memory.mem','w');
% fprintf(fid,'%016.0f%016.0f\n',mem);
```

```
% fclose(fid);
% End text file creation

% Create VHDL file for use by the SRC-6.
fid = fopen('slopeintlu.vhd','w');
%%%%%%%%%%%%%%%%%%%%%% VHDL Code %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
fprintf(fid,'%s\n','library IEEE;');
fprintf(fid,'%s\n','use IEEE.STD_LOGIC_1164.ALL;');
fprintf(fid,'%s','use IEEE.STD_LOGIC_ARITH.ALL;');
fprintf(fid,'%s\n','use IEEE.STD_LOGIC_UNSIGNED.ALL;');
fprintf(fid,'%s\n','--- GENERATED BY MATLAB ROUTINE LinAppxPfit.m ---');
fprintf(fid,'%s\n','--- Written by Tom Mack, 5/10/2006. Modified 1/16/07.');
fprintf(fid,'%s\n','--- Segment Encoder outputs the corresponding slope and
intercept for the segment based');
fprintf(fid,'%s\n','--  upon segment endpoints.');
fprintf(fid,'%s\n','--- Segendpt and slope and intercept determined by
MATLAB7 program LinAppxPfit.m');
fprintf(fid,'%s\n','---');
fprintf(fid,'%s\n','library UNISIM;');
fprintf(fid,'%s\n','use UNISIM.VComponents.all;');
fprintf(fid,'%s\n','entity slopeintlu is');
fprintf(fid,'%s%d%s\n','     generic(x_bits:integer:=16; s_bits:integer:=16;
i_bits:integer:=16; segs:integer:=',length(seg_end_point),');');
fprintf(fid,'%s\n','     Port ( x : in std_logic_vector(x_bits-1 downto
0);');
fprintf(fid,'%s\n','             slope : out std_logic_vector(s_bits-1 downto
0);');
fprintf(fid,'%s\n','        intercept : out std_logic_vector(i_bits-1 downto
0));');
fprintf(fid,'%s\n','     type ENDPT is array(0 to segs-1) of
std_logic_vector(x_bits-1 downto 0);');
fprintf(fid,'%s\n',' end slopeintlu;');
fprintf(fid,'%s\n','');
fprintf(fid,'%s\n',' architecture Beh of slopeintlu is');
fprintf(fid,'%s\n',' begin');
fprintf(fid,'%s\n','    process (x)');
fprintf(fid,'%s\n','        variable SEGENDPT:ENDPT;');
fprintf(fid,'%s\n','    begin');

for i = 1:length(seg_end_point);
    fprintf(fid,'%s%d%s%016.0f%s\n','           SEGENDPT(',i-1,') :=
```

```
"',xbin(i)*10^9,'";');
end

fprintf(fid,'%s\n','');
fprintf(fid,'%s\n','        if x < SEGENDPT(0) then');
fprintf(fid,'%s%016.0f%s\n','           slope <= "',c_1bin(1)*10^9,'";');
fprintf(fid,'%s%016.0f%s\n','           intercept <= "',c_0bin(1)*10^9,'";');

for i = 1:length(seg_end_point)-2;
    fprintf(fid,'%s%d%s\n','        elsif x < SEGENDPT(',i,') then');
    fprintf(fid,'%s%016.0f%s\n','           slope <= "',c_1bin(i+1)*10^9,'";');
    fprintf(fid,'%s%016.0f%s\n','           intercept <= "',c_0bin(i+1)*10^9,'";');
end

fprintf(fid,'%s%016.0f%s\n','            else slope <=
"',c_1bin(length(seg_end_point))*10^9,'";');
fprintf(fid,'%s%016.0f%s\n','                 intercept <=
"',c_0bin(length(seg_end_point))*10^9,'";');
fprintf(fid,'%s\n','        end if;');
fprintf(fid,'%s\n','    end process;');
fprintf(fid,'%s\n',' end Beh;');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
fclose(fid);
% End text file creation

if eqn == 2
%%%%%%%%%%%%%%%%% The following created from: Extract_PL_Params.m
%
% This program extracts from the segmentation and the function, the
%       1.  Slope
%       2.  Intercept
%       3.  Pivot
%
% which are the parameters needed to store in the coefficients memory.  It
% produces the BINARY values of these parameters.
%
% The segmentation occurs as a vector of end points.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
fprintf('\n')
fprintf('\n*************************************************************************************')
```

```
fprintf('\n')
segment(1) = 0;
for i = 1:length(segment)
    seg_index(i) = floor(N*segment(i)/(x_high-x_low))+1;
end %for i = 1:length(segment)
seg_index;
for i = 2:length(segment)
    slope(i-1) = (F(seg_index(i)-1) - F(seg_index(i-1)))/(x(seg_index(i)-1)
- x(seg_index(i-1)));
    intercept(i-1) = F(seg_index(i)-1) - slope(i-1)*x(seg_index(i)-1);
    a = max(F(seg_index(i-1):seg_index(i)-1) -
(slope(i-1).*x(seg_index(i-1):seg_index(i)-1)+intercept(i-1) )    );
    b = min(F(seg_index(i-1):seg_index(i)-1) -
(slope(i-1).*x(seg_index(i-1):seg_index(i)-1)+intercept(i-1) )    );
    error(i-1) = 0.5*(a + b);   %YES, it is a + b.  One of a and b is 0, and
a >=0 and b <=0.
    intercept(i-1) = intercept(i-1) + error(i-1) + slope(i-1)*segment(i-1);
    s_m_e(i-1) = segment(i) - segment(i-1);
    c1x(i-1) = s_m_e(i-1)*slope(i-1);
    approx(i-1) = c1x(i-1) + intercept(i-1);
    exact(i-1) = 2^segment(i);         %Exact value of f(x) at the end of
the segment.
end %for i = 2:length(segment)
    fprintf('\nDECIMAL values for Approx = slope*(x - pivot) + intercept.')
    fprintf('\nseg no.  [s,  e]         slope   intercept     pivot
approx_error    e-s    (e-s)*slope (e-s)*slope+intercept exact f(x)\n')
for i = 1:length(segment)-1
    fprintf('%1.0f [%8.6f  %8.6f] %8.6f   %8.6f   %8.6f  %8.6f   %8.6f
%8.6f      %8.6f      %8.6f \n', i-1, segment(i), segment(i+1), slope(i),
intercept(i), segment(i), error(i), s_m_e(i), c1x(i), approx(i), exact(i))
end %for i = 1:length(segment)-1
%hold on
%plot(x(1:N),slope(1).*x(1:N)+intercept(1))
%Convert s, e, slope, intercept, and pivot to binary.
fprintf('\nBINARY values')
fprintf('\nseg no.      [s,  e]             slope      intercept
approx_error    e-s    (e-s)*slope (e-s)*sl+intercept exact f(x)\n')
for i = 1:length(segment)-1
    digits = ceil(log2(length(segment)-1));
    s_seg_no = dec2bin(i-1,digits);
    s_s(i) = dec2binfp(segment(i));
```

```
    s_e(i) = dec2binfp(segment(i+1));
    s_slope(i) = dec2binfp(slope(i));
    s_intercept(i) = dec2binfp(intercept(i));
    if error(i) < 0;
        error(i) = abs(error(i));
    end % if error(i) < 0;
    s_error(i) = dec2binfp(error(i));
    s_s_m_e(i) = dec2binfp(s_m_e(i));
    s_c1x(i) = dec2binfp(c1x(i));
    s_approx(i) = dec2binfp(approx(i));
    s_exact(i) = dec2binfp(exact(i));
    fprintf('%s [%10.8f %10.8f] %10.8f %10.8f %10.8f %10.8f %10.8f %10.8f
%10.8f \n', s_seg_no, s_s(i), s_e(i), s_slope(i),
s_intercept(i),s_error(i),s_s_m_e(i), s_c1x(i), s_approx(i), s_exact(i))
end %for i
end % if eqn == 2
fprintf('\n')
fprintf('\n*************************************************************************************')
fprintf('\n')
    if vari_or_const ~= 3
        repeat = 0;
    end
    if vari_or_const == 3
        vari_or_const = 4;
    end
end % while repeat = 1
% End file: LinAppxPfit.m
```

B. MULTIPLE LINE APPROXIMATION

```
function [endpt,indx,c1,c0] = multiplelinapprox(x,fct,max_error)
% This function will produce multiple straight-line approximations of a
% given function to within the bounds of max error provided.
% Created by Tom Mack
% Created: Mar 31, 2006
%
i = 1; indx = 1; seg_no = 1; endpt = []; c1=[]; c0=[];
while i < length(fct)
    [endpt(seg_no),indx(seg_no),c1(seg_no),c0(seg_no)] = varlinapprox(x,fct,max_error,i);
    i = indx(seg_no) + 1;
```

```
    seg_no = seg_no + 1;
end
```

C. NON-UNIFORM LINEAR APPROXIMATION

```
function [endpt,i,c1,c0] = varlinapprox(x,fct,max_error,indx)
% This function creates a straight line approximation of a given function
% using the polyfit function. It continues to calculate polyfits until
% maximum error is exceeded.
% Created by Tom Mack
% Created: Mar 31, 2006
% Modified: Jul 10, 2006
for i=indx+1:length(fct);
    p = polyfit(x(indx:i),fct(indx:i),1);
    c_1(i) = p(1); c_0(i) = p(2);
    approx(indx:i) = p(1)*x(indx:i)+p(2);
    errors = approx(indx:i) - fct(indx:i);
    maxposerror = max(errors);
    maxnegerror = min(errors);
    c_0delta(i) = abs((abs(maxposerror) - abs(maxnegerror))/2);
    if abs(maxnegerror) > abs(maxposerror)
        c_0delta(i) = -1 * c_0delta(i);
    end % if
    approx(indx:i) = approx(indx:i)- c_0delta(i);
    errors = approx(indx:i) - fct(indx:i);
    error = max(abs(errors));
    if error > max_error
        endpt = x(i-1);
        i = i-1;
        c1 = c_1(i-1);
        c0 = c_0(i-1)- c_0delta(i-1);
        return
    end % if error > max
    endpt = x(i);
    i = i-1;
    c1 = c_1(i);
    c0 = c_0(i)- c_0delta(i);
end % for i=indx+1:length(fct)
```

D. UNIFORM LINEAR APPROXIMATION

```
function [endpt,indx,c1,c0] = constantlinapprox(x,fct,consegs)
%
% This function will produce multiple straight-line approximations of a
% given function to within the bounds of the number of segments provided.
% Slope and intercept calculated by polyfit.  Intercept adjusted to
% balance maximum positive and negative errors.
% Created by Tom Mack
% Created: June 4, 2006
% Modified: July 11, 2006
%
idx=1;
for i = 1:consegs
    indx(i) = round((length(x)/consegs)*i);
    if i==consegs
        indx(i) = length(x);
    end
    endpt(i) = x(indx(i));
    p = polyfit(x(idx:indx(i)),fct(idx:indx(i)),1);
    approx(idx:indx(i)) = p(1)*x(idx:indx(i))+p(2);
    errors = approx(idx:indx(i)) - fct(idx:indx(i));
    maxposerror = max(errors);
    maxnegerror = min(errors);
    c_0delta = abs((abs(maxposerror) - abs(maxnegerror))/2);
    if abs(maxnegerror) > abs(maxposerror)
        c_0delta = -1 * c_0delta;
    end % if
    c1(i) = p(1); c0(i) = p(2)- c_0delta;   % Intercept shift to balance pos & neg error
    idx = indx(i)+1;
    i = i+1;
end
```

E. UNIFORM LINEAR APPROXIMATION WITH ERROR BOUNDS

```
function [endpt,indx,c1,c0] = constlinappxwerr(x,fct,max_error)
%
% This function will produce multiple straight-line approximations of a
% constant size of a given function to within the bounds of the
```

```
% max error provided.  Slope and intercept calculated using polyfit.
% Intercept adjusted to balance max positive and negative errors.
% Created by Tom Mack
% Created: July 10, 2006
% Modified: July 11, 2006
%
% Compute # of segs
%
firstderiv = diff(fct)./diff(x);
secndderiv = diff(firstderiv)./diff(x(1:length(firstderiv)));
[dermax,i] = max(abs(secndderiv));
error = 0;
loop_stop = 0;
i_low = i - 1;
    if i_low <= 0
        i_low = 1;
    end
    i_high = i + 1;
    if i_high > length(fct)
        i_high = length(fct);
    end
while error < max_error || loop_stop < length(fct)
    i_low = i_low - 1;
    if i_low <= 0
        i_low = 1;
    end
    i_high = i_high + 1;
    if i_high > length(fct)
        i_high = length(fct);
    end
    p = polyfit(x(i_low:i_high),fct(i_low:i_high),1);
        approx(i_low:i_high) = p(1)*x(i_low:i_high)+p(2);
        errors = approx(i_low:i_high) - fct(i_low:i_high);
        maxposerror = max(errors);
        maxnegerror = min(errors);
        c_0delta = abs((abs(maxposerror) - abs(maxnegerror))/2);
        if abs(maxnegerror) > abs(maxposerror)
            c_0delta = -1 * c_0delta;
        end % if
        approx(i_low:i_high) = approx(i_low:i_high) - c_0delta;
        errors = approx(i_low:i_high) - fct(i_low:i_high);
```

```
        error = max(abs(errors));
        if error > max_error
            i_low = i_low + 1;
            i_high = i_high -1;
        end
        loop_stop = loop_stop + 1;
end
segsize = i_high - i_low;
consegs = ceil(length(fct)/segsize);
%
% Determine slope & intercept of segments
%
idx=1;
for i = 1:consegs
    indx(i) = round((length(x)/consegs)*i);
    if indx(i) == 0
        indx(i) = 1;
    end
    if i==consegs
        indx(i) = length(x);
    end
    endpt(i) = x(indx(i));
    p = polyfit(x(idx:indx(i)),fct(idx:indx(i)),1);
    approx(idx:indx(i)) = p(1)*x(idx:indx(i))+p(2);
    errors = approx(idx:indx(i)) - fct(idx:indx(i));
    maxposerror = max(errors);
    maxnegerror = min(errors);
    c_0delta = abs(abs(maxposerror) - abs(maxnegerror))/2;
    if abs(maxnegerror) > abs(maxposerror)
        c_0delta = -1 * c_0delta;
    end % if
   % Intercept shift to balance pos & neg error
    c1(i) = p(1); c0(i) = p(2)- c_0delta;    idx = indx(i)+1;
    i = i+1;
end
```

F. FIXED-POINT DECIMAL TO BINARY

```
function [binfp] = dec2binfp(x,n)
% Function converts a decimal number to a fixed point binary number
```

```
% with one integer followed by n points to the right of the decimal
%
% Created by Tom Mack
% Last modified: August 22, 2006
%
% Inputs
%   x = decimal number to be converted (does not have to be an integer)
%   n (optional, default 9) = bits resolution to the right and left of %   decimal point
% Outputs
%   binfp = binary floating point representation
%   Negative inputs are output in 16-bit (7.9) format
%
if nargin < 2, n = 9; end
if isnan(x) == 1,
    binfp = NaN;
    return
elseif x == Inf
    binfp = Inf;
    return
elseif x < 0,
x = (x * 2^n) + 2^(2*(n-1));
    x = dec2bin(x);
    x = str2num(x);
    x = x / 10^n;
    binfp = x;
    return
else
x = x * 2^n;
    x = dec2bin(x,18);
    x = str2num(x);
    x = x / 10^n;
    binfp = x;
end
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. VHDL SOFTWARE CODE

A. NFG TOP-LEVEL VHDL CODE

```
--------------------------------------------------------------------------------
-- Copyright (c) 1995-2003 Xilinx, Inc.
-- All Right Reserved.
--------------------------------------------------------------------------------
--   ____  ____
--  /   /\/   /
-- /___/  \  /    Vendor: Xilinx
-- \   \   \/     Version : 6.3.03i
--  \   \         Application :
--  /   /         Filename : signedfct.vhf
-- /___/   /\     Timestamp : 01/31/2007 15:57:55
-- \   \  /  \
--  \___\/\___\
--
--Command:
--Design Name: FD16CE_MXILINX_signedfct
--

library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.ALL;
-- synopsys translate_off
library UNISIM;
use UNISIM.Vcomponents.ALL;
-- synopsys translate_on

entity FD16CE_MXILINX_signedfct is
  port ( C   : in    std_logic;
         CE  : in    std_logic;
         CLR : in    std_logic;
         D   : in    std_logic_vector (15 downto 0);
         Q   : out   std_logic_vector (15 downto 0));
end FD16CE_MXILINX_signedfct;

architecture BEHAVIORAL of FD16CE_MXILINX_signedfct is
  attribute INIT       : string ;
  attribute BOX_TYPE   : string ;
  component FDCE
    -- synopsys translate_off
    generic( INIT : bit :=  '0');
    -- synopsys translate_on
    port ( C   : in    std_logic;
```

```
        CE  : in    std_logic;
        CLR : in    std_logic;
        D   : in    std_logic;
        Q   : out   std_logic);
  end component;
  attribute INIT of FDCE : component is "0";
  attribute BOX_TYPE of FDCE : component is "BLACK_BOX";

begin
  I_Q0 : FDCE
    port map (C=>C,
              CE=>CE,
              CLR=>CLR,
              D=>D(0),
              Q=>Q(0));

  I_Q1 : FDCE
    port map (C=>C,
              CE=>CE,
              CLR=>CLR,
              D=>D(1),
              Q=>Q(1));

  I_Q2 : FDCE
    port map (C=>C,
              CE=>CE,
              CLR=>CLR,
              D=>D(2),
              Q=>Q(2));

  I_Q3 : FDCE
    port map (C=>C,
              CE=>CE,
              CLR=>CLR,
              D=>D(3),
              Q=>Q(3));

  I_Q4 : FDCE
    port map (C=>C,
              CE=>CE,
              CLR=>CLR,
              D=>D(4),
              Q=>Q(4));

  I_Q5 : FDCE
    port map (C=>C,
```

```
            CE=>CE,
            CLR=>CLR,
            D=>D(5),
            Q=>Q(5));

I_Q6 : FDCE
  port map (C=>C,
            CE=>CE,
            CLR=>CLR,
            D=>D(6),
            Q=>Q(6));

I_Q7 : FDCE
  port map (C=>C,
            CE=>CE,
            CLR=>CLR,
            D=>D(7),
            Q=>Q(7));

I_Q8 : FDCE
  port map (C=>C,
            CE=>CE,
            CLR=>CLR,
            D=>D(8),
            Q=>Q(8));

I_Q9 : FDCE
  port map (C=>C,
            CE=>CE,
            CLR=>CLR,
            D=>D(9),
            Q=>Q(9));

I_Q10 : FDCE
  port map (C=>C,
            CE=>CE,
            CLR=>CLR,
            D=>D(10),
            Q=>Q(10));

I_Q11 : FDCE
  port map (C=>C,
            CE=>CE,
            CLR=>CLR,
            D=>D(11),
            Q=>Q(11));
```

```
  I_Q12 : FDCE
    port map (C=>C,
              CE=>CE,
              CLR=>CLR,
              D=>D(12),
              Q=>Q(12));

  I_Q13 : FDCE
    port map (C=>C,
              CE=>CE,
              CLR=>CLR,
              D=>D(13),
              Q=>Q(13));

  I_Q14 : FDCE
    port map (C=>C,
              CE=>CE,
              CLR=>CLR,
              D=>D(14),
              Q=>Q(14));

  I_Q15 : FDCE
    port map (C=>C,
              CE=>CE,
              CLR=>CLR,
              D=>D(15),
              Q=>Q(15));

end BEHAVIORAL;


--------------------------------------------------------------------------------
-- Copyright (c) 1995-2003 Xilinx, Inc.
-- All Right Reserved.
--------------------------------------------------------------------------------
--   ____  ____
--  /   /\/   /
-- /___/  \  /    Vendor: Xilinx
-- \   \   \/     Version : 6.3.03i
--  \   \         Application :
--  /   /         Filename : signedfct.vhf
-- /___/   /\     Timestamp : 01/31/2007 15:57:55
-- \   \  /  \
--  \___\/\___\
--
```

```
--Command:
--Design Name: signedfct
--

library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.ALL;
-- synopsys translate_off
library UNISIM;
use UNISIM.Vcomponents.ALL;
-- synopsys translate_on

entity signedfct is
  port ( CLK : in    std_logic;
         X   : in    std_logic_vector (14 downto 0);
         FX  : out   std_logic_vector (14 downto 0));
end signedfct;

architecture BEHAVIORAL of signedfct is
  attribute BOX_TYPE   : string ;
  attribute HU_SET     : string ;
  signal CE       : std_logic;
  signal CLR      : std_logic;
  signal GND1     : std_logic;
  signal INT1     : std_logic_vector (15 downto 0);
  signal Prod     : std_logic_vector (15 downto 0);
  signal XLXN_65  : std_logic_vector (15 downto 0);
  signal XLXN_66  : std_logic_vector (15 downto 0);
  signal XLXN_73  : std_logic_vector (15 downto 0);
  signal XLXN_86  : std_logic_vector (15 downto 0);
  signal XLXN_134 : std_logic_vector (31 downto 16);
  signal XLXN_135 : std_logic_vector (15 downto 0);
  component VCC
    port ( P : out   std_logic);
  end component;
  attribute BOX_TYPE of VCC : component is "BLACK_BOX";

  component GND
    port ( G : out   std_logic);
  end component;
  attribute BOX_TYPE of GND : component is "BLACK_BOX";

  component FD16CE_MXILINX_signedfct
    port ( C   : in    std_logic;
           CE  : in    std_logic;
           CLR : in    std_logic;
```

```
        D   : in    std_logic_vector (15 downto 0);
        Q   : out   std_logic_vector (15 downto 0));
  end component;

  component mult16x16s
    port ( M  : in    std_logic_vector (15 downto 0);
        X  : in    std_logic_vector (15 downto 0);
        MX : out   std_logic_vector (15 downto 0));
  end component;

  component adder16x16s
    port ( X : in    std_logic_vector (15 downto 0);
        Y : in    std_logic_vector (15 downto 0);
        S : out   std_logic_vector (14 downto 0));
  end component;

  component slopeintlu
    port ( x         : in    std_logic_vector (15 downto 0);
        slope     : out   std_logic_vector (15 downto 0);
        intercept : out   std_logic_vector (15 downto 0));
  end component;

  attribute HU_SET of XLXI_43 : label is "XLXI_43_4";
  attribute HU_SET of XLXI_44 : label is "XLXI_44_0";
  attribute HU_SET of XLXI_46 : label is "XLXI_46_1";
  attribute HU_SET of XLXI_47 : label is "XLXI_47_2";
  attribute HU_SET of XLXI_48 : label is "XLXI_48_3";
begin
  XLXI_41 : VCC
    port map (P=>CE);

  XLXI_42 : GND
    port map (G=>CLR);

  XLXI_43 : FD16CE_MXILINX_signedfct
    port map (C=>CLK,
          CE=>CE,
          CLR=>CLR,
          D(15 downto 1)=>X(14 downto 0),
          D(0)=>GND1,
          Q(15 downto 0)=>XLXN_73(15 downto 0));

  XLXI_44 : FD16CE_MXILINX_signedfct
    port map (C=>CLK,
          CE=>CE,
          CLR=>CLR,
```

```
            D(15 downto 0)=>XLXN_66(15 downto 0),
            Q(15 downto 0)=>Prod(15 downto 0));

  XLXI_46 : FD16CE_MXILINX_signedfct
    port map (C=>CLK,
            CE=>CE,
            CLR=>CLR,
            D(15 downto 0)=>XLXN_135(15 downto 0),
            Q(15 downto 0)=>XLXN_65(15 downto 0));

  XLXI_47 : FD16CE_MXILINX_signedfct
    port map (C=>CLK,
            CE=>CE,
            CLR=>CLR,
            D(15 downto 0)=>XLXN_86(15 downto 0),
            Q(15 downto 0)=>INT1(15 downto 0));

  XLXI_48 : FD16CE_MXILINX_signedfct
    port map (C=>CLK,
            CE=>CE,
            CLR=>CLR,
            D(15 downto 0)=>XLXN_134(31 downto 16),
            Q(15 downto 0)=>XLXN_86(15 downto 0));

  XLXI_49 : mult16x16s
    port map (M(15 downto 0)=>XLXN_65(15 downto 0),
            X(15 downto 0)=>XLXN_73(15 downto 0),
            MX(15 downto 0)=>XLXN_66(15 downto 0));

  XLXI_50 : adder16x16s
    port map (X(15 downto 0)=>Prod(15 downto 0),
            Y(15 downto 0)=>INT1(15 downto 0),
            S(14 downto 0)=>FX(14 downto 0));

  XLXI_51 : GND
    port map (G=>GND1);

  XLXI_55 : slopeintlu
    port map (x(15 downto 1)=>X(14 downto 0),
            x(0)=>GND1,
            intercept(15 downto 0)=>XLXN_134(31 downto 16),
            slope(15 downto 0)=>XLXN_135(15 downto 0));

end BEHAVIORAL;
```

B. SLOPE AND INTERCEPT LOOK-UP CODE

This is the 'if,then,else' code used for the *sin(πx)* where $0 \leq x < 0.5$. It results in 7 non-uniform segments. Numbers are written in a 7.9 fixed-point binary format (7 digits to the left, 9 digits to the right of the decimal point).

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;use IEEE.STD_LOGIC_UNSIGNED.ALL;
--- GENERATED BY MATLAB ROUTINE LinAppxPfit.m ---
--- Written by Tom Mack, 5/10/2006. Modified 1/16/07.
--- Segment Encoder outputs the corresponding slope and intercept for
the segment based
--  upon segment endpoints.
--- Segendpt and slope and intercept initialized based upon MATLAB 7
--  segment fct approximation.
---
library UNISIM;
use UNISIM.VComponents.all;
entity slopeintlu is
     generic(x_bits:integer:=16; s_bits:integer:=16;
i_bits:integer:=16; segs:integer:=7);
     Port ( x : in std_logic_vector(x_bits-1 downto 0);
            slope : out std_logic_vector(s_bits-1 downto 0);
            intercept : out std_logic_vector(i_bits-1 downto 0));
     type ENDPT is array(0 to segs-1) of std_logic_vector(x_bits-1
downto 0);
 end slopeintlu;

 architecture Beh of slopeintlu is
 begin
      process (x)
            variable SEGENDPT:ENDPT;
      begin
                  SEGENDPT(0) := "0000000000111110";
                  SEGENDPT(1) := "0000000001100110";
                  SEGENDPT(2) := "0000000010001001";
                  SEGENDPT(3) := "0000000010101001";
                  SEGENDPT(4) := "0000000011000111";
                  SEGENDPT(5) := "0000000011100101";
                  SEGENDPT(6) := "0000000011111111";
            if x < SEGENDPT(0) then
                  slope <= "0000011000100101";
                  intercept <= "0000000000000000";
            elsif x < SEGENDPT(1) then
                  slope <= "0000010101111100";
                  intercept <= "0000000000010100";
            elsif x < SEGENDPT(2) then
                  slope <= "0000010010100100";
                  intercept <= "0000000001000000";
            elsif x < SEGENDPT(3) then
```

```
                slope <= "0000001110101111";
                intercept <= "0000000010000010";
          elsif x < SEGENDPT(4) then
                slope <= "0000001010101000";
                intercept <= "0000000011011001";
          elsif x < SEGENDPT(5) then
                slope <= "0000000110010100";
                intercept <= "0000000101000100";
          else slope <= "0000000010000100";
                intercept <= "0000000110111110";
          end if;
     end process;
end Beh;
```

C. MULTIPLIER CODE

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

-- 16x16 Signed Multiplier
-- Created by: Tom Mack,  8/22/06.   Modified:  8/26/06.
-- Intended to multiply a 16-bit (7.9) number (Slope) with a 15-bit (7.9) number (X)
-- Output is a 16-bit (8.8) number.

entity mult16x16s is
   Port ( M : in std_logic_vector(15 downto 0);
          X : in std_logic_vector(15 downto 0);
          MX  : out std_logic_vector(15 downto 0));
end mult16x16s;

architecture Beh of mult16x16s is
signal MX32bit : std_logic_vector(31 downto 0);

begin
          MX32bit(31 downto 0) <= M(15 downto 0) * X(15 downto 0);
          MX(15 downto 0) <= MX32bit(24 downto 9);
end Beh;
```

D. ADDER CODE

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
```

```
use IEEE.STD_LOGIC_SIGNED.ALL;

--library UNISIM;
--use UNISIM.VComponents.all;

entity adder16x16s is
        generic (in_bits:integer:=16; out_bits:integer:=15);
    Port ( X : in std_logic_vector(in_bits-1 downto 0);
                Y : in std_logic_vector(in_bits-1 downto 0);
           S : out std_logic_vector(out_bits-1 downto 0));
end adder16x16s;

architecture Beh of adder16x16s is
signal S16bit : std_logic_vector(15 downto 0);
begin

        S16bit <= X + Y;
        S <= S16bit(15 downto 1);

end Beh;
```

APPENDIX C. SRC COMPUTER VHDL MACRO IMPLEMENTATION CODE[3]

A. C IMPLEMENTATION CODE USING MATH LIBRARY

1. Main.c

```
static char const cvsid[] = "$Id: main.c,v 2.1 2005/06/14 22:16:46 jls Exp $";

#include <libmap.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define SZ 65536

void subr (float*, float*, int, int64_t*, int);

int main (int argc, char *argv[]) {
    int i, num;
    float *A, *D, HIGH, LOW, *F;
    int64_t tm;
    int mapnum = 0;

    if (argc < 2) {
      fprintf (stderr, "need number of elements (up to %d) as arg\n", SZ);
      exit (1);
      }

    if (sscanf (argv[1], "%d", &num) < 1) {
      fprintf (stderr, "need number of elements (up to %d) as arg\n", SZ);
```

[3] All SRC code adapted from code developed by SRC Computers, Inc. as provided in the *SRC Carte™ Training Exercises, Release 2.1* [19].

```
  exit (1);
  }

if (num > SZ) {
  fprintf (stderr, "need number of elements (up to %d) as arg\n", SZ);
  exit (1);
  }

A = (float*) malloc (SZ * sizeof (float));
F = (float*) malloc (SZ * sizeof (float));
D = (float*) malloc (SZ * sizeof (float));

srandom (99);

HIGH = 0.5;
LOW = 0.0;

for (i=0; i<num; i++) {
    A[i] = rand() % 256 * (HIGH - LOW) / 256 + LOW;
    }

map_allocate (1);

// call the MAP routine
subr (A, D, num, &tm, mapnum);

printf ("%lld clocks\n", tm);

time_t start;
time_t stop;
time( &start );
for (i=0; i<num; i++) {
   F[i] = sin(A[i]*3.14159);
    }
time ( &stop );
printf( "\t\tTime = %Le nano-seconds\n", difftime( stop, start ) * 1000000000);

for (i=0; i<num; i++) {
```

```
       printf ("Iteration %d. Sine of (%f * pi) equals %f MAP-C\n", i, A[i], D[i]);
       printf ("             Sine of (%f * pi) equals %f   C\n", A[i], F[i]);
      }

    map_free (1);

    exit(0);
    }
```

2. Sin.mc

```
#include <libmap.h>

#define SZ 65536

void subr (float A[], float D[], int num, int64_t *time, int mapnum) {
    OBM_BANK_A (AL, int64_t, SZ)
    OBM_BANK_D (DL, int64_t, SZ)
    int64_t t0, t1;
    float v0, v1, res0, res1;
    int i;

    DMA_CPU (CM2OBM, AL, MAP_OBM_stripe(1,"A"), A, 1, SZ*sizeof(float), 0);
    wait_DMA (0);

    read_timer (&t0);

    for (i=0; i<num/2; i++) {
      split_64to32_flt_flt(AL[i], &v0, &v1);
      res0 = sinf(v0*3.14159);
      res1 = sinf(v1*3.14159);
      comb_32to64_flt_flt(res0,res1,&DL[i]);
      }

    read_timer (&t1);
```

```
    *time = t1 - t0;

    DMA_CPU (OBM2CM, DL, MAP_OBM_stripe(1,"D"), D, 1, SZ*sizeof(float), 0);
    wait_DMA (0);
    }
```

B. C IMPLEMENTATION CODE USING IF, THEN, ELSE

1. Floating Point

a. Main.c

```
#include <libmap.h>
#include <stdlib.h>
#include <math.h>

#define SZ 65536

void subr (double*, double*, int, int64_t*, int);

int main (int argc, char *argv[]) {
    int i, num;
    double *A, *D, HIGH, LOW;
    int64_t tm;
    int mapnum = 0;

    if (argc < 2) {
      fprintf (stderr, "need number of elements (up to %d) as arg\n", SZ);
      exit (1);
      }

    if (sscanf (argv[1], "%d", &num) < 1) {
```

```
  fprintf (stderr, "need number of elements (up to %d) as arg\n", SZ);
  exit (1);
  }

if (num > SZ) {
  fprintf (stderr, "need number of elements (up to %d) as arg\n", SZ);
  exit (1);
  }

A = (double*) malloc (SZ * sizeof (double));
D = (double*) malloc (SZ * sizeof (double));

srandom (99);

HIGH = 0.5;
LOW = 0.0;

for (i=0; i<num; i++) {
    A[i] = rand() % 256 * (HIGH - LOW) / 256 + LOW;
  if (A[i] < 0) A[i] = -A[i];
    }

map_allocate (1);

// call the MAP routine
subr (A, D, num, &tm, mapnum);

// print results
printf ("%lld clocks\n", tm);

for (i=0; i<num; i++) {
   printf ("Iteration %d. Sine of (%f * pi) equals %f ITE\n", i, A[i], D[i]);
   printf ("             Sine of (%f * pi) equals %f  C\n", A[i], sin(A[i]*3.14159));
  }

map_free (1);

exit(0);
```

```
    }
```

b. Sin.mc

```
#include <libmap.h>

#define SZ 65536

void subr (double A[], double D[], int num, int64_t *time, int mapnum) {
    OBM_BANK_A (AL, double, SZ)
    OBM_BANK_D (DL, double, SZ)
    int64_t t0, t1;
    float slope, intercept;
    int i;

    DMA_CPU (CM2OBM, AL, MAP_OBM_stripe(1,"A"), A, 1, SZ*sizeof(double), 0);
    wait_DMA (0);

    read_timer (&t0);

   for (i=0; i<num; i++) {
      if (AL[i] < 0.121224) {
            slope = 3.07373;
            intercept = 0.00105;
            }
      else if (AL[i] < 0.200940) {
            slope = 2.74354;
            intercept = 0.04085;
            }
      else if (AL[i] < 0.269054) {
            slope = 2.32099;
            intercept = 0.12564;
            }
      else if (AL[i] < 0.331366) {
```

```
            slope = 1.84314;
            intercept = 0.25413;
            }
      else if (AL[i] < 0.390378) {
            slope = 1.32869;
            intercept = 0.42456;
            }
      else if (AL[i] < 0.447590) {
            slope = 0.79036;
            intercept = 0.63469;
            }
        else  {
            slope = 0.25817;
            intercept = 0.87262;
            }
      DL[i] = slope * AL[i] + intercept;
      }

    read_timer (&t1);

    *time = t1 - t0;

    DMA_CPU (OBM2CM, DL, MAP_OBM_stripe(1,"D"), D, 1, SZ*sizeof(double), 0);
    wait_DMA (0);
    }
```

2. Fixed Point

a. Main.c

```
#include <libmap.h>
#include <stdlib.h>
#include <math.h>
```

```
#define SZ 65536

void subr (int64_t*, int64_t*, int, int64_t*, int);

int main (int argc, char *argv[]) {
    int i, num;
    int64_t *A, *D;
    int64_t tm;
    int mapnum = 0;

    if (argc < 2) {
      fprintf (stderr, "need number of elements (up to %d) as arg\n", SZ);
      exit (1);
      }

    if (sscanf (argv[1], "%d", &num) < 1) {
      fprintf (stderr, "need number of elements (up to %d) as arg\n", SZ);
      exit (1);
      }

    if (num > SZ) {
      fprintf (stderr, "need number of elements (up to %d) as arg\n", SZ);
      exit (1);
      }

    A = (int64_t*) malloc (SZ * sizeof (int64_t));
    D = (int64_t*) malloc (SZ * sizeof (int64_t));

    srandom (99);

    for (i=0; i<num; i++) {
        A[i] = rand() % 256;
      if (A[i] < 0) A[i] = -A[i];
        }

    map_allocate (1);
```

```
    // call the MAP routine
    subr (A, D, num, &tm, mapnum);

    // print results
    printf ("%lld clocks\n", tm);

    for (i=0; i<num; i++) {
      float input = A[i] * pow(2,-9);
      float output = D[i] * pow(2,-14);
       printf ("Iteration %d. Sine of (%f * pi) equals %f ITE INT\n", i, input, output);
            }

    map_free (1);

    exit(0);
    }
```

b. Sin.mc

```
#include <libmap.h>

#define SZ 65536

void subr (int64_t A[], int64_t D[], int num, int64_t *time, int mapnum) {
    OBM_BANK_A (AL, int64_t, SZ)
    OBM_BANK_D (DL, int64_t, SZ)
    int64_t t0, t1;
    int64_t slope, intercept;
    int i;

    DMA_CPU (CM2OBM, AL, MAP_OBM_stripe(1,"A"), A, 1, SZ*sizeof(int64_t), 0);
    wait_DMA (0);

    read_timer (&t0);
```

```
for (i=0; i<num; i++) {
   if (AL[i] < 0x1F) {
         slope = 0x312;
         intercept = 0x00;
         }
   else if (AL[i] < 0x33) {
         slope = 0x2BE;
         intercept = 0x0A;
         }
   else if (AL[i] < 0x44) {
         slope = 0x252;
         intercept = 0x20;
         }
   else if (AL[i] < 0x54) {
         slope = 0x1D7;
         intercept = 0x41;
         }
   else if (AL[i] < 0x63) {
         slope = 0x154;
         intercept = 0x6C;
         }
   else if (AL[i] < 0x72) {
         slope = 0x0CA;
         intercept = 0xA2;
         }
     else  {
         slope = 0x042;
         intercept = 0xDF;
         }
   DL[i] = slope * AL[i] + intercept;
   }

 read_timer (&t1);

 *time = t1 - t0;

 DMA_CPU (OBM2CM, DL, MAP_OBM_stripe(1,"D"), D, 1, SZ*sizeof(int64_t), 0);
 wait_DMA (0);
```

```
    }
```

C. SRC VHDL MACRO IMPLEMENTATION CODE

1. SRC C Coding

a. Main.c

```
#include <libmap.h>
#include <stdlib.h>
#include <math.h>

#define SZ 65536

void subr (int64_t*, int64_t*, int, int64_t*, int);

int main (int argc, char *argv[]) {
    int i, num;
    int64_t *A, *D;
    int64_t tm;
    int mapnum = 0;

    if (argc < 2) {
      fprintf (stderr, "need number of elements (up to %d) as arg\n", SZ);
      exit (1);
      }

    if (sscanf (argv[1], "%d", &num) < 1) {
      fprintf (stderr, "need number of elements (up to %d) as arg\n", SZ);
      exit (1);
      }
```

```
    if (num > SZ) {
      fprintf (stderr, "need number of elements (up to %d) as arg\n", SZ);
      exit (1);
      }

    A = (int64_t*) malloc (SZ * sizeof (int64_t));
    D = (int64_t*) malloc (SZ * sizeof (int64_t));

    srandom (99);

    for (i=0; i<num; i++) {
      A[i] = rand() % 512;  // 9-bits
      if (A[i] < 0) A[i] = -A[i]; // Keeping it positive
      }

    map_allocate (1);

    // call the MAP routine
    subr (A, D, num, &tm, mapnum);

    // print results
    printf ("%lld clocks\n", tm);

    for (i=0; i<num; i++) {
       printf ("Iteration %d. Sine of (%llx * pi) equals %llx VHDL Signed\n", i, A[i], D[i]);
       }

    map_free (1);

    exit(0);
    }
```

b. Sine.mc

```
#include <libmap.h>
```

```
#define SZ 65536

void subr (int64_t A[], int64_t D[], int num, int64_t *time, int mapnum) {
    OBM_BANK_A (AL, int64_t, SZ)
    OBM_BANK_D (DL, int64_t, SZ)
    int64_t t0, t1;
    int i, x, fx;
    void sinfct(int x, int *fx);

    DMA_CPU (CM2OBM, AL, MAP_OBM_stripe(1,"A"), A, 1, SZ*sizeof(int64_t), 0);
    wait_DMA (0);

    read_timer (&t0);

    for (i=0; i<num; i++) {
            x = AL[i];
            sinfct(x,&fx);
            DL[i] =  fx;
      }
    read_timer (&t1);

    *time = t1 - t0;

    DMA_CPU (OBM2CM, DL, MAP_OBM_stripe(1,"D"), D, 1, SZ*sizeof(int64_t), 0);
    wait_DMA (0);
    }
```

c. Makefile

```
# $Id: Makefile,v 2.0.0.1 2005/06/10 23:12:59 hammes Exp $
#
# Copyright 2003 SRC Computers, Inc.  All Rights Reserved.
#
#       Manufactured in the United States of America.
#
```

```
# SRC Computers, Inc.
# 4240 N Nevada Avenue
# Colorado Springs, CO 80907
# (v) (719) 262-0213
# (f) (719) 262-0223
#
# No permission has been granted to distribute this software
# without the express permission of SRC Computers, Inc.
#
# This program is distributed WITHOUT ANY WARRANTY OF ANY KIND.
#
# ---------------------------------

# ---------------------------------
# User defines FILES, MAPFILES, and BIN here
# ---------------------------------
FILES       = main.c

MAPFILES    = sine.mc

BIN         = sinebin

# ---------------------------------
# Multi chip info provided here
# (Leave commented out if not used)
# ---------------------------------
#PRIMARY    = <primary file 1>   <primary file 2>

#SECONDARY  = <secondary file 1> <secondary file 2>

#CHIP2            = <file to compile to user chip 2>

#---------------------------------
# User defined directory of code routines
# that are to be inlined
#---------------------------------

#INLINEDIR  =
```

```
# ----------------------------------
# User defined macros info supplied here
#
# (Leave commented out if not used)
# ----------------------------------
#
# SRC is case-sensitive, but vhdl code created in Xilinx from schematics will ignore and not use upper-case
# Recommend not using upper-case in macro files / module names / etc
#
MACROS      = macro/sin.vhd macro/adder16x16s.vhd macro/mult16x16s.vhd macro/slopeintlu.vhd

MY_BLKBOX   = macro/blk.v
MY_NGO_DIR  = macro
MY_INFO     = macro/info
# ----------------------------------
# Floating point macros selection
# ----------------------------------

#FPMODE           = SRC_IEEE_V1 # Default SRC version IEEE
#FPMODE           = SRC_IEEE_V2 # Size reduced SRC IEEE with
                       # special rounding mode
# ----------------------------------
# User supplied MCC and MFTN flags
# ----------------------------------

MY_MCCFLAGS       = -log -v
MY_MFTNFLAGS      = -log -v

# ----------------------------------
# User supplied flags for C & Fortran compilers
# ----------------------------------

CC          = icc       # icc   for Intel cc for Gnu
FC          = ifort     # ifort for Intel f77 for Gnu
#LD         = ifort     # for Fortran or C/Fortran mixed
LD          = icc # for C codes
```

```
MY_CFLAGS   =
MY_FFLAGS   =
MY_LDFLAGS  =     # Flags to include libs if needed
# ----------------------------------
# VCS simulation settings
# (Set as needed, otherwise just leave commented out)
# ----------------------------------

#USEVCS           = yes # YES or yes to use vcs instead of vcsi
#VCSDUMP    = yes # YES or yes to generate vcd+ trace dump
# ----------------------------------
# No modifications are required below
# ----------------------------------
MAKIN   ?= $(MC_ROOT)/opt/srcci/comp/lib/AppRules.make
include $(MAKIN)
```

2. SRC Macro Files

a. Info

```
BEGIN_DEF "sinfct"
 MACRO = "sin";
 STATEFUL = NO;
 EXTERNAL = NO;
 PIPELINED = YES;
 LATENCY = 2;
 INPUTS = 1:
  I0 = INT 32 BITS (X[14:0])
  ;
 OUTPUTS = 1:
  O0 = INT 32 BITS (FX[14:0])
```

```
  ;
 IN_SIGNAL : 1 BITS "CLK" = "CLOCK";

DEBUG_HEADER = #
 void sinfct__dbg (int x, int *fx);
 #;

DEBUG_FUNC = #
 void sinfct__dbg (int x, int *fx) {

 }
 #;
END_DEF
```

b. Blk.v

```
module sin (CLK, X, FX)/* synthesis syn_black_box */;
  input CLK;
  input [14:0] X;
  output [14:0] FX;
endmodule
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D. PC C++ CODE

```
//*****************************************************************
// File: FctTime.cpp
// Name: Tom Mack
// Course: Thesis
// OS: WinXP Pro
// Compiler: Visual Studio 2005
// Date: 7 December 2006
// Description: Function Time
//
// Inputs:  NONE
//
// Output:
//
// Process:
//
// Warnings: None.
//*****************************************************************

#include <iostream>  //Header for I/O
using std::cout;
using std::endl;

#include <ctime>
using std::time;

#include <cstdlib>
using std::rand;
using std::srand;

#include <cmath>
using std::sin;
using std::tan;
using std::cos;
```

```
#include <iomanip>
using std::setprecision;
using std::fixed;
using std::scientific;


int main()
{
   double x, fx, timePerCalc;
   double  duration, duration1, duration2;
   clock_t start, finish1, finish2;
   int time1, time2;

   double base = 10;
   double exp = 8;

   const int ITERATIONS = pow(base, exp);

   x = static_cast<double>(( rand() % 5000 )) / 10000;

   start = clock();  //Set Start Time

   for (int ix = 1; ix <= ITERATIONS; ix++)
   {
     //Do Nothing
   }

   finish1 = clock(); //Get End Time

   for (int ix = 1; ix <= ITERATIONS; ix++)
   {
      fx = sin(3.14159 * x);
   }

   finish2 = clock(); //Get End Time

   duration1 = static_cast<double>(finish1 - start) / CLOCKS_PER_SEC;
   duration2 = static_cast<double>(finish2 - finish1) / CLOCKS_PER_SEC;
```

```
   duration = duration2 - duration1;

   timePerCalc = duration / ITERATIONS;

   cout << "\n*** sin(x) ***\n";
   cout << "\n\nx: " << x << " fx: " << fx << "\n\n";
   cout << "Number of iterations: " << ITERATIONS << " = " << base << "e" << exp << "\n\n";
   /*cout << "Start Time: " << start << "\n";
   cout << "Finish1  Time: " << static_cast<double>(finish1) / CLOCKS_PER_SEC << " seconds\n";
   cout << "Finish2  Time: " << static_cast<double>(finish2) / CLOCKS_PER_SEC << " seconds\n";
   cout << "Difference: " << duration << " seconds\n\n";*/
   cout << "Time per calculation " << timePerCalc << " = " << timePerCalc / 0.000000001
      << " nanoseconds " << '\n' << endl;

   return 0;
}//End main()
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX E. LESSONS LEARNED

The following is a collection of Lessons Learned while working with the SRC 6 and related software discussed in this thesis. The intent is to provide future users with a reference where they may be able to find potential solutions if encountered with similar issues.

A. FILE NAMING PROBLEMS

Problem: When you compile your VHDL code using Xilinx's ISE Navigator, it accepts upper and lower case versions of letters as the same. That is, `adderVerilog.vhd` and `adderverilog.vhd` are the same file to Xilinx's ISE Navigator. However, files in the SRC are case sensitive. That is, `adderVerilog.vhd` and `adderverilog.vhd` are DIFFERENT files in the SRC-6. So, if you have listed `adderverilog.vhd` in your Makefile as a macro, it will not recognize `adderVerilog.vhd` as the target file. Additionally, if you let Xilinx create VHDL code from a schematic which contains the module `adderVerilog.vhd` it will list refer to the module in the VHDL code as `adderverilog.vhd.`

Solution: Use lower case letters for ALL files.

Author: J.T. Butler
Date: 26 FEB 07

B. USING THE `CONST` CONSTRUCT IN C

Problem: A martello64 error is obtained when using

```
int64_t  array[5][5] = { {1,2,3,4,5};
                         {6,7,8,9,10};
                         {11,12,13,14,15};
                         {16,17,18,19,20};
                         {21,22,23,24,25} };
```

The error is caused by "too many accesses to BRAM".

Background: This is a correct C construct when used on a PC or workstation. However, when it is in a `.mc` file, this declaration will cause a martello64 error. It is possibly due to too many accesses to a BRAM (arrays are usually stored in BRAM).

This was a problem that Scott Bailey experienced. The initial writeup is based on a conversation between Scott Bailey and Jon Butler on December 1, 2006

Solution: In discussing this with Dave Caliga, Scott learned that the CARTE 2.2 version should correct this error. At the time the error occurred, we were using CARTE 2.1. Apparently, CARTE 2.2 spaces out the accesses to BRAM so that it can be changed to include ALL 25 data values. However, in order to use it in CARTE 2.2, you need to declare the array as a `constant`, like so

```
const  int64_t  array[5][5] = { {1,2,3,4,5};
                                {6,7,8,9,10};
                                {11,12,13,14,15};
                                {16,17,18,19,20};
                                {21,22,23,24,25} }
```

The intent of `const` is to set up a constant array that is not changed in the rest of the program, much like a ROM instead of RAM.

Scott Bailey tried to work around this error by simply defining the array without populating it with initial values, using, for example: `int64_t array[5][5];` The compiler accepted this. He then put the desired values into `array` using `for` loops. These arrays will then work as normal C arrays within the .mc code. However, this decreases performance, since the values placed into the array must come from either OBM or streams, access of which will incur a time penalty. Scott believes that the problem is in putting too many values into BRAM too quickly. In a dialog with Dave Caliga (SRC Computers), Dave said that the problem occurs when there are more than 8 initialized values placed in the array. Scott believes that this problem will occur in BOTH CARTE 2.1 and 2.2 for non-constant BRAM arrays.

Author: J.T. Butler
Date: 26 FEB 07

C. INCORRECT ARGUMENTS IN SYSTEM SUPPLIED MACROS

Problem: A core dump occurs when the call-by-value and call-by-reference conventions are not adhered to

```
popcount_64(int64_t a, int array[i])
```

Instead of an error message, there will be a core dump.

Background: This was provided by Scott Bailey in a conversation with Jon Butler on December 1, 2006.

Solution: To solve this problem, use the following code.

```
popcount_64(int64_t a, &temp)
array[i] = temp;
```

For most system macros, SRC requires that the input values be passed as call-by-value (e.g. `a`) and all output values be done as call-by-reference (e.g. `&temp`).

Author: J.T. Butler
Date: 26 FEB 07

D. IF / THEN / ELSE LIMITATION

Problem: When programming in C within the .mc file (no macro) an error occurs when the "If, then, else" chain is too long (approx 26 long).

Background: This was discovered by Prof. Jon Butler when trying to implement a long "if,then,else" string during testing.

Solution: SRC Carte V2.2 fixes this problem.

Author: T.J. Mack
Date: 26 FEB 07

E. MULTIPLE FILES USED IN A MACRO

Problem: When using multiple files to describe a circuit in a macro, the SRC won't successfully compile.

Background: This was discovered while developing the NFG macro where different modules are described in separate VHDL files.

Solution: List all of the VHDL files within the *Makefile* under macros, separated by a space.

Author: T.J. Mack
Date: 26 FEB 07

F. XILINX / SYNPLIFY INCONSISTENCIES

Problem: VHDL code synthesizes correctly (no errors) in Xilinx XST, but does not in Synplify PRO.

Background: When developing VHDL code for the NFG, the code was originally written in the Xilinx ISE. Checking for errors using Xilinx XST resulted in no errors. When the code was transported to the SRC, errors resulted. Further troubleshooting produced the same errors when using the stand-alone Synplify.

Solution: Not all code is universal. Always test code using a stand-alone version of Synplify. If it results in errors, the code must be modified.

Author: T.J. Mack
Date: 26 FEB 07

G. MODELSIM AND MULTIPLE HDL'S

Problem: ModelSim XE (Xilinx Edition) which is obtained for free from the Xilinx website does not support multiple HDL's.

Background: When developing the NFG, some code was provided by SRC in Verilog. When attempting to analyze the circuit with a test bench, an error occurred in ModelSim. The error stated that ModelSim XE does not support multiple HDL's.

Solution: Download ModelSim SE. NPS has a license. Details available from Dan Zulaica.

Author: T.J. Mack
Date: 26 FEB 07

H. INITIALIZING MEMORY FROM A SEPARATE FILE

Problem: Xilinx allows one to synthesize a ROM where the ROM contents are specified in a separate file. When transferring the VHDL files to the SRC and synthesizing with Synplify, an error results. This is another artifact of problem F. above.

Background: Because of the potentially large amount of data needed to load into a ROM, it is useful to have a separate file with just this data. The HDL must then access this data file during synthesis.

Solution: Problem not completely solved, yet. Some potential solutions are:

1. Below is a ROM provided by SRC Computers. Written in Verilog, (SRC Computer's preferred language) it is comprised of 32, 4-input, 1-bit output LUTs. It has a 32-bit output. It is initialized using a separate *.sdc* file.

```
module MY_ROM (
         data,
         adr
         );
     output [31:0] data;
     input [3:0] adr;

    ROM16X1 M0 (
        .O      (data[0]),
        .A0     (adr[0]),
```

```
        .A1     (adr[1]),
        .A2     (adr[2]),
        .A3     (adr[3])
        );

    ROM16X1 M1 (
        .O      (data[1]),
        .A0     (adr[0]),
        .A1     (adr[1]),
        .A2     (adr[2]),
        .A3     (adr[3])
        );
                  ...
      ***     Fill-In Remaining Modules       ***
                  ...
    ROM16X1 M31 (
        .O      (data[31]),
        .A0     (adr[0]),
        .A1     (adr[1]),
        .A2     (adr[2]),
        .A3     (adr[3])
        );

endmodule
```

The ROM initialization values are in the *.sdc* file below. The `INIT`s are somewhat cumbersome, since the LUTs are 1-bit wide. So each of the LUTs has one bit position for all of the 16 values. The `INIT` values essentially represent a 32 row by 16 column matrix. Each column represents one of 16, 32-bit outputs.

```
define_attribute {i:M0}  xc_props  "INIT=ba5d"
define_attribute {i:M1}  xc_props  "INIT=8801"
                  ...
      ***     Fill-In Missing Values    ***
                  ...
define_attribute {i:M31}  xc_props  "INIT=1321"
```

This is the most promising example of a ROM with an external file for initialization. However, the 1-bit format of the init values makes it difficult to implement.

2. Below is a another ROM example provided by SRC Computers. It uses the `RAMB16_S18_S18` module which is a 16 Kb Block RAM with two 18-bit outputs (16-bits plus 2-bits for parity). It is initialized using the `xc_props` lines within the same file.

```
module MY_ROM (
        din_0,
        dout_0,
        din_1,
        dout_1,
        adr_0,
        adr_1,
        w_en_0,
        w_en_1,
```

```
        clk
        );
    input [15:0] din_0;
    output [15:0] dout_0;
    input [15:0] din_1;
    output [15:0] dout_1;
    input [9:0] adr_0;
    input [9:0] adr_1;
    input w_en_0;
    input w_en_1;
    input clk  /*  synthesis syn_noclockbuf=1  */  ;

   RAMB16_S18_S18 M0 (
       .DOA    (dout_0[15:0]),
       .DOB    (dout_1[15:0]),
       .DOPA   (),             // ignore the parity outputs
       .DOPB   (),             // ignore the parity outputs
       .ADDRA  (adr_0),
       .ADDRB  (adr_1),
       .CLKA   (clk),
       .CLKB   (clk),
       .DIA    (din_0[15:0]),
       .DIB    (din_1[15:0]),
       .DIPA   (2'b0),         // zero the parity inputs
       .DIPB   (2'b0),         // zero the parity inputs
       .ENA    (1'b1),
       .ENB    (1'b1),
       .SSRA   (1'b0),
       .SSRB   (1'b0),
       .WEA    (w_en_0),
       .WEB    (w_en_1)
       )  /*  synthesis

xc_props="INIT_00=76931fac9dab2b36c248b87d6ae33f9a62d7183a5d5789e4b2d6b441e2411dc7,\
INIT_01=09e111c7e1e7acb6f8cac0bb2fc4c8bc2ae3baaab9165cc458e199cb89f51b13,\
INIT_02=5f7091a5abb0874df3e8cb4543a5eb93b0441e9ca4c2b0fb3d30875cbf29abd5,\
INIT_3e=1a0bf9b00ffd21b6210b11dc59ec947be86d11e10de2e980b8bc988e26aba269,\
                    ...
      ***      Fill-In Missing Values     ***
                    ...
INIT_3f=ac6bd4cd2bf0471ffcb95377922449de5393850a00a57b47800d374d961dfeb5"  */  ;

endmodule
```

3. The following code is a 16 x 32-bit ROM written in Verilog. It will synthesize in Xilinx XST, but not in Synplify PRO.

```
module romverlog(input [3:0] raddr, output [31:0] slope_int);

reg [15:0] mem [31:0];

initial
      begin
         $readmemb("memory.mem", mem);
      end

         assign slope_int = mem[raddr];

endmodule
```

The associated *memory.mem* file is a simple, binary text file with the memory initialization values.

```
00000110010001000000000000000000
00000110001011010000000000000000
00000101111111110000000000000100
00000101101110100000000000001100
00000101011000000000000000011010
00000100111100010000000000101111
00000100011100000000000001001101
00000011110111110000000001110100
00000011001111110000000010100101
00000010100100110000000011100001
00000001110111100000000100100111
00000001001000010000000101110111
00000000011000000000000111001111
00000001110111100000000100100111
00000001001000010000000101110111
00000000011000000000000111001111
```

Author: T.J. Mack
Date: 26 FEB 07

I. MACRO LATENCY AND SRC OVERHEAD

Problem: When implementing a macro, SRC requires additional clocks to accomplish *overhead* operations. The overhead appears to be 5 clock cycles to pass data *to* a macro and an additional 5 clock cycles to receive data *from* a macro. One would expect a macro with a latency of 3 to take a total of 13 clock cycles. However, it takes only 12. The last clock cycle is absorbed into the 5 clock cycles needed to receive data from the macro. In this case, the *latency* in the *info* file must be set equal to 2, even though the schematic may show a latency of 3.

Background: When developing the NFG, *pipeline depth* reports for the loop that calls the NFG macro were always 10 clock cycles more.

Solution: No solution. This is a characteristic of the SRC architecture.

Author: T.J. Mack
Date: 26 FEB 07

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX F. SRC OUTPUT

The following is *Place and Route Summary* Reports from the SRC for various functions. The function name, input domain and number of segments is shown. Also, sample input to output and total number of clock cycles is shown.

sine(pi*x) from 0 ≤ x ≤ 0.5
7 segments

```
########################################################################
###############       PLACE AND ROUTE SUMMARY    ###################
 Number of Slice Flip Flops:      4,058 out of  67,584    6%
 Number of 4 input LUTs:          1,530 out of  67,584    2%
 Number of occupied Slices:       2,425 out of  33,792    7%
 Number of MULT18X18s:                1 out of     144    1%
 freq = 90.7 MHz
########################################################################

42 clocks
Iteration 0. Sine of (14 * pi) equals 3d VHDL Signed
Iteration 1. Sine of (54 * pi) equals db VHDL Signed
Iteration 2. Sine of (15 * pi) equals 40 VHDL Signed
Iteration 3. Sine of (35 * pi) equals 9a VHDL Signed
Iteration 4. Sine of (31 * pi) equals 90 VHDL Signed
Iteration 5. Sine of (50 * pi) equals d4 VHDL Signed
Iteration 6. Sine of (1 * pi) equals 3 VHDL Signed
Iteration 7. Sine of (5b * pi) equals e5 VHDL Signed
Iteration 8. Sine of (7e * pi) equals ff VHDL Signed
Iteration 9. Sine of (19 * pi) equals 4c VHDL Signed
```

sine(πx) from 0 ≤ x ≤ 2
28 segments

```
########################################################################
###############       PLACE AND ROUTE SUMMARY    ###################
 Number of Slice Flip Flops:      4,077 out of  67,584    6%
 Number of 4 input LUTs:          1,699 out of  67,584    2%
 Number of occupied Slices:       2,513 out of  33,792    7%
 Number of MULT18X18s:                1 out of     144    1%
 freq = 92.1 MHz
########################################################################

42 clocks
Iteration 0. Sine of (194 * pi) equals 7f06 VHDL Signed
Iteration 1. Sine of (1d4 * pi) equals 7f7b VHDL Signed
Iteration 2. Sine of (15 * pi) equals 40 VHDL Signed
Iteration 3. Sine of (1b5 * pi) equals 7f33 VHDL Signed
Iteration 4. Sine of (1b1 * pi) equals 7f2b VHDL Signed
Iteration 5. Sine of (d0 * pi) equals 8d VHDL Signed
Iteration 6. Sine of (1 * pi) equals 3 VHDL Signed
```

Iteration 7. Sine of (1db * pi) equals 7f8f VHDL Signed
Iteration 8. Sine of (fe * pi) equals 5 VHDL Signed
Iteration 9. Sine of (99 * pi) equals f3 VHDL Signed

sqrt(- ln (x)) from 1/256 ≤ x ≤ 1/4
144 segments

```
##############################################################################
###############       PLACE AND ROUTE SUMMARY    ###################
  Number of Slice Flip Flops:      4,067 out of  67,584    6%
  Number of 4 input LUTs:          1,823 out of  67,584    2%
  Number of occupied Slices:       2,567 out of  33,792    7%
  Number of MULT18X18s:                1 out of     144    1%
  freq = 88.5 MHz
##############################################################################
```

Iteration 0. Sqrt(- ln (x) of 14 equals 198 VHDL Signed
Iteration 1. Sqrt(- ln (x)) of 54 equals 10a VHDL Signed
Iteration 2. Sqrt(- ln (x)) of 15 equals 194 VHDL Signed
Iteration 3. Sqrt(- ln (x)) of 35 equals 140 VHDL Signed
Iteration 4. Sqrt(- ln (x)) of 31 equals 148 VHDL Signed
Iteration 5. Sqrt(- ln (x)) of 50 equals 111 VHDL Signed
Iteration 6. Sqrt(- ln (x)) of 1 equals 255 VHDL Signed
Iteration 7. Sqrt(- ln (x)) of 5b equals fe VHDL Signed
Iteration 8. Sqrt(- ln (x)) of 7e equals c2 VHDL Signed
Iteration 9. Sqrt(- ln (x)) of 19 equals 186 VHDL Signed

sqrt(x), 0 <= x < 2, error = 0.01
488 segments

```
##############################################################################
###############       PLACE AND ROUTE SUMMARY    ###################
  Number of Slice Flip Flops:      4,060 out of  67,584    6%
  Number of 4 input LUTs:          2,594 out of  67,584    3%
  Number of occupied Slices:       2,974 out of  33,792    8%
  Number of MULT18X18s:                1 out of     144    1%
  freq = 81.3 MHz
##############################################################################
```

42 clocks
Iteration 0. Sqrt(x) 194 equals 140 VHDL Signed
Iteration 1. Sqrt(x) 1d4 equals 159 VHDL Signed
Iteration 2. Sqrt(x) 15 equals 49 VHDL Signed
Iteration 3. Sqrt(x) 1b5 equals 14d VHDL Signed
Iteration 4. Sqrt(x) 1b1 equals 14c VHDL Signed
Iteration 5. Sqrt(x) d0 equals e5 VHDL Signed
Iteration 6. Sqrt(x) 1 equals f VHDL Signed
Iteration 7. Sqrt(x) 1db equals 15b VHDL Signed
Iteration 8. Sqrt(x) fe equals fe VHDL Signed
Iteration 9. Sqrt(x) 99 equals c5 VHDL Signed

sqrt(x), 0 ≤ x < 2, error = 2 x 2^{-9}
3330 segments

```
##########################################################################
###############       PLACE AND ROUTE SUMMARY    ###################
  Number of Slice Flip Flops:       4,060 out of  67,584    6%
  Number of 4 input LUTs:           2,581 out of  67,584    3%
  Number of occupied Slices:        2,982 out of  33,792    8%
  Number of MULT18X18s:                 1 out of     144    1%
  freq = 80.9 MHz
##########################################################################

42 clocks
Iteration 0. Sine of (194 * pi) equals 140 VHDL Signed
Iteration 1. Sine of (1d4 * pi) equals 159 VHDL Signed
Iteration 2. Sine of (15 * pi) equals 49 VHDL Signed
Iteration 3. Sine of (1b5 * pi) equals 14d VHDL Signed
Iteration 4. Sine of (1b1 * pi) equals 14c VHDL Signed
Iteration 5. Sine of (d0 * pi) equals e6 VHDL Signed
Iteration 6. Sine of (1 * pi) equals 10 VHDL Signed
Iteration 7. Sine of (1db * pi) equals 15b VHDL Signed
Iteration 8. Sine of (fe * pi) equals fe VHDL Signed
Iteration 9. Sine of (99 * pi) equals c5 VHDL Signed
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX G. SYNPLICITY AREA REPORT

Below is an example of an Area Report generated by Synplicity during the synthesis process. The report shows what resources on the FPGHA are used for each of the modules of the NFG circuit. This report is for $sin(\pi x)$ where $0 \leq x < 0.5$ using 7 segments. It is interesting to note that the slope and intercept look-up module is constructed entirely with LUT's and without memory.

Other circuits including the $\sqrt{x}$ were generated with 2855 segments. The slope and intercept look-up module was still constructed entirely with LUT's.

```
#### START OF AREA REPORT #####[

Part:                    XC2V40CS144-6 (Xilinx)

---------------------------------------------------------------------
########  Utilization report for  Top level view:  signedfct  ########
=====================================================================

SEQUENTIAL ELEMENTS
*******************

Name         Total elements    Utilization    Notes
-------------------------------------------------
REGISTERS    80                100 %
LATCHES      0                 0.0 %
=================================================
Total SEQUENTIAL ELEMENTS in the block signedfct:  80 (39.02 % Utilization)


COMBINATIONAL LOGIC
*******************

Name                 Total elements    Utilization    Notes
-----------------------------------------------------------
LUTS                 50                100 %
MUXCY                15                100 %
XORCY                15                100 %
MULT18x18/MULT18x18S 1                 100 %
===========================================================
Total COMBINATIONAL LOGIC in the block signedfct:  81 (39.51 % Utilization)
```

MEMORY ELEMENTS
****

Name Total elements Number of bits Utilization Notes

SYNC RAMS 0 0 0.0 %
ROMS 0 0 0.0 %
==========
Total MEMORY ELEMENTS in the block signedfct: 0 (0.00 % Utilization)

IO PADS
****

Name Total elements Utilization Notes

PADS 31 100 %
==========
Total IO PADS in the block signedfct: 31 (15.12 % Utilization)

######## Utilization report for cell: FD16CE_MXILINX_signedfct ########
Instance path: signedfct.FD16CE_MXILINX_signedfct
==========

SEQUENTIAL ELEMENTS
****

Name Total elements Utilization Notes

REGISTERS 16 20. %
LATCHES 0 0.0 %
==========
Total SEQUENTIAL ELEMENTS in the block signedfct.FD16CE_MXILINX_signedfct: 16 (7.80 % Utilization)

COMBINATIONAL LOGIC
****

Name Total elements Utilization Notes

LUTS 0 0.0 %
MUXCY 0 0.0 %

XORCY 0 0.0 %
MULT18x18/MULT18x18S 0 0.0 %
==========

Total COMBINATIONAL LOGIC in the block signedfct.FD16CE_MXILINX_signedfct: 0 (0.00 % Utilization)

MEMORY ELEMENTS
****

Name Total elements Number of bits Utilization Notes

SYNC RAMS 0 0 0.0 %
ROMS 0 0 0.0 %
==========

Total MEMORY ELEMENTS in the block signedfct.FD16CE_MXILINX_signedfct: 0 (0.00 % Utilization)

IO PADS
****

Name Total elements Utilization Notes

PADS 0 0.0 %
==========

Total IO PADS in the block signedfct.FD16CE_MXILINX_signedfct: 0 (0.00 % Utilization)

######## Utilization report for cell: FD16CE_MXILINX_signedfct_1 ########
Instance path: signedfct.FD16CE_MXILINX_signedfct_1
==========

SEQUENTIAL ELEMENTS
****

Name Total elements Utilization Notes

REGISTERS 16 20. %
LATCHES 0 0.0 %
==========

Total SEQUENTIAL ELEMENTS in the block signedfct.FD16CE_MXILINX_signedfct_1: 16 (7.80 % Utilization)

COMBINATIONAL LOGIC
****

Name	Total elements	Utilization	Notes
LUTS	0	0.0 %	
MUXCY	0	0.0 %	
XORCY	0	0.0 %	
MULT18x18/MULT18x18S	0	0.0 %	

Total COMBINATIONAL LOGIC in the block signedfct.FD16CE_MXILINX_signedfct_1: 0 (0.00 % Utilization)

MEMORY ELEMENTS
****

Name	Total elements	Number of bits	Utilization	Notes
SYNC RAMS	0	0	0.0 %	
ROMS	0	0	0.0 %	

Total MEMORY ELEMENTS in the block signedfct.FD16CE_MXILINX_signedfct_1: 0 (0.00 % Utilization)

IO PADS
****

Name	Total elements	Utilization	Notes
PADS	0	0.0 %	

Total IO PADS in the block signedfct.FD16CE_MXILINX_signedfct_1: 0 (0.00 % Utilization)

######## Utilization report for cell: FD16CE_MXILINX_signedfct_2 ########
Instance path: signedfct.FD16CE_MXILINX_signedfct_2

SEQUENTIAL ELEMENTS
****

Name	Total elements	Utilization	Notes

REGISTERS 16 20. %
LATCHES 0 0.0 %
==========
Total SEQUENTIAL ELEMENTS in the block signedfct.FD16CE_MXILINX_signedfct_2: 16 (7.80 % Utilization)

COMBINATIONAL LOGIC
****

Name Total elements Utilization Notes

LUTS 0 0.0 %
MUXCY 0 0.0 %
XORCY 0 0.0 %
MULT18x18/MULT18x18S 0 0.0 %
==========
Total COMBINATIONAL LOGIC in the block signedfct.FD16CE_MXILINX_signedfct_2: 0 (0.00 % Utilization)

MEMORY ELEMENTS
****

Name Total elements Number of bits Utilization Notes

SYNC RAMS 0 0 0.0 %
ROMS 0 0 0.0 %
==========
==========
Total MEMORY ELEMENTS in the block signedfct.FD16CE_MXILINX_signedfct_2: 0 (0.00 % Utilization)

IO PADS
****

Name Total elements Utilization Notes

PADS 0 0.0 %
==========
Total IO PADS in the block signedfct.FD16CE_MXILINX_signedfct_2: 0 (0.00 % Utilization)

######## Utilization report for cell: FD16CE_MXILINX_signedfct_3 ########

Instance path: signedfct.FD16CE_MXILINX_signedfct_3
==========

SEQUENTIAL ELEMENTS
****

Name	Total elements	Utilization	Notes
REGISTERS	16	20. %	
LATCHES	0	0.0 %	

==========
Total SEQUENTIAL ELEMENTS in the block signedfct.FD16CE_MXILINX_signedfct_3: 16 (7.80 % Utilization)

COMBINATIONAL LOGIC
****

Name	Total elements	Utilization	Notes
LUTS	0	0.0 %	
MUXCY	0	0.0 %	
XORCY	0	0.0 %	
MULT18x18/MULT18x18S	0	0.0 %	

==========
Total COMBINATIONAL LOGIC in the block signedfct.FD16CE_MXILINX_signedfct_3: 0 (0.00 % Utilization)

MEMORY ELEMENTS
****

Name	Total elements	Number of bits	Utilization	Notes
SYNC RAMS	0	0	0.0 %	
ROMS	0	0	0.0 %	

==========
Total MEMORY ELEMENTS in the block signedfct.FD16CE_MXILINX_signedfct_3: 0 (0.00 % Utilization)

IO PADS
****

Name	Total elements	Utilization	Notes

PADS 0 0.0 %
==========
Total IO PADS in the block signedfct.FD16CE_MXILINX_signedfct_3: 0 (0.00 % Utilization)

######## Utilization report for cell: FD16CE_MXILINX_signedfct_4 ########
Instance path: signedfct.FD16CE_MXILINX_signedfct_4
==========

SEQUENTIAL ELEMENTS
****

Name Total elements Utilization Notes

REGISTERS 16 20. %
LATCHES 0 0.0 %
==========
Total SEQUENTIAL ELEMENTS in the block signedfct.FD16CE_MXILINX_signedfct_4: 16 (7.80 % Utilization)

COMBINATIONAL LOGIC
****

Name Total elements Utilization Notes

LUTS 0 0.0 %
MUXCY 0 0.0 %
XORCY 0 0.0 %
MULT18x18/MULT18x18S 0 0.0 %
==========
Total COMBINATIONAL LOGIC in the block signedfct.FD16CE_MXILINX_signedfct_4: 0 (0.00 % Utilization)

MEMORY ELEMENTS
****

Name Total elements Number of bits Utilization Notes

SYNC RAMS 0 0 0.0 %
ROMS 0 0 0.0 %
==========
Total MEMORY ELEMENTS in the block signedfct.FD16CE_MXILINX_signedfct_4: 0 (0.00 % Utilization)

IO PADS
****

Name Total elements Utilization Notes

PADS 0 0.0 %
==========
Total IO PADS in the block signedfct.FD16CE_MXILINX_signedfct_4: 0 (0.00 % Utilization)

######## Utilization report for cell: adder16x16s ########
Instance path: signedfct.adder16x16s
==========

SEQUENTIAL ELEMENTS
****

Name Total elements Utilization Notes

REGISTERS 0 0.0 %
LATCHES 0 0.0 %
==========
Total SEQUENTIAL ELEMENTS in the block signedfct.adder16x16s: 0 (0.00 % Utilization)

COMBINATIONAL LOGIC
****

Name Total elements Utilization Notes

LUTS 16 32. %
MUXCY 15 100 %
XORCY 15 100 %
MULT18x18/MULT18x18S 0 0.0 %
==========
Total COMBINATIONAL LOGIC in the block signedfct.adder16x16s: 46 (22.44 % Utilization)

MEMORY ELEMENTS
****

Name Total elements Number of bits Utilization Notes

SYNC RAMS 0 0 0.0 %
ROMS 0 0 0.0 %
==========
Total MEMORY ELEMENTS in the block signedfct.adder16x16s: 0 (0.00 % Utilization)

IO PADS
****

Name Total elements Utilization Notes

PADS 0 0.0 %
==========
Total IO PADS in the block signedfct.adder16x16s: 0 (0.00 % Utilization)

######## Utilization report for cell: mult16x16s ########
Instance path: signedfct.mult16x16s
==========

SEQUENTIAL ELEMENTS
****

Name Total elements Utilization Notes

REGISTERS 0 0.0 %
LATCHES 0 0.0 %
==========
Total SEQUENTIAL ELEMENTS in the block signedfct.mult16x16s: 0 (0.00 % Utilization)

COMBINATIONAL LOGIC
****

Name Total elements Utilization Notes

LUTS 0 0.0 %
MUXCY 0 0.0 %
XORCY 0 0.0 %
MULT18x18/MULT18x18S 1 100 %
==========
Total COMBINATIONAL LOGIC in the block signedfct.mult16x16s: 1 (0.49 % Utilization)

MEMORY ELEMENTS
****

Name Total elements Number of bits Utilization Notes

SYNC RAMS 0 0 0.0 %
ROMS 0 0 0.0 %
==========
Total MEMORY ELEMENTS in the block signedfct.mult16x16s: 0 (0.00 % Utilization)

IO PADS
****

Name Total elements Utilization Notes

PADS 0 0.0 %
==========
Total IO PADS in the block signedfct.mult16x16s: 0 (0.00 % Utilization)

######## Utilization report for cell: slopeintlu ########
Instance path: signedfct.slopeintlu
==========

SEQUENTIAL ELEMENTS
****

Name Total elements Utilization Notes

REGISTERS 0 0.0 %
LATCHES 0 0.0 %
==========
Total SEQUENTIAL ELEMENTS in the block signedfct.slopeintlu: 0 (0.00 % Utilization)

COMBINATIONAL LOGIC
****

Name Total elements Utilization Notes

LUTS 34 68. %
MUXCY 0 0.0 %

XORCY 0 0.0 %
MULT18x18/MULT18x18S 0 0.0 %
==========

Total COMBINATIONAL LOGIC in the block signedfct.slopeintlu: 34 (16.59 % Utilization)

MEMORY ELEMENTS
****

Name Total elements Number of bits Utilization Notes

SYNC RAMS 0 0 0.0 %
ROMS 0 0 0.0 %
==========

Total MEMORY ELEMENTS in the block signedfct.slopeintlu: 0 (0.00 % Utilization)

IO PADS
****

Name Total elements Utilization Notes

PADS 0 0.0 %
==========

Total IO PADS in the block signedfct.slopeintlu: 0 (0.00 % Utilization)

END OF AREA REPORT #####]

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] B. Parhami, *Computer Arithmetic : Algorithms and Hardware Designs*. New York: Oxford Univeristy Press, 2000.

[2] T. Sasao, J.T. Butler, and M. Riedel, "Application of LUT cascades to numerical function generators," *The 12th Workshop on Synthesis And System Integration of Mixed Information Technologies 2004*, Kanazawa, Japan, October 18-19, 2004.

[3] S. Nagayama, T. Sasao, and J. T. Butler, "Numerical function generators using edge-valued binary decision diagrams," *Proc. of the 12th Asian-South Pacific Design Automation Conference (ASP-DAC) 2007*, pp. 535-540, Yokohama, Japan, January 23-26, 2007.

[4] C.L. Frenzen, T. Sasao, and J.T. Butler, "The tradeoff between memory size and approximation error in numeric function generators based on table lookup," preprint, February 25, 2007.

[5] C.L. Frenzen, T. Sasao and J.T. Butler, "A direct design method for the segment index encoder," preprint, February 2007.

[6] S. Nagayama, T. Sasao, and J. T. Butler, "Compact numerical function generators based on quadratic approximation: Architecture and synthesis method", IEICE Trans. Fundamentals, (Special Section on VLSI Design and CAD Algorithms). Vol. E89-A, No. 12, pp. 3510-3518, December 2006.

[7] J. Cao, B. Wei, J. Cheng, "High-performance architectures of elementary function generation," *Proc. of the 15th IEEE Symposium on Computer Arithmetic,"* pp. 136-144, IEEE Computer Society, Washington, DC, 2001.

[8] The Math Works Inc., MATLAB On-line Help, Release 14 with Service Pack 1, The Math Works Inc. Natick, MA, September 13, 2004.

[9] Xilinx Inc., "Xilinx ISE 6 Software Manuals," UGD00 (v3.4.1), Xilinx Inc., San Jose, CA, February 25, 2003.

[10] Xilinx Inc., *Xilinx ISE 6 Help Menu*, Xilinx Project Navigator Ver. 6.3.03.i, Xilinx Inc., San Jose, CA, 2004.

[11] Mentor Graphics, Corp., *ModelSim SE 6.2E Help Menu*., ModelSime SE 6.2E, Mentor Graphics, Corp., Wilsonville , OR, November 16, 2006.

[12] Synplicity, Inc., "Synplicity FPGA Synthesis User Guide," Synplcity Inc., Sunnyvale, CA, December 2005.

[13] Synplicity, Inc., *Synplicity PRO 8.5.1 Help Menu*, Synplicity PRO 8.5.1, Synplicity, Inc., Sunnyvale, CA, 2006.

[14] Bhasker, J., *A VHDL Primer*, third edition. Upper Saddle River, NJ: Prentice Hall, 1999.

[15] Yalamanchili, S., *Introductory VHDL from Simulation to Synthesis*. Upper Saddle River, NJ: Prentice Hall, 2001.

[16] The Institute of Electrical and Electronics Engineers, Inc, *IEEE Standard VHDL Language Reference Manual*, IEEE Std 2076, 2000 Edition, IEEE, Inc., New York, NY, December 29, 2000.

[17] SRC Computers Inc., "SRC-6 C Programming Environment V2.2 Guide," SRC-007-18, SRC Computers Inc., Colorado Springs, Colorado, August 21, 2006.

[18] Xilinx Inc., "Virtex-II Platform FPGA: Complete Data Sheet," DS031 (v3.4), Xilinx Inc., San Jose, CA, March 1, 2005.

[19] Microsoft, Corp., *Microsoft Visual Studio 2005 On-Line Help*, Microsoft Visual Studio 2005 Professional Edition, ver. 8.0.50727.42, Microsft, Corp. Redmond, WA, 2005.

[20] D. Caliga, (private communication), SRC Computers, Inc., 2005.

[21] N. Macaria, "Title TBD," Master's Thesis, Naval Postgraduate School, Monterey, CA, preprint, September 2007.

[22] SRC Computers Inc., "SRC Carte™ Training Exercises Release 2.1," SRC-007-16, SRC Computers Inc., Colorado Springs, CO, August 31, 2005.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Chairman, Code EC
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

4. Prof. Jon T. Butler
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

5. Prof. Herschel H. Loomis
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

6. LCDR Thomas J. Mack
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

7. Mr. Alan Hunsberger
   National Security Agency
   Ft. Meade, Maryland

8. LT Njuguna Macaria
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

9. Prof. Christopher L. Frenzen,
   Department of Applied Math
   Naval Postgraduate School
   Monterey, California