HIGH-SPEED NUMERIC FUNCTION GENERATOR USING PIECEWISE QUADRATIC APPROXIMATIONS

by

Njuguna Macaria

September 2007

Thesis Advisor: Jon T. Butler
Thesis Co-Advisors: Herschel H. Loomis, Christopher L. Frenzen

Approved for public release; distribution is unlimited


**1. AGENCY USE ONLY (Leave blank)**

**2. REPORT DATE**

September 2007

**3. REPORT TYPE AND DATES COVERED**

Master’s Thesis

**4. TITLE AND SUBTITLE**

High-Speed Numeric Function Generator, Using Piecewise Quadratic Approximations

**5. FUNDING NUMBERS**

N/A

**6. AUTHOR(S)**

Njuguna Macaria

**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)**

The CORDIC algorithm is an accurate way to compute the value of a function like \( \sin(x) \), for a given value of \( x \). However, it is iterative and slow. In this thesis, we show that a wide class of arithmetic functions can be realized on the SRC-6, a reconfigurable computer, using polynomial approximations. The function is realized by partitioning its domain into segments and then approximating the function in each segment by a quadratic polynomial. This is not an iterative approach, and so it is faster than the CORDIC algorithm.

Two approximation methods are implemented. In one method, non-uniform segments are used. Here, larger segments can be used where the function is close to quadratic, while highly non-quadratic regions require smaller segments. This approach minimizes the number of segments. In the other method, uniform segments are used. Although more segments are needed than in the non-uniform method, the circuit is simpler.

We show that accuracies of up to 33 bits are possible. A pipelined circuit was built on the SRC-6 in two’s complement and floating point. We also show an efficient algorithm for segmenting the function, which is faster than previous methods.

**14. SUBJECT TERMS**

Numerical Function Generator, Piecewise Quadratic Approximation, Field Programmable Gate Array (FPGA), Reconfigurable Computer, VHSIC Hardware Description Language, Computer Arithmetic.

**15. NUMBER OF PAGES**

216

**16. PRICE CODE**


---

The CORDIC algorithm is an accurate way to compute the value of a function like \( \sin(x) \), for a given value of \( x \). However, it is iterative and slow. In this thesis, we show that a wide class of arithmetic functions can be realized on the SRC-6, a reconfigurable computer, using polynomial approximations. The function is realized by partitioning its domain into segments and then approximating the function in each segment by a quadratic polynomial. This is not an iterative approach, and so it is faster than the CORDIC algorithm.

Two approximation methods are implemented. In one method, non-uniform segments are used. Here, larger segments can be used where the function is close to quadratic, while highly non-quadratic regions require smaller segments. This approach minimizes the number of segments. In the other method, uniform segments are used. Although more segments are needed than in the non-uniform method, the circuit is simpler.

We show that accuracies of up to 33 bits are possible. A pipelined circuit was built on the SRC-6 in two’s complement and floating point. We also show an efficient algorithm for segmenting the function, which is faster than previous methods.
ABSTRACT

The CORDIC algorithm is an accurate way to compute the value of a function like \( \sin(x) \), for a given value of \( x \). However, it is iterative and slow. In this thesis, we show that a wide class of arithmetic functions can be realized on the SRC-6, a reconfigurable computer, using polynomial approximations. The function is realized by partitioning its domain into segments and then approximating the function in each segment by a quadratic polynomial. This is not an iterative approach, and so it is faster than the CORDIC algorithm.

Two approximation methods are implemented. In one method, non-uniform segments are used. Here, larger segments can be used where the function is close to quadratic, while highly non-quadratic regions require smaller segments. This approach minimizes the number of segments. In the other method, uniform segments are used. Although more segments are needed than in the non-uniform method, the circuit is simpler.

We show that accuracies of up to 33 bits are possible. A pipelined circuit was built on the SRC-6 in two’s complement and floating point. We also show an efficient algorithm for segmenting the function, which is faster than previous methods.
# TABLE OF CONTENTS

## I. INTRODUCTION

A. PROBLEM STATEMENT AND PURPOSE ...........................................1
B. IMPLEMENTATION OVERVIEW .....................................................3
C. THESIS ORGANIZATION ...............................................................5

## II. FUNCTION APPROXIMATION

A. QUADRATIC VS LINEAR ...............................................................7
B. SEGMENTATION ............................................................................9
   1. Uniform and Non-Uniform Segmentation ...................................9
      a. Summary of Advantages and Disadvantages of Uniform
         and Non-Uniform Segmentation ............................................15
   2. Segment Coefficients Using Polyfit and the Remez Algorithm ....16
   3. Algorithms Investigated to Speed-Up the Segmentation ..........19
      a. Brute Force .............................................................................20
      b. Binary Search .........................................................................20
      c. Divide by Thirds .....................................................................23
      d. Increment by Ratio Numbers ..................................................25
      e. Estimated Segment Widths (1, 2, 3, more and Average) .......26
      f. Hybrid of Thirds and 3 Estimates ..........................................32
C. MATLAB RESULTS ........................................................................34
D. SUMMARY ......................................................................................35

## III. NFG CIRCUIT

A. CIRCUIT OVERVIEW .................................................................37
   1. Number System .........................................................................38
   2. 16, 32, 64 Bit Accuracy vs. 16, 32, 64 Bit Architecture .............40
B. CIRCUIT COMPONENTS ...............................................................40
   1. Segment Index Encoder ............................................................40
   2. Indexing in Uniform Segmentation ...........................................42
      a. Floating Point Implementation ..............................................43
      b. Fixed Point Implementation ..................................................43
   3. Coefficients Table .................................................................44
   4. Multiplier ..................................................................................44
      a. Floating Point Multiplier ......................................................45
      b. Two’s Complement Fixed Point Multiplier ..........................45
   5. Adder .......................................................................................46
C. SUMMARY ....................................................................................47

## IV. SRC BACKGROUND

A. INTRODUCTION ............................................................................49
B. HARDWARE ..................................................................................50
C. SOFTWARE CODE .........................................................................51
   1. main.c .......................................................................................51
2. `<subroutine>.mc` .................................................................52
3. Makefile ...........................................................................52
4. Macros...............................................................................52
   a. info...........................................................................53
   b. blk.v...........................................................................53
   c. HDL Files....................................................................53
   d. Location for NGO Directory......................................53
D. SUMMARY ..............................................................................54
V. IMPLEMENTATION RESULTS..............................................55
A. UNIFORM SEGMENTATION..................................................55
  1. Floating Point Implementation.....................................55
  2. Fixed Point Implementation..........................................60
B. NON-UNIFORM SEGMENTATION........................................65
  1. Floating Point Implementation.....................................65
  2. Fixed Point Implementation..........................................69
     a. No Macro Multiplier (non-uniform)..........................70
     b. Macro Multiplier Implementation..............................70
C. SOURCES OF ERROR..........................................................71
  1. Function Approximation..............................................71
  2. Absence of Rounding in the Multiplier.........................71
  3. Insufficient Bits ................................................................71
D. SUMMARY ..............................................................................72
VI. CONCLUSION ..........................................................................73
A. SUMMARY OF WORK..........................................................73
B. SUGGESTED FUTURE WORK................................................74
  1. Hybrid of Uniform and Non-Uniform Segmentation...........74
  2. Expand the Domain of the NFG via Mapping................75
  3. Build an HDL Multiplier Macro and Tap of Desired Bits....75
  4. Build a Rounding Macro..................................................75
  5. Efficient Segment Index Encoder vice Priority Selector Macros..75
  6. Different Architecture......................................................75
APPENDIX A. MATLAB ALGORITHMS........................................79
A.1 QUADRATIC APPROXIMATION USING POLYFIT..............79
A.2 QUADRATIC APPROXIMATION USING REMEZ ALGORITHM....93
   A.2.1 Remez Algorithm With Chebyshev Initial Points............99
   A.2.1 Variable Length Approximation Speed-Up Algorithms......103
      a. Hybrid of 3 estimates, average and thirds..................103
      b. Binary Search.........................................................108
      c. Thirds....................................................................109
      d. Ratios......................................................................110
      e. 1 estimate................................................................111
      f. 2 estimates .............................................................112
      g. 3 estimates .............................................................113
   A.2.2 Non-Uniform Quadratic Approximation.......................114
A.2.3 Uniform Quadratic Approximation ...............................................116
A.2.4 Uniform Quadratic Approximation with Constraints .................117
A.2.5 Fixed-Point Decimal to HEXADECIMAL or BINARY .............120
A.2.6 User Interface and Function Information Files .................121

APPENDIX B. HDL CODE ......................................................................................125
B.1 MULTIPLIER CODE .................................................................................125
1. VHDL ................................................................................................125
2. Verilog ...............................................................................................135

APPENDIX C. SRC C CODE ...........................................................................141
C.1 UNIFORM SEGMENTATION ................................................................141
1. Floating Point ...................................................................................141
   a. Main.c ..............................................................................141
   b. subr.mc ........................................................................144
   c. Sample memory file (memD13.mem) ...................................146
2. Fixed Point ......................................................................................146
   a. Main.c ...........................................................................146
   b. subr.mc ........................................................................149
C.2 NON-UNIFORM SEGMENTATION ........................................................152
1. Floating Point ...................................................................................152
   a. Main.c ...........................................................................152
   b. subr.mc ........................................................................154
2. Fixed Point ......................................................................................157
   a. Main.c ...........................................................................157
   b. subr.mc ........................................................................159
3. Fixed Point with Macro ................................................................164
   a. Makefile ........................................................................164
   b. subr.mc ........................................................................166
   c. blk.v .............................................................................168
   d. info .................................................................................169

APPENDIX D. COPY OF PROFILE REPORT .....................................................171

APPENDIX E. LESSONS LEARNED .................................................................177
E.1 FILE NAMING PROBLEMS ................................................................177
E.2 USING THE const CONSTRUCT IN C .................................................177
E.3 INCORRECT ARGUMENTS IN SYSTEM SUPPLIED MACROS ....178
E.4 IF / THEN / ELSE LIMITATION ...........................................................179
E.5 MULTIPLE FILES USED IN A MACRO ............................................179
E.6 XILINX / SYNLIFY INCONSISTENCIES ...........................................179
E.7 MODELSIM AND MULTIPLE HDL’S ................................................180
E.8 INITIALIZING MEMORY FROM A SEPARATE FILE .....................180
E.9 MACRO LATENCY AND SRC OVERHEAD ....................................183
E.10 CANNOT USE PRIORITY SELECTOR GREATER THAN 128 ....183
LIST OF FIGURES

Figure 1. Numeric function generator (NFG) architecture................................................3
Figure 2. Quadratic segmentation of $\sqrt{\ln(x)}$ shows the difference in the size of segments due to curvature of the function ......................................................10
Figure 3. Segment error of $\sqrt{\ln(x)}$ when $\varepsilon = 2^{-16}$ ..............................................11
Figure 4. Quadratic uniform segmentation for $\sqrt{\ln(x)}$ when limited when $\varepsilon = 2^{-16}$ ...12
Figure 5. Uniform segmentation error for $\cos(\pi x)$ when limited by $\varepsilon = 2^{-17}$. ............13
Figure 6. Error for non-uniform segmentation for $\cos(\pi x)$ when limited by $\varepsilon = 2^{-17}$ ...13
Figure 7. Quadratic approximation user-interface when non-uniform segmentation has been used. ..................................................................................................14
Figure 8. Quadratic approximation user-interface when uniform segmentation has been specified. ..................................................................................................15
Figure 9. Quadratic non-uniform segmentation approximation error using $\text{Polyfit}$. ......17
Figure 10. Quadratic non-uniform segmentation approximation error using $\text{Remez}$. (Only the first four segments are shown). ......................................................18
Figure 11. Shows the interval and segmentation notation ....................................................21
Figure 12. Visual aid for description of divide by thirds algorithm .......................................25
Figure 13. IMPLICIT+EXPLICIT™ architecture [16]. .........................................................49
Figure 14. MAP® Hardware overview diagram [18]. ..........................................................50
Figure 15. NFG Pipeline depth and place and route summary. ............................................55
Figure 16. Pipeline depth (NFG and SRC Cosine Macro). Place and route summary. ...56
Figure 17. Pipeline depth (NFG and SRC $\sqrt{\ln(x)}$ implemented in macros). Place and route summary with subtraction hardware included for computing offset (when finding the index of coefficients). ........................................57
Figure 18. Results from Uniform Segmentation NFG compared with SRC Cosine Macro, MATLAB and Excel. .................................................................59
Figure 19. Uniform Segmentation of $2^x$, $N=1,000,000$ and $\varepsilon = 2^{-24}$ ...............62
Figure 20. NFG and macro both built on the FPGA for numeric function; $\frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}}$. .....67
Figure 21. NFG built on the FPGA for numeric function; $\frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}}$ ..............................68
Figure 22. Horner’s rule NFG architecture overview ..........................................................77
### LIST OF TABLES

Table 1. Suite of numeric functions and their domains ................................................................. 4
Table 2. Segmentation required for linear and quadratic approximations ................................. 8
Table 3. Summary of Advantages and Disadvantages of Uniform and Non-uniform Segmentation .................................................................................................................. 16
Table 4. Various methods show the number of calls to the function chebyRemz; segmentation of $\sqrt{-\ln(x)}$, $\epsilon = 2^{-17}$ and various values of N ........................................ 23
Table 5. Comparison of “3 estimates”, mean of all estimates computed on proposed segment that was calculated after taking 3 estimates; “3 average” and a hybrid that exaggerates the approximation error by 5%. All cases, $N=100,000$ and $\epsilon = 2^{-17}$ ................................................................................................................................. 29
Table 6. Profile Report for $-\left( x \log_x x + (1-x) \log_2 (1-x) \right)$, $N=1,000,000$ and $\epsilon = 2^{-33}$. Shows 44.438s for the `varQuadApprox` function that averages only three estimates ....................................................................................................................................................... 30
Table 7. Profile Report for $-\left( x \log_x x + (1-x) \log_2 (1-x) \right)$, $N=1,000,000$ and $\epsilon = 2^{-33}$. Shows 20.078s for the `varQuadApprox` function and 0.061s for the average of all the estimates on the entire segment ................................................................................................................................. 32
Table 8. Sample memory-files (Decimal and Hexadecimal). Non-uniform segmentation of $\cos(\pi x)$, $N=1,000,000$ and $\epsilon = 2^{-33}$ ......................................................................................................................................................................................... 35
Table 9. Maximum and minimum values encountered for each function in the NFG computation. Last column is the number of bits required for the integer portion ........................................................................................................................................................................... 39
Table 10. Code that uses two selectors to implement 48 segments ................................................ 41
Table 11. Comparison of NFG uniform segmentation and macros: NFG alone, Macro alone and both (function is $\cos(\pi x)$). Implementations without offset ................................................................................................................................................................................. 57
Table 12. Number of segments required for Uniform Segmentation computed with $N=1,000,000$ for various values of $\epsilon$ ................................................................................................................................. 60
Table 13. Fixed point implementation of $2^x$, no bit shifts, $N=1,000,000$ and $\epsilon = 2^{-24}$ .... 61
Table 14. Fixed point, uniform segmentation of $\sqrt{-\ln(x)}$, multiplier operands shifted by 8 bits, $N=1,000,000$ and $\epsilon = 2^{-24}$ ................................................................. 63
Table 15. Pipeline depth and hardware resources for uniform implementation with no adjustments ................................................................................................................................................................................. 64
Table 16. Comparison of uniform segmentation NFG between fixed point and floating point ............................................................................................................................................................................. 65
Table 17. Pipeline depth for various implementations of using the available macros or the NFG in floating point number system ................................................................................................................................. 66
Table 18. Comparison between SRC macro and NFG; numeric function $\frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$, N=1,000,000 and $\varepsilon = 2^{-24}$ .................................................................69

Table 19. Pipeline depth, place and route summary for $\sqrt{-\ln(x)}$, N=1,000,000 and $\varepsilon = 2^{-24}$. Non-uniform segmentation using priority selector macro. ........70

Table 20. Speed-up in computation time for 15 functions (expressed as a percentage of the time needed when the domain is divided into 1,000,000 points) for $\varepsilon = 2^{-24}$ ........................................................................................................73
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
CSA  Carry Save Adder
CLAH  Carry Look Ahead Adder
DEL  Direct Execution Logic
DLD  Dense Logic Device
DSP  Digital Signal Processing
ECS  Engineering Capture System
EVBDD  Edge-Valued Binary Decision Diagram
FPGA  Field Programmable Gate Array
HDL  Hardware Description Language
I/O  Input / Output
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
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM</td>
<td>Read Only Memory</td>
</tr>
<tr>
<td>SRC</td>
<td>Seymour R Cray</td>
</tr>
<tr>
<td>USN</td>
<td>United States Navy</td>
</tr>
<tr>
<td>Verilog</td>
<td>A C-Based HDL</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>VHSIC</td>
<td>Very High Speed Integrated Circuit</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

I would like to thank my wife, Mary, and my boys, Damian and Jimmy, for being great sports while I spent many hours ignoring them to complete my thesis.

I would also like to thank my advisors, Prof. Jon Butler, Prof. Herschel Loomis and Prof. Chris Frenzen. I thoroughly enjoyed my classroom experiences, but nothing can replace all the enthusiasm and support I received from my advisors while working on this thesis. Thanks.

Also, thanks 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.
EXECUTIVE SUMMARY

This thesis focuses on the high-speed implementation of arithmetic functions, such as $\sin(\pi x)$, $\ln(x)$ and $2^x$. Meteorological computations, scientific calculations and graphics are applications that require fast mathematical computation.

The CORDIC algorithm and Taylor series expansion are methods used to compute trigonometric functions. The CORDIC algorithm is hardware efficient, precise, but iterative in design and therefore slow.

In this thesis, we investigate a way to speed up mathematical computations by using piecewise quadratic approximations built on reconfigurable hardware. The function is realized by partitioning its domain into segments and then approximating the function in each segment by a quadratic polynomial. This is not an iterative approach, and so it is faster than the CORDIC algorithm.

The reconfigurable hardware used is the SRC-6E that is designed by SRC Computers in Colorado Springs, Colorado.

The objectives were to:

- Find an efficient algorithm to segment any numeric function using piecewise quadratic approximations.
- Find an accurate segmentation (accurate when evaluated using the approximation polynomial) to any numeric function given an accuracy constraint in terms of number of bits.
- Design pipelined hardware for the Numeric Function Generator (NFG) with a small pipeline depth (compared to what is currently available).
- Design NFG to operate at 100MHz or faster on the FPGA.

Segmentation is a preliminary step to provide a memory file that contains the number of segments for the numeric function, and each segment’s coefficients needed to compute the approximation polynomial.
MATLAB is used to segment any function over a defined interval. The MATLAB program needs to know the function, interval, desired accuracy and the number of discrete points in the interval. The MATLAB built-in function, *Polyfit*, was used to compute the coefficients of the approximation polynomial, but analysis showed that the approximation computed using this method did not efficiently segment the function. *Polyfit* is computationally fast, but results in an inefficiently segmented function.

The *Remez* algorithm is used to efficiently segment the numeric function. The *Remez* algorithm evenly distributes the approximation error on each segment, but is computationally intensive and slow. Several methods were investigated to speed up the algorithm. The best method to speed up the program, involved a hybrid of three methods.

- Segment width estimation that requires the third derivative of the numeric function and the accuracy desired by the user.
- Search algorithm similar to a binary search
- Single stepping through points and testing to determine if the accuracy has been met.

The program computes an estimated segment width and a metric is used to determine the quality of the estimation. If the metric indicates the estimation quality is poor, then the program will use the search algorithm to get closer to the optimum width. In the final step, the program single steps through the points and tests each approximation to determine when the accuracy has been met. When the segmentation of the function is complete, the optimum segment width and the associated coefficients are saved in a memory file for use in the NFG.

The segmentation algorithm sped up the program tremendously. If the domain is divided into over a million points, the original program would take at least one million tests to segment a function. In each test, the program computes the coefficients and tests the polynomial against the numeric function to see if the accuracy is met. When the speed up algorithm is used, the program requires much less than 0.1% of the number of tests than without the speed up. Table 1 shows the results when 15 functions were tested.

xx
The interval is shown in the second column, the speed up is shown in percentage format in the third column and the last column shows the number of segments. The percentage is computed as: $\frac{\text{# of tests} \times 100}{1,000,000}$.

<table>
<thead>
<tr>
<th>Function</th>
<th>Interval</th>
<th>%Of tests</th>
<th># of Segments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^x$</td>
<td>[0,1]</td>
<td>0.00910</td>
<td>35</td>
</tr>
<tr>
<td>$1/x$</td>
<td>[1,2]</td>
<td>0.01020</td>
<td>50</td>
</tr>
<tr>
<td>$\sqrt{x}$</td>
<td>[1,2]</td>
<td>0.00750</td>
<td>24</td>
</tr>
<tr>
<td>$1/\sqrt{x}$</td>
<td>[1,2]</td>
<td>0.00720</td>
<td>36</td>
</tr>
<tr>
<td>$\log_2(x)$</td>
<td>[1,2]</td>
<td>0.00900</td>
<td>44</td>
</tr>
<tr>
<td>$\log(x)$</td>
<td>[1,2]</td>
<td>0.00780</td>
<td>39</td>
</tr>
<tr>
<td>$\sin(\pi x)$</td>
<td>[0,1/2]</td>
<td>0.01990</td>
<td>58</td>
</tr>
<tr>
<td>$\cos(\pi x)$</td>
<td>[0,1/2]</td>
<td>0.01740</td>
<td>58</td>
</tr>
<tr>
<td>$\tan(\pi x)$</td>
<td>[0,1/4]</td>
<td>0.01240</td>
<td>58</td>
</tr>
<tr>
<td>$\sqrt{-\log(x)}$</td>
<td>[1/512,1/4]</td>
<td>0.04070</td>
<td>163</td>
</tr>
<tr>
<td>$\tan(\pi x)^{\cdot\cdot}$</td>
<td>[0,1/4]</td>
<td>0.02180</td>
<td>79</td>
</tr>
<tr>
<td>$-(x*\log_2(x))$</td>
<td>[1/256,1-1/256]</td>
<td>0.04710</td>
<td>183</td>
</tr>
<tr>
<td>$1/(1+\exp(-x))$</td>
<td>[0,1]</td>
<td>0.00920</td>
<td>20</td>
</tr>
<tr>
<td>$(1/\sqrt(2*\pi))$</td>
<td>[0,sqrt(2)]</td>
<td>0.01670</td>
<td>45</td>
</tr>
<tr>
<td>$\sin(\exp(x))$</td>
<td>[0,2]</td>
<td>0.07810</td>
<td>265</td>
</tr>
</tbody>
</table>

Table 1. Speed-up in computation time for 15 functions (expressed as a percentage of the time needed when the domain is divided into 1,000,000 points) for $\varepsilon = 2^{-24}$.

The NFG circuit consists of three multipliers, one 3-input adder, a segment indexing method and the memory that contains the approximation polynomials’ coefficients for each segment.

Figure 1 is a block diagram that shows an overview of the NFG circuit.
Two approximation methods are implemented. In one method, non-uniform segments are used. Here, larger segments can be used where the function is close to quadratic, while highly non-quadratic regions require smaller segments. This approach minimizes the number of segments. In the other method, uniform segments are used. Although more segments are needed than in the non-uniform method, the circuit is simpler.

We show that accuracies of up to 33 bits are possible. A pipelined circuit was built on the SRC-6 in two’s complement and floating point. The floating point implementation is easier to program via the interface that SRC provides.

![Numeric function generator (NFG) architecture.](image-url)
<subroutine>.mc file is a C-like file that is compiled into the hardware that resides on the FPGAs in the SRC Multi-Adaptive Programming (MAP) board.

Using fixed point implementation produces a shorter pipeline depth (approximately 30% of the floating point pipeline depth), but requires more effort by the programmer to ensure the bits are aligned correctly. In fixed point implementation, the bits are truncated instead of rounded. This introduces errors in the intermediate computations that propagate to the final answer.

The best solution to this problem is to build a user macro multiplier that takes care of the rounding and ensures the bits are aligned in the intermediate results of the polynomial computation.
I. INTRODUCTION

A. PROBLEM STATEMENT AND PURPOSE

High-speed numeric computation has many applications including digital signal processing, graphics rendering, meteorological modeling, etc. These applications require numeric calculations to be computed quickly. In addition, the hardware may be required to compute large amounts of data or streaming data, which means long periods of time, may be expended performing the one type of computation. Personal computers are general purpose and not specifically designed for numeric calculations alone; instead they provide the best compromise between speed and flexibility.

The CORDIC algorithm can be very precise, but it has the disadvantage of being iterative and slow; the operations can take hundreds to thousands of clock cycles. Each iteration in the CORDIC algorithm provides increased accuracy at the output [4].

It would be beneficial to have specialized and fast hardware for high speed numeric calculations. Conventional methods for computing numeric functions include the CORDIC algorithm [2], [3], [4]. The problem is that specialized hardware is inflexible to computing different numeric functions as well as to changes in requirements or software updates. However, specialized hardware is fast.

A very fast method for numeric calculations is a look-up table [5], i.e. for every possible input, store the desired output of the numeric function. The disadvantage of this approach is that a large amount of memory is needed.

Field programmable devices have the advantage that one can quickly design, test and replace hardware functionality. This is compared to traditional methods, whereby a prototype is designed and simulated in software, prototyped on a prototyping board, and then sent to a manufacturer. This is expensive and time consuming, especially if there are changes required.

FPGA technology has improved to the point that a large amount of logic is available. If we have a few divergent needs that may require particularly heavy-computation that can best be solved by specialized hardware, we can use the FPGA devices to implement a specialized hardware design. Once the task has been completed,
the hardware can be reconfigured for other uses. The NFG we will discuss uses this principle on the SRC-6 computer system.

Lee, Wayne, Villasenor and Cheung [6], used a cascade of AND and OR gates to calculate segment addresses in a non-uniform segmentation implementation for hardware function evaluation. This circuit is useful for a limited class of functions. Sasao, Butler and Riedel [5] present a universal circuit that can cater to a wider class of functions.

Sasao, Butler and Riedel [5] have shown that elementary and non-elementary numeric functions can be computed quickly and accurately using a piecewise linear approximation method. This method provides some advantages over the memory method and the CORDIC algorithm. Less memory is required than a look-up table because the numeric function is segmented and the coefficients of the piecewise linear approximation are stored vice storing every possible input value and its corresponding output. The other advantage is that the accuracy can be determined at the outset and therefore is faster than the CORDIC algorithm; especially at higher accuracy when the CORDIC must go through several iterations to attain the desired accuracy. One more advantage to this approach is that it allows for one hardware design, with the memory contents being changed to handle different numeric functions [1].

This thesis investigates a piecewise quadratic implementation. The quadratic implementation requires fewer segments than the linear implementation to compute the same numeric functions to the same accuracy. This also means that the memory required is less than that required to implement a piecewise linear approximation NFG.
B. IMPLEMENTATION OVERVIEW

Figure 1 shows of the hardware required to build the NFG using quadratic approximation. The NFG architecture requires three multipliers. Each requires significant logic and causes significant delay.

\[
f(x) = C_2 X^2 + C_1 X + C_0
\]

Figure 1. Numeric function generator (NFG) architecture.
Table 1 shows the suite of functions used to test and design the NFG. Unlike logic or software design, there is no set of benchmarks. The specific functions have been chosen because they have appeared in previous papers on this subject [1], [5], [8], [9],[11], [12], [15].

<table>
<thead>
<tr>
<th>#</th>
<th>Function</th>
<th>Interval</th>
<th>Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(2^x)</td>
<td>([0,1])</td>
<td>([1,2])</td>
</tr>
<tr>
<td>2</td>
<td>(1/x)</td>
<td>([1,2])</td>
<td>([1/2,1])</td>
</tr>
<tr>
<td>3</td>
<td>(\sqrt{x})</td>
<td>([1,2])</td>
<td>([0,\sqrt{2}])</td>
</tr>
<tr>
<td>4</td>
<td>(1/\sqrt{x})</td>
<td>([1,2])</td>
<td>([1/\sqrt{2},1])</td>
</tr>
<tr>
<td>5</td>
<td>(\log_2(x))</td>
<td>([1,2])</td>
<td>([0,1])</td>
</tr>
<tr>
<td>6</td>
<td>(\ln(x))</td>
<td>([1,2])</td>
<td>([0,\ln2])</td>
</tr>
<tr>
<td>7</td>
<td>(\sin(\pi x))</td>
<td>([0,1/2])</td>
<td>([0,1])</td>
</tr>
<tr>
<td>8</td>
<td>(\cos(\pi x))</td>
<td>([0,1/2])</td>
<td>([0,1])</td>
</tr>
<tr>
<td>9</td>
<td>(\tan(\pi x))</td>
<td>([0,1/4])</td>
<td>([0,1])</td>
</tr>
<tr>
<td>10</td>
<td>(\sqrt{-\ln(x)})</td>
<td>([1/512,1/4])</td>
<td>([\sqrt{-\ln(1/4)},\sqrt{-\ln(1/512)}])</td>
</tr>
<tr>
<td>11</td>
<td>(\tan^2(\pi x) + 1)</td>
<td>([0,1/4])</td>
<td>([1,2])</td>
</tr>
<tr>
<td>12</td>
<td>-(x \log_2 x + (1-x) \log_2(1-x))</td>
<td>([1/256,1-1/256])</td>
<td>([0,1])</td>
</tr>
<tr>
<td>13</td>
<td>(\frac{1}{1+e^{-x}})</td>
<td>([0,1])</td>
<td>([1/2,\frac{1}{1+e^{-1}}])</td>
</tr>
<tr>
<td>14</td>
<td>(\frac{1}{\sqrt{2\pi}} e^{-x^2/2})</td>
<td>([0,\sqrt{2}])</td>
<td>([\frac{1}{\sqrt{2\pi}}, \frac{1}{\sqrt{2\pi}e^1}])</td>
</tr>
<tr>
<td>15</td>
<td>(\sin(e^x))</td>
<td>([0,2])</td>
<td>([1,\sqrt{1}])</td>
</tr>
</tbody>
</table>

Table 1. Suite of numeric functions and their domains.
C. THESIS ORGANIZATION

This thesis is organized into six chapters. Chapter I is the introduction, Chapter II covers the segmentation of numeric functions and the methods used for computing the approximation of the functions; this includes the discussion on how the coefficients were computed and how the memory files were used in the NFG. These programs were designed in MATLAB [7]. In Chapter III, the circuit description design is covered. Chapter IV introduces the SRC computer architecture. The experimental results are discussed in Chapter V. The summary and suggested future work is discussed in Chapter VI.
II. FUNCTION APPROXIMATION

The NFG approximates the realized function by polynomial. In a typical realization, many polynomials are used. A segment is a sub-domain in the interval of approximation where one polynomial is used to approximate the function. In this thesis quadratic polynomials are used. The benefit of using a polynomial approximation is that only one hardware design is required to realize a multitude of functions. The only change required to the hardware is to change the specific endpoints of the segmentation of the functions to be realized and the associated coefficients. The segmentation endpoint and coefficients are generated in MATLAB and are stored in a memory file. Segmentation is described in detail below.

The realized functions are approximated and the output of the hardware is only as accurate as the user-defined precision. The approximation error is $\varepsilon$. The exact function is evaluated for various values in the domain. The polynomial that is used to approximate the function is evaluated for the same values in the domain. The difference between these two results is the approximation error $\varepsilon$. The approximation error $\varepsilon$ is the constraint used to keep the approximation in check.

The approximation error $\varepsilon$, directly impacts how many segments are required and therefore dictates how much memory is used to store the coefficients. Small values require many segments.

A. QUADRATIC VS LINEAR

Nagayama, Sasao and Butler [8] showed that using quadratic approximations in the NFG requires an average of only 4% of the memory required when using linear approximations. This gives the motivation to pursue quadratic approximation following the work on linear approximation that was performed by Mack [1].

In Table 2, the number of segments required for different accuracies is tabulated for both quadratic approximation and linear approximation. A column is also included that shows the ratio of quadratic to linear segments required.
<table>
<thead>
<tr>
<th>Function</th>
<th>$\epsilon = 2^{-17}$</th>
<th>$\epsilon = 2^{-24}$</th>
<th>$\epsilon = 2^{-33}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Segments Quad/Lin %</td>
<td>Segments Quad/Lin %</td>
<td>Segments Quad/Lin %</td>
</tr>
<tr>
<td>$2^x$</td>
<td>7/75 9.33</td>
<td>35/849 4.12</td>
<td>278/19008 1.46</td>
</tr>
<tr>
<td>$1/x$</td>
<td>10/75 13.33</td>
<td>50/849 5.89</td>
<td>400/18996 2.11</td>
</tr>
<tr>
<td>$\sqrt{x}$</td>
<td>5/35 14.29</td>
<td>24/388 6.19</td>
<td>189/8729 2.17</td>
</tr>
<tr>
<td>$1/\sqrt{x}$</td>
<td>8/50 16.00</td>
<td>36/565 6.37</td>
<td>288/12684 2.27</td>
</tr>
<tr>
<td>$\log_2(x)$</td>
<td>9/76 11.84</td>
<td>44/853 5.16</td>
<td>351/19097 1.84</td>
</tr>
<tr>
<td>$\ln(x)$</td>
<td>8/63 12.70</td>
<td>39/710 5.49</td>
<td>311/15927 1.95</td>
</tr>
<tr>
<td>$\sin(\pi x)$</td>
<td>12/109 11.01</td>
<td>58/1227 4.73</td>
<td>461/27361 1.68</td>
</tr>
<tr>
<td>$\cos(\pi x)$</td>
<td>12/109 11.01</td>
<td>58/1227 4.73</td>
<td>459/27361 1.68</td>
</tr>
<tr>
<td>$\tan(\pi x)$</td>
<td>12/73 16.44</td>
<td>58/822 7.06</td>
<td>459/18371 2.50</td>
</tr>
<tr>
<td>$\sqrt{-\ln(x)}$</td>
<td>33/207 15.94</td>
<td>163/2356 6.92</td>
<td>1312/47188 2.78</td>
</tr>
<tr>
<td>$\tan^2(\pi x) + 1$</td>
<td>16/152 10.53</td>
<td>79/1721 4.59</td>
<td>631/38087 1.65</td>
</tr>
<tr>
<td>$-(x \log_2 x + (1-x) \log_2(1-x))$</td>
<td>37/314 11.78</td>
<td>183/3556 5.15</td>
<td>1459/76334 1.91</td>
</tr>
<tr>
<td>$\frac{1}{1+e^{-x}}$</td>
<td>4/20 20.00</td>
<td>20/226 8.85</td>
<td>158/5087 3.11</td>
</tr>
<tr>
<td>$\frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$</td>
<td>9/53 16.98</td>
<td>45/595 7.56</td>
<td>357/13312 2.68</td>
</tr>
<tr>
<td>$\sin(e^x)$</td>
<td>54/449 11.80</td>
<td>265/5099 5.20</td>
<td>2121/101065 2.10</td>
</tr>
</tbody>
</table>

Table 2. Segmentation required for linear and quadratic approximations.

To calculate the memory required for a single segment, one needs to take into account that memory for linear approximations only requires two quantities (slope and intercept) and memory for quadratic approximation requires three quantities. That is a 50% increase in memory requirements for a single segment when compared to linear.
However, the sheer difference in number of segments required for quadratic vice linear, more than counterbalances for the increase in memory requirements.

Table 2 shows that quadratic approximations can cover more functions with fewer segments than linear approximations and on average, quadratic approximations take up only 4% of the memory required to represent the same function when using linear approximations [8].

B. SEGMENTATION

To evaluate a numeric function using polynomial approximation, we need to segment the domain of the numeric function such that each segment has one set of coefficients that evaluate to the polynomial approximation of the given numeric function. The polynomial approximation needs to satisfy the user defined $\varepsilon$ such that any value in the domain that is evaluated using the polynomial will produce an output $f(x)$ that has an error no greater than $\varepsilon$ in magnitude. The segmentation is performed in MATLAB routines.

Segmentation can be performed using either uniform or non-uniform segments. The coefficients of the approximation polynomial can be computed using Polyfit [7], which is a built-in MATLAB function or the Chebyshev and the Remez [13] algorithm which is a user function. We will discuss these approaches in more detail.

1. Uniform and Non-Uniform Segmentation

There are two general methods used in approximating a function; uniform and non-uniform segmentation. Different functions behave differently when segmented using uniform or non-uniform segmentation. Non-uniform segmentation allows the user to take advantage of functions that have both rapidly changing and non-rapidly changing sections. When functions have sections of high curvature, non-uniform segmentation can create smaller segments to ensure the polynomial approximation does not exceed $\varepsilon$. The more quadratic or linear the function is, the better the polynomial approximation can fit a quadratic polynomial to it. As a result, segments are longer in regions where the function
is linear or quadratic. The goal is to achieve the fewest segments possible and yet achieve the approximation error specified by the user. Figure 2 shows the non-uniform segmentation of $\sqrt{-\ln(x)}$ using $\varepsilon = 2^{-16}$ (accurate to 16 binary bits). This function illustrates the advantage of non-uniform segmentation. The smaller segments are located at the beginning of the domain and the larger segments are at the end.

![Graph of non-uniform segmentation](image)

Figure 2. Quadratic segmentation of $\sqrt{-\ln(x)}$ shows the difference in the size of segments due to curvature of the function.

As mentioned above, the error associated with this segmentation should not exceed $\varepsilon = 2^{-16}$. Figure 3 shows the error across the interval of approximation when non-uniform segmentation is used. For all but the last segment, the maximum absolute error is the same (about $2^{-16.01}$). As shown in Figure 3, the error does not exceed $\varepsilon$ anywhere. Note that the error in the last (right most) segment is much less than in all other segments. This is because the last segment is truncated by the boundary of the domain interval before the algorithm has a chance to maximize the size of the segment.
Figure 3. Segment error of $\sqrt{-\ln(x)}$ when $\varepsilon = 2^{-16}$.

Figure 4, shows the approximation error in the case of this same function when uniform segmentation is applied\(^1\). To achieve uniform segmentation within the same approximation error specification i.e. $2^{-16}$, we are required to use the width of the narrowest segment which in this case is the very first segment.

\(^1\) Because a large number of segments are required, the line width occupies the whole of the figure, making it appear completely solid.
The error function for a uniform segmentation looks different from that of the non-uniform segmentation. The error for uniform segmentation is maximum i.e. $\varepsilon$ is attained in the most limiting segment. However, when looking at the other segments the error does not reach $\varepsilon$. Therefore a tapered effect is observed. To best demonstrate this effect, we shall use a less “dramatic” function than $\sqrt{-\ln(x)}$. Instead $\cos(\pi x)$ is used in Figure 5 and Figure 6 to show the difference in the error between the uniform and non-uniform segmentation.

Below in Figure 5, the error is tapered showing that the earlier segments don’t take full advantage of the entire segment because they have been limited by the smallest segment, located at the end of the domain for the $\cos(\pi x)$ function.

In Figure 6 however, you can see that non-uniform segmentation has taken full advantage of all the space and has fewer segments to represent the same function. This is the advantage of the non-uniform segmentation over uniform segmentation.
Figure 5. Uniform segmentation error for \( \cos(\pi x) \) when limited by \( \varepsilon = 2^{-17} \).

Figure 6. Error for non-uniform segmentation for \( \cos(\pi x) \) when limited by \( \varepsilon = 2^{-17} \).
In the segmentation of a numeric function, a user interface was designed in a MATLAB program to get the user’s choices. The user interface allows the user to select which function he/she would like to segment and allows the user to select the number of points (to subdivide the domain), $\varepsilon$, and whether uniform or non-uniform segmentation is used.

If the user selects non-uniform segmentation, the interface looks like that shown in Figure 7.

Figure 7. Quadratic approximation user-interface when non-uniform segmentation has been used.
If the user selects non-uniform segmentation, the user interface allows the user to select whether he/she wants to specify $\varepsilon$ or if they would like to use a fixed number of segments instead. The new user interface looks like that shown in Figure 8.

**QUADRATIC APPROXIMATION OF A FUNCTION USING CHEBYSHEV AND REMEZ ALGORITHM**

**Table 3**

<table>
<thead>
<tr>
<th>Functions to be compared</th>
<th>Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $2^x$</td>
<td>[0,1]</td>
</tr>
<tr>
<td>2. $1/x$</td>
<td>[1,2]</td>
</tr>
<tr>
<td>3. $\sqrt{x}$</td>
<td>[1,2]</td>
</tr>
<tr>
<td>4. $1/\sqrt{</td>
<td>x</td>
</tr>
<tr>
<td>5. $\log_2(x)$</td>
<td>[1,2]</td>
</tr>
<tr>
<td>6. $\log</td>
<td>x</td>
</tr>
<tr>
<td>7. $\sin</td>
<td>\pi</td>
</tr>
<tr>
<td>8. $\cos</td>
<td>\pi</td>
</tr>
<tr>
<td>9. $\tan</td>
<td>\pi</td>
</tr>
<tr>
<td>10. $\sqrt{-\log(x)} = \sqrt{-\ln(x)}$</td>
<td>[1/256,1/4]</td>
</tr>
<tr>
<td>11. $\tan</td>
<td>\pi</td>
</tr>
<tr>
<td>12. $-(x^2\log_2(x) + (1-x)\log_2(1-x))$</td>
<td>[1/256,1-1/256]</td>
</tr>
<tr>
<td>13. $1/(1+e^{-x})$</td>
<td>[0,1]</td>
</tr>
<tr>
<td>14. $(1/\sqrt{2\pi}))exp(-x^2/2)$</td>
<td>[0,\sqrt{2}]</td>
</tr>
<tr>
<td>15. $\sin</td>
<td>\exp(x)</td>
</tr>
</tbody>
</table>

**Input the Function, func[\sqrt{-1*log(x)}]: 8**

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

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

Input the number of Desired Segments [20]: 50

Input the no. of pts the fct is to be evaluated, N[1000000]: |
<table>
<thead>
<tr>
<th>Uniform Segmentation</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• No need for segment index encoder</td>
<td>• High curvature functions require many segments (wastes memory)</td>
</tr>
<tr>
<td></td>
<td>• Less complex hardware</td>
<td></td>
</tr>
<tr>
<td>Non-Uniform Segmentation</td>
<td>• High curvature functions with segments that are as wide as possible (Saves on memory)</td>
<td>• Requires segment index encoder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• More complex design</td>
</tr>
</tbody>
</table>

Table 3. Summary of Advantages and Disadvantages of Uniform and Non-uniform Segmentation.

2. **Segment Coefficients Using Polyfit and the Remez Algorithm**

To obtain the coefficients of a segment when segmenting any function, several different algorithms may be used. In [5], Sasao et al use the Douglas-Peucker algorithm [10] for segmenting and providing linear approximations to the functions. However this algorithm does not yield an optimum segmentation [11].

The initial work in this thesis used the *Polyfit* [7] function, available in MATLAB, to find the coefficients. *Polyfit* is computationally efficient and has been optimized for MATLAB. It requires a set of data points that represent the function that the user intends to best fit a polynomial of order $n$. In this thesis, we are working with quadratic functions and therefore use $n = 2$. *Polyfit* finds the coefficients to the approximating polynomial in a least squares sense [7] and returns a row vector with the coefficients of the polynomial. Least squares approximations minimize the average error on the interval selected. However, the worst-case error can be large. That is, it yields an average error that satisfies the constraint given, i.e. $\varepsilon$, but the worst-case errors may still exceed the constraint.

In analyzing the approximation polynomials produced from the coefficients provided by the *Polyfit* function, the graphs showing the error over each segment had the largest error at the begin and end points of the segment as can be seen in Figure 9 below.
This graph shows the weakness in using least squares approximation methods like that used by Polyfit. Our goal is to reduce the number of segments for the given function in order to restrain the maximum error to no greater than $\varepsilon$. Therefore, Polyfit was abandoned and instead the Remez algorithm [13] was used.

The Remez algorithm uses a method of approximation that minimizes the worst-case error. It belongs to the set of least maximum approximations (minimax approximations). The program ensures that there was no point in the interval where the error found by evaluating the difference between the approximation polynomial and the real function was greater than the constraint given.

![Graph](image)

Figure 9. Quadratic non-uniform segmentation approximation error using Polyfit.

The advantage of the Remez algorithm is to evenly distribute the error over the segment so that the maximum error is constrained by $\varepsilon$. This can be clearly observed by
comparing Figure 9 and Figure 10. The function, \( \cos(\pi x) \) with \( \varepsilon = 2^{-17} \), was used in both cases. Notice \textit{Polyfit} needed 14 segments while \textit{Remez} only required 12 segments. Both figures display only the first 4 segments. The difference is readily noticeable. Thus the \textit{Remez} covers a larger portion of the domain in the four segments than \textit{Polyfit}. As a result, it tends to reduce the number of segments. In the \textit{Remez} implementation, the 4\textsuperscript{th} segment extends right past 0.21 in the \( x \) domain, while \textit{Polyfit} barely makes it to 1.9.

The \textit{Remez} algorithm attempts to achieve the \textit{minimax} degree-\( n \) polynomial approximation of the given function on a defined interval. In the program that was used for this thesis, the interval is iteratively revised and the \textit{Remez} algorithm is repeatedly called until a degree-2 polynomial approximation that satisfies the constraint is achieved. The process is constrained by \( \varepsilon \), and the interval is increased or decreased until the optimum segment endpoint lies between the current point and the next point on the domain interval.

![Graph showing error for non-uniform segmentation approximation using Remez.](image)

\textbf{Figure 10.} Quadratic non-uniform segmentation approximation error using \textit{Remez}. (Only the first four segments are shown).
The *Remez* algorithm requires much more computational time and effort than the *Polyfit* function (which is already optimized for MATLAB). In general, for an $f$ with an interval $[a, b]$, there are several polynomials, but only one polynomial $p^*$ is the *minimax* degree-$n$ approximation. This approximation will have at least $n+2$ points, as described in inequality (0.1) that evaluate to yield an error that will be maximum magnitude and will alternate in sign.

$$ a \leq x_0 < x_1 < \ldots < x_{n+1} < b $$

(0.1)

The begin point and end point of the interval are included. In the case of quadratic approximations, a degree-2 polynomial can expect at least 4 points where the error will be maximum and will alternate in sign, as seen in Figure 10. The *Remez* algorithm is iterative and requires an estimate of the point where the error is maximum. The *Chebyshev* approximation is better than most other approximation algorithms in obtaining a polynomial close to the *minimax* polynomial $p^*$. When compared to *Taylor Series, Legendre, Chebyshev* provides a better estimate in most cases. For this reason, *Chebyshev* approximation is used to provide a set of starting points in the Remez algorithm in this thesis. The previous discussion is described in more detail in [13].

The function *ChebyRemez* in Appendix B was written to implement the *Remez* algorithm with an initial set of points where the error is maximum. Using *Remez* slowed down the program written to compute the coefficients; especially when higher accuracy was desired or in general, when the $x$ domain interval was assigned more points; $N$. To neutralize this effect, different algorithms were investigated to speed up the program. These are discussed further in the section three below.

3. **Algorithms Investigated to Speed-Up the Segmentation**

In the program proposed by Sasao, Butler and Riedel [5], the domain was divided into points and segmentation was determined by brute force, i.e. point by point to determine the required size of the segment. To attain high accuracy, the domain needs to
be subdivided into hundreds of thousands and even millions of points. This results in slow execution. We investigate ways to speed up the segmentation.

**a. Brute Force**

The lower value of the domain is established as the begin point. The program steps through each point computing the \textit{minimax} degree-2 polynomial approximation of the function. When evaluating any segment, (even two consecutive points), the program creates 1000 points between the given begin point and the end point. This ensures enough points for the program to locate the points in the segment where the maximum and minimum error is achieved, as described above. The coefficients required are then computed and next, the approximated polynomial is used to evaluate all the points in the current segment. These values are compared with the actual values from computing the real function. The maximum error is determined. If the error is smaller than \( \varepsilon \), the program steps one point to the right and repeats the process. Eventually, the polynomial approximation will produce an approximation where the maximum error exceeds \( \varepsilon \). At this point the program steps back one step and records the end point of the segment. For a typical segmentation with \( N = 1,000,000 \), this program takes much time. \( N \) is defined as the number of points on the entire interval of the domain, i.e. number of points on the interval \([a, b]\).

**b. Binary Search**

Binary search is really a two step process:

1. Locate: A point close to the optimum point is determined.

2. Pinpoint: Use brute force to move up to the optimum point.

In step 1, given a function \( f \), and an interval \([a, b]\), starting on the left at \( a \), the lower value of the domain is established as the begin point and the end point is set to \( b \). This is the entire domain interval over which the program computes the \textit{minimax} degree-2 polynomial approximation. Given the constraint, \( \varepsilon \), the program tests the error of the approximation and if the error is greater than the constraint, the program divides
the interval into two equal parts and decreases the proposed interval. Figure 11 shows a graphic representation of the first 4 iterations. These iterations are part of step 1; Locate.

The optimum is endpoint of the first segment is labeled $x_0$. Figure 11 shows the first iteration, interval $[a, b]$ is tested to determine if it is a good segment size. Since it is too large, the interval is divided into 2. The new interval is $[a, 1st\ proposed\ x_0]$. The process is repeated and the approximation of this new proposed segment is tested against the constraint. This is an iterative process that decreases the width of the segment. The next proposed segment is $[a, 2nd\ proposed\ x_0]$ as shown in Figure 11. Again the segment is tested. If the constraint is not met, the segment is decreased by 1/2.

Figure 11. Shows the interval and segmentation notation.
The process is repeated until the constraint is met. In Figure 11, the constraint is met on the fourth try and results in a proposed segment \([a, 3\text{rd } proposed \ x_0]\). Once below the optimum end point, the program increases the proposed segment endpoint until the constraint is exceeded. This means the segment is increased by half of the last width used to decrease the proposed segments. In Figure 11, the last width was \(2^{nd} \text{ proposed } x_0 - 3\text{rd } proposed \ x_0\). The process of increasing and decreasing the segment size by widths that are halved per iteration is repeated until the width being used to increment or decrement is 1. At this point, we are done with step 1 (Locate) and we move to step 2. Step 2 uses brute force to Pinpoint the optimum segment.

The binary search finds the actual segment end point in approximately \(s\) steps as described by inequality (0.2) where \(npts\) is the number of points in the initial proposed segment.

\[
s \geq 1 + \log_2(npts)
\]  

(0.2)

Compared to the number of steps required by brute force, this is a dramatic improvement. Consider \(N=1,000,000\), then the binary search for the first segment should yield around 21 steps to find the optimum segment end point \(x_0\); \(npts\) in this case is 1,000,000. The number of steps required to reach the segment end point is reduced as the program progresses to the end of the domain interval. This is because the argument \(npts\) in equation (0.2) decreases. In Table 4 the binary search takes 924 calls to the function \(\text{chebyRemz}\) as opposed to the brute force method which makes 1,000,000 calls.

The number of calls to the user programmed MATLAB function \(\text{chebyRemz}\) is used as a metric for two reasons: (1) the code for \(\text{chebyRemz}\) takes longer to execute than any other piece of code in the program and (2) the number of calls to the user programmed MATLAB function \(\text{chebyRemz}\) will vary depending on what numeric function is being segmented. Appendix D shows a copy of profile results [7] that shows
the execution time of each function. The goal is to minimize the number of calls to 
*chebyRemz*, thus speeding up the program.

Appendix A.2.1, part b shows the portion of the program that applies this method. The file name is *varQuadApproxBinSearch.m*.

Table 4 shows the number of calls to the function *chebyRemez* for 9 different algorithms that were investigated to speedup the segmentation. The first column is the number of points used to subdivide the domain. The next 9 columns are the different algorithms and the results. Only one function and one accuracy was used; $\sqrt{-\ln(x)}$ and $\varepsilon = 2^{-17}$ respectively.

<table>
<thead>
<tr>
<th>N</th>
<th>Binary Search</th>
<th>Thirds</th>
<th>Ratio</th>
<th>1 Est</th>
<th>2Est</th>
<th>3Est</th>
<th>Avg 1Est</th>
<th>Avg 3Est</th>
<th>Hybrid w/ Thirds &amp; 3Avg *1.05</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 M</td>
<td>924</td>
<td>764</td>
<td>1143</td>
<td>65400</td>
<td>3369</td>
<td>1903</td>
<td>5972</td>
<td>1960</td>
<td>98</td>
</tr>
<tr>
<td>100 K</td>
<td>764</td>
<td>640</td>
<td>699</td>
<td>6620</td>
<td>430</td>
<td>293</td>
<td>697</td>
<td>298</td>
<td>98</td>
</tr>
<tr>
<td>10 K</td>
<td>649</td>
<td>529</td>
<td>563</td>
<td>739</td>
<td>132</td>
<td>127</td>
<td>166</td>
<td>129</td>
<td>103</td>
</tr>
<tr>
<td>1 K</td>
<td>488</td>
<td>429</td>
<td>450</td>
<td>181</td>
<td>114</td>
<td>120</td>
<td>128</td>
<td>122</td>
<td>117</td>
</tr>
</tbody>
</table>

Table 4. Various methods show the number of calls to the function chebyRemz; segmentation of $\sqrt{-\ln(x)}$, $\varepsilon = 2^{-17}$ and various values of N.

c. **Divide by Thirds**

A second program was implemented that applied the same principle as *binary search*, however instead of taking off half of the width, the program took off two thirds (i.e. divide the remaining width by three). Therefore this method is also a two step process:

1. Locate: A point close to the optimum point is determined.

2. Pinpoint: Use brute force to move up to the optimum point.
Figure 12 shows the segmentation for the 5th segment. The domain interval is \([a, b]\), we start the segmentation of segment 5 at the end the 4th segment; \(x_4\).

Step 1: Denote the unsegmented part of the interval as \([x_4, b]\). A call to the function \(\text{chebyRemez}\) is used to generate a quadratic approximation. This approximation is tested to see if any points exceed the constraint \(\varepsilon\). If the constraint is met, then we have the final segment. Exit.

Step 2: Divide the initial width by three; the new value is \(1/3\) of the initial width. This is labeled as \(L_1\) in Figure 12. \(L_1\) is now the new proposed segment width and \(\text{chebyRemez}\) is called to establish a quadratic approximation for the interval. The point labeled \(x_5\) is the optimum segment endpoint. In Figure 12, \(L_1\) is clearly not the optimal width.

Step 3: The program divides \(L_1\) by three and the result is \(L_2\). A quadratic approximation is computed to test the approximation error against the constraint. Since \(L_2\) is below the optimum point, we initialize a new variable, \(\text{delta}\), to be used to keep track of the width which is being added or subtracted to the proposed width of the segment. \(\text{delta}\) is \(1/3\) of \(L_2\).

Step 4: Increase \(L_2\) by \(1/3\) of \(L_2\). This results in \(L_3\), which is tested to determine the approximation error. In Figure 12, \(L_3\) is still short of the optimum segment.

Step 5: Increase \(L_3\) by the same \(\text{delta}\), i.e. \(1/3\) of \(L_2\). The approximation is computed for the new proposed segment of width \(L_4\), and the approximation error tested against the constraint. This time we have exceeded the optimum endpoint, i.e. approximation error is greater than \(\varepsilon\). In Figure 12, \(L_4\) is larger than the optimum point.

Step 6: Since we have exceeded the optimum segment, we now reduce the variable \(\text{delta}\) to \(1/3\) of \(\text{delta}\). This value is the used to reduce \(L_4\) to a narrower width, i.e. \(L_5\). In Figure 12, \(L_5\) is still wider than the optimum width.

When the increment width is 2 or less, Locate is complete and the program goes to Pinpoint. The process stops when two adjacent points straddle the optimum.
segment endpoint. The lower value is \( x_3 \), the segment endpoint for the program. Since the domain has been divided into discrete points, \( x_3 \) is just shy of the optimum point. The approximation error of the new segment meets the constraint; however, the next point to the right of the optimum point has an approximation error that exceeds \( \epsilon \).

The results showed an improvement over binary search. Table 4 shows that the method of Thirds called the function chebyRemez 764 times as opposed to the binary search method that took 924 calls to achieve the same segmentation.

Other values besides one-third were tested, but they did not perform consistently better. Appendix A.2.1, part c shows the portion of the program that applies this method. The file name is varQuadApproxTHIRD.m

---

**Divide Interval by Thirds**

![Diagram](image)

**Figure 12.** Visual aid for description of divide by thirds algorithm.

**d. Increment by Ratio Numbers**

In this method, the width of the proposed segment is increased or decreased by multiplying the current proposed width by a series of fixed values. We have the same 2-step process of Locate and Pinpoint.
In Locate, the proposed width is the entire remaining width of the domain interval \([a, b]\) i.e. the width from point \(a\) to point \(b\). The width is tested to see if the constraint has been exceeded or not; except for the last segment, the width will always exceed the optimum segment because the entire remainder of the interval is used per iteration. As an example, consider that the first segment \([a, x_0]\) is already established (segment \([a, x_0]\) as shown in Figure 11). Next, the program needs to compute the second segment. The program will establish a proposed width \([x_0, b]\). This is the entire remainder of the interval. The ratios are applied to the width \([x_0, b]\). The result is shorter widths that are tested until the constraint is met. This method is similar to the method \textit{“Divide by Thirds,”}\ except that, a set of ratios are applied to the increment/decrement width.

Table 4 shows the implementation of increment by ratio numbers took 1143 calls to chebyRemz function. Appendix A.2.1, part d shows the portion of the program that applies this method. The file name is varQuadApproxRatio.m

e. \textit{Estimated Segment Widths (1, 2, 3, more and Average)}

Again, the 2-step process of Locate followed by Pinpoint is applied here. In Locate, an estimate of the segment is calculated.

Equation (0.3) is adapted from [15] to compute segment estimates for quadratic approximations. The derivation is in Appendix F. The accuracy \(\varepsilon\), and the third derivative of the function used to estimate the width of the segments. The proposed segment widths are tested and the program falls back on the brute force method after the initial estimate. This yields a large improvement from using the brute force method alone.

\[
EstSegLen = 4 \left[ \frac{3\varepsilon}{d^3 \frac{dy}{dx}} \right]^{1/3} \tag{0.3}
\]

\[
\begin{array}{c}
\int \\
\end{array}
\]
One Estimate: In Table 4, when one estimate is used, i.e. the third derivative is computed at \( x = \) begin point of the segment. The estimated width is added to the begin point and the proposed segment is tested. The brute force method takes over and single steps to the optimum segment width. The result was 65,400 calls to `chebyRemez`.

Two Estimates: The first estimated width is calculated using equation (0.3) and the third derivative is computed at the begin point of the segment. The resulting estimated width is added to the begin point and the resulting endpoint is used in equation (0.3) to make a second estimated width using the third derivative at the endpoint. The average of these two widths is the estimated width that is applied to the begin point to obtain a proposed endpoint. Again, the program uses the brute force method to complete the segmentation. This method improved the performance and took 3369 calls to `chebyRemez`.

Three Estimates: Two estimates are computed as described above. The result is divided in half the half-way point is used to compute the third estimate. The third estimate is averaged with the other two estimated widths to obtain the proposed segment width. As in the other two cases, the brute force method is then applied to complete the segmentation. Even further improvement was achieved; 1903 calls to `chebyRemez`.

Estimates with more than three widths were tested, but the performance began to degrade. So, an average was applied to the segments.

Average of one estimate: In the average method, one estimate was computed from the begin point. The estimate was used to define a proposed segment. The entire set of points on this proposed width are evaluated using equation (0.3). Then, the mean of the resulting vector of estimated widths was computed and used as the proposed segment width. The result appeared to be similar (not exact) to taking two estimates (when multiple functions are tested, on average the results of two estimates and the average method are similar). Table 4 shows that this method called `chebyRemez` 5972 times.
Average of three estimates: This method is a combination of taking three estimates as described above. All the points on the proposed width are evaluated with equation (0.3). This creates a vector of proposed estimates. Next evaluate the mean of the vector of proposed estimates to get one estimate. The results of this method are similar to taking three estimates. However, since we evaluate all the points on the interval, it takes slightly longer.

In [15], a comparison was made to show the benefit of three estimates over two estimates and one estimate in the case of linear approximation. While it is not discussed in [15], one estimate was computed in the linear approximation and the resulting proposed width was used to compute the mean of all the estimates obtained from evaluating all the points on the proposed width. The mean of the estimates was similar to taking the mean of just two estimates (begin point and proposed endpoint). In the quadratic case, the same method yielded results that were comparable to taking the mean of two estimates, just like the results in the linear case. However, when the mean of three estimates was used to define a proposed segment and the average of all the estimates on the newly proposed width was computed, the result was very close to taking the mean of just three estimates.

Closer analysis revealed that, in many cases, the average of all the points worked well and sometimes even better than just the mean of three individual estimates. The results appear in Table 5. The first column is the suite of numeric functions represented by a number; the focus should be on the comparison, not any particular numeric function. The second column is just the three estimates as described above, the third column is the average of the estimates calculated using all points on the proposed segment. The fourth column is the difference between the second and third column. The last column is a method described in part f; Hybrid of Thirds and three Estimates. Table 5 shows that taking the average of all estimates on the segment has a slight advantage over taking the average of just three single estimates. Therefore, looking back, Table 4 used only one numeric function, and that made it appear that the method of 3Avg was slightly worse, whereas in Table 5, we can see that the when applied to the entire suite of functions, the average over the entire segment (which was selected after three estimates),
was slightly better. The values at the bottom are the sum of all the calls to the approximating algorithm, \textit{chebyRemez}, which was the metric used to determine the comparative speed of the program.

<table>
<thead>
<tr>
<th>Function by Numbers</th>
<th>3Est</th>
<th>3Avg</th>
<th>Comparison (3Est – 3Avg)</th>
<th>*1.05 Hyb 3Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23</td>
<td>29</td>
<td>-6</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>93</td>
<td>103</td>
<td>-10</td>
<td>29</td>
</tr>
<tr>
<td>3</td>
<td>148</td>
<td>146</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>133</td>
<td>145</td>
<td>-12</td>
<td>23</td>
</tr>
<tr>
<td>5</td>
<td>83</td>
<td>84</td>
<td>-1</td>
<td>26</td>
</tr>
<tr>
<td>6</td>
<td>90</td>
<td>95</td>
<td>-5</td>
<td>23</td>
</tr>
<tr>
<td>7</td>
<td>266</td>
<td>87</td>
<td>179</td>
<td>59</td>
</tr>
<tr>
<td>8</td>
<td>6326</td>
<td>6210</td>
<td>116</td>
<td>61</td>
</tr>
<tr>
<td>9</td>
<td>128</td>
<td>92</td>
<td>36</td>
<td>35</td>
</tr>
<tr>
<td>10</td>
<td>293</td>
<td>298</td>
<td>-5</td>
<td>98</td>
</tr>
<tr>
<td>11</td>
<td>6233</td>
<td>6203</td>
<td>30</td>
<td>65</td>
</tr>
<tr>
<td>12</td>
<td>925</td>
<td>581</td>
<td>344</td>
<td>172</td>
</tr>
<tr>
<td>13</td>
<td>230</td>
<td>81</td>
<td>149</td>
<td>39</td>
</tr>
<tr>
<td>14</td>
<td>7378</td>
<td>7203</td>
<td>175</td>
<td>95</td>
</tr>
<tr>
<td>15</td>
<td>650</td>
<td>963</td>
<td>-313</td>
<td>222</td>
</tr>
<tr>
<td>SUM</td>
<td>22999</td>
<td>22320</td>
<td>679</td>
<td>981</td>
</tr>
</tbody>
</table>

Table 5. Comparison of “3 estimates”, mean of all estimates computed on proposed segment that was calculated after taking 3 estimates; “3 average” and a hybrid that exaggerates the approximation error by 5%. All cases, \(N=100,000\) and \(\epsilon = 2^{-17}\).

The next question is; should we use just three estimates or should we use the average of all the estimates computed from all the points on a proposed segment? The difference is small. The impact of the additional code that takes the average of an
entire segment did not exceed the time taken by *chebyRemez* and did not significantly impact the computing time of the program.

The additional code does not take add significantly to the program and since it has advantages, we kept the program that averages the estimates over the entire segment. The analysis to support that decision follows: Consider the small section of a *Profile report* from MATLAB that is similar to the one in Appendix D.

Table 6 shows the total time for *varQuadApprox* implemented with only three estimates. The time for the function, including all child functions is 44.438s. These values come from running the program with the function \(- (x \log_x x + (1-x) \log_2 (1-x))\), \(N=1,000,000\) and \(\varepsilon = 2^{-33}\).

<table>
<thead>
<tr>
<th>Function name</th>
<th>Calls</th>
<th>Total Time</th>
<th>Self Time</th>
<th>Total Time Plot</th>
</tr>
</thead>
<tbody>
<tr>
<td>multipleQuadApprox</td>
<td>1</td>
<td>44.906 s</td>
<td>0.156 s</td>
<td></td>
</tr>
<tr>
<td>varQuadApproxHyb3EstThird</td>
<td>1460</td>
<td>44.438 s</td>
<td>3.516 s</td>
<td></td>
</tr>
<tr>
<td>chebyRemz</td>
<td>13187</td>
<td>39.156 s</td>
<td>16.406 s</td>
<td></td>
</tr>
<tr>
<td>inline.subsref</td>
<td>87050</td>
<td>20.031 s</td>
<td>3.031 s</td>
<td></td>
</tr>
<tr>
<td>inlineeval</td>
<td>87202</td>
<td>17.031 s</td>
<td>17.031 s</td>
<td></td>
</tr>
<tr>
<td>polyval</td>
<td>69483</td>
<td>3.828 s</td>
<td>3.359 s</td>
<td></td>
</tr>
<tr>
<td>twosComp</td>
<td>5840</td>
<td>3.000 s</td>
<td>0.188 s</td>
<td></td>
</tr>
</tbody>
</table>

Table 6. Profile Report for \(- (x \log_x x + (1-x) \log_2 (1-x))\), \(N=1,000,000\) and \(\varepsilon = 2^{-33}\). Shows 44.438s for the *varQuadApprox* function that averages only three estimates.

The same function and parameters were run with the additional code that takes the average of all estimates over the entire segment. The results appear in Table 7. The total time for *varQuadApprox*, and all its child functions is 20.078s. The additional
code to compute the averages took 0.061s which translates to less than 1% of the time spent in \textit{varQuadApprox}. Therefore, the additional code is negligible. This particular function clearly shows the advantage of taking the average; greater than 50% improvement (44s to 20s).

It should be noted, that, in a few cases, the improvement was not as dramatic and in $\sqrt{-\ln(x)}$, the average code performed worse by 20% (20 seconds to 25 seconds). However, on average, it was better to take the average over the entire segment.

A slightly different problem; what happens when the third derivative is zero? This presents a problem in the computation of estimates (the third derivative is in the denominator of equation(0.3)). Therefore, one way to tackle the problem is to find the smallest non-zero, third derivative magnitude over the entire domain interval $[a, b]$ and use that to calculate the largest expected segment. This large segment is substituted whenever the third derivative is zero. In many cases, the resulting estimate is a poor estimate of the segment size, and tends to slow down the program when encountered. Therefore, a hybrid of the best segmentation processes was used and is described below.
f. Hybrid of Thirds and 3 Estimates

In this algorithm, we take advantage of the strengths of two programs. As with the other algorithms, we have a Locate and Pinpoint step. However, Locate is a combination of Divide by Thirds and 3 Estimates.
We know that $\varepsilon$ is the constraint and that when the approximation is good, then a ratio of the maximum approximation error to $\varepsilon$ should be very close to 1.0. This ratio can be used as a metric to determine the quality of our estimate. If the ratio is much larger than 1.0, because the segment is too large, then our estimate is too wide. If it is much less than 1.0, our estimate is too small.

To take advantage of the ratio of approximation error and $\varepsilon$, the program first takes the average of the three estimates and using the estimated width, computes the approximation error. If the ratio of the approximation error to $\varepsilon$ is large (greater than 1.002) or small (less than 0.9) the program takes the estimated width as a starting width. The program then takes a small fraction of that width (5%) and stores it in a variable that is used to decrease or increase the proposed width. The algorithm used is Divide by Thirds.

In addition to the steps taken above, the program was modified to exaggerate the error calculated from the approximation. This only happens in the final steps when trying to Pinpoint the end of the segment. This has two effects:

1. It drastically reduces the number of steps required because many of the estimations fall short and by exaggerating the error when the segment falls short, you reduce the distance that Pinpoint has to travel to exceed $\varepsilon$. If you combine the effect of saving two or three steps per segment, it adds up to 100 steps if the segmentation produces 33 segments.

2. Exaggerating the approximation error has the effect of making some of the segments slightly smaller than they would otherwise be if the approximation error were not adjusted. However, remember that the final segment is usually truncated and therefore can absorb the extra space created by making the previous segments narrower. In a way, by decreasing the size of the each segment by a small amount, it builds in a little slack per segment because the approximation error is slightly smaller than $\varepsilon$. The truncated segment is not optimized and can be increased to accommodate the small adjustments in all the other segments. Only in the very high precision segmentation do the segments increase noticeably. The increase is on the order of single digits when
considering hundreds or thousands of segments. This compromise is acceptable because it dramatically reduces the number of calls to *chebyRemez* as shown in the last columns of Table 4 and Table 5. Further, it does not increase the segments by any significant amount.

This hybrid method produces by far the best solution among all the algorithms discussed. Consider the function, $\sqrt{-\ln(x)}$, as shown in Table 4, only 98 calls to *chebyRemez* were needed to achieve segmentation, which is 0.0098% of the steps that brute force would take when $N=1,000,000$.

### C. MATLAB RESULTS

MATLAB was used to segment the numeric functions into piecewise quadratic segments. The uniform and non-uniform segmentation, number of segments required for each of the numeric functions and a comparison of the segmentation algorithms have been discussed in part B above.

The coefficients that represent the piecewise quadratic approximation for the segments are computed and stored in a file. These files can store the coefficients and segment boundaries in hexadecimal, binary or decimal form. The NFG implemented in the floating point number representation, uses the coefficients saved as decimal values. However, when the NFG is in fixed point number system, the coefficients saved are hexadecimal values.

Table 8 shows the data in the memory file for the non-uniform segmentation of $\cos(\pi x)$. At the top of the memory files is a decimal number that states the number of segments in the memory file. This is useful when reading the file to determine how many elements need to be read into the program.
The first column shows the segment end points. The next three columns are the coefficients of the quadratic polynomial that determines values in the segment. The order is $c_2$, $c_1$ and $c_0$ from left to right. Equation (0.4) shows the relationship of the coefficients to the polynomial.

$$f(x) = p = c_2 x^2 + c_1 x + c_0 \quad (0.4)$$

The hexadecimal values in Table 8 use a fixed point number system, where the first 17 bits are the integer including a sign bit and the last 47 bits are the fraction. The number is a two’s complement number. The number system is discussed in section III.

**D. SUMMARY**

MATLAB is used to segment the suite of functions in Table 1. The segmentation algorithm results in the fewest segments for a given accuracy constraint. In each segment
the \textit{minimax} quadratic approximation is achieved by computing the coefficients using the \textit{Remez} algorithm which performs better approximation than MATLAB’s available function; \textit{Polyfit}

The \textit{Remez} algorithm is slow; therefore various methods were investigated to find an efficient algorithm to compute the segmentation of the numeric functions. A hybrid of three algorithms is chosen as the best algorithm to compute fast segmentation of the suite of functions. Table 4 uses only one function, but summarizes the results of the comparisons.

Quadratic segmentation at high accuracy ($2^{-33}$) results in over 96\% fewer segments, compared to linear approximation as shown in Table 2.

The segmentation is the first step to building the NFG. Next the circuit has to be designed in hardware. In section III, we look at the components that make up the NFG circuit.
III. NFG CIRCUIT

A. CIRCUIT OVERVIEW

Figure 1 is duplicated here from section I for convenience. Figure 1 shows three multipliers, the segment index encoder, coefficients table and one 3-input adder. These are the hardware components for the NFG.

\[
f(x) = C_2 x^2 + C_1 x + C_0
\]

Figure 1. NFG Overview (duplicated from Section I).
The architecture has three 64 bit multipliers and one 3-input 64 bit adder. The adder and multiplier can be implemented in two’s complement or floating point by using the prescribed math operators. To generate a floating point multiplier or adder, the operands need to be declared as doubles or floats. To generate a two’s complement multiplier or adder, each operand needs to be declared as an integer, e.g. int64_t or int.

The segment index encoder is designed using a priority selector macro supplied by SRC and provided as a user callable macro. In uniform segmentation, multiplying by a segment density number can obtain the desired index.

1. Number System

To determine the number system to use, we need to know the range of values the NFG will have to handle. An analysis of the domain, range and coefficients provides the boundaries for the number system.

Table 9 shows the analysis of the numeric functions. The numeric functions have been ordered to show the most demanding to the least demanding. At the top, $\sqrt{-\ln(x)}$ requires 15 bits to accommodate any integer value the hardware may encounter, based on the range of values and coefficients.

The columns, Max and Min are the maximum and minimum values among all coefficient values, all possible domain and range values, i.e. any number that would appear in the computation done by the NFG. The column labeled $\log_2(\text{abs(largest one)})$ is obtained by comparing the absolute value of Max and Min and choosing the larger. We then compute the logarithm base 2 of this value. The final column shows the maximum number of bits required to represent the largest possible integer the NFG may encounter. Note that these values have been computed for a specific domain and different domains may require more or less bits. Table 2 shows the domains for each of the numeric functions that appear in Table 9.

The NFG requires at least 15 bits to represent the largest integer that may be encountered when computing the approximation of a numeric function. Therefore, the number system chosen is 16 bit integer and 16 bit fraction (i.e. 32 bit implementation). A
64 bit implementation has 32 bit integer and 32 bit fraction. The decimal point in the two’s complement number system is interpreted to be between bit 32 and bit 31 in a 64 bit number when the LSB is 0.

The 64 bit implementation benefits from using a 16 bit integer and 48 bit fraction, however the number of segments required is very large and these implementations were not investigated in detail. As an example, \( \cos(\pi x) \) at \( x = 2^{-49} \) and \( N=5,000,000 \) would require 19,167 segments.

<table>
<thead>
<tr>
<th>Function</th>
<th>Max</th>
<th>Min</th>
<th>( \log_2(\text{abs(largest one)}) )</th>
<th>Number of bits Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sqrt{-\ln(x)} )</td>
<td>24047.26212</td>
<td>-196.4301496</td>
<td>14.55358503</td>
<td>15</td>
</tr>
<tr>
<td>(-x \log_2 x + (1-x) \log_2(1-x))</td>
<td>360.5900787</td>
<td>-185.0149295</td>
<td>8.494215892</td>
<td>9</td>
</tr>
<tr>
<td>( \tan^2(\pi x) + 1 )</td>
<td>78.89563478</td>
<td>-26.88144904</td>
<td>6.301873574</td>
<td>7</td>
</tr>
<tr>
<td>( \sin(e^x) )</td>
<td>94.22597144</td>
<td>-96.6450472</td>
<td>6.594623895</td>
<td>7</td>
</tr>
<tr>
<td>( \tan(\pi x) )</td>
<td>19.70724959</td>
<td>-3.570442576</td>
<td>4.300654538</td>
<td>5</td>
</tr>
<tr>
<td>( \ln(x) )</td>
<td>4.934751084</td>
<td>-4.934751014</td>
<td>2.302977315</td>
<td>3</td>
</tr>
<tr>
<td>( \sin(\pi x) )</td>
<td>1.569925541</td>
<td>-4.934645908</td>
<td>2.302945666</td>
<td>3</td>
</tr>
<tr>
<td>( \cos(\pi x) )</td>
<td>1.569925541</td>
<td>-4.934645908</td>
<td>2.302945666</td>
<td>3</td>
</tr>
<tr>
<td>( 1/x )</td>
<td>2.997676487</td>
<td>-2.995354324</td>
<td>1.58384694</td>
<td>2</td>
</tr>
<tr>
<td>( \log_2(x) )</td>
<td>2.882537585</td>
<td>-2.162615784</td>
<td>1.52739419</td>
<td>2</td>
</tr>
<tr>
<td>( 2^x )</td>
<td>1.093679242</td>
<td>0.004061004</td>
<td>0.129189682</td>
<td>1</td>
</tr>
<tr>
<td>( \sqrt{x} )</td>
<td>2</td>
<td>-0.124634328</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( 1/\sqrt{x} )</td>
<td>2</td>
<td>-1.247861112</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( \frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}} )</td>
<td>1.414213562</td>
<td>-0.414997832</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>( \frac{1}{1+e^{-x}} )</td>
<td>1</td>
<td>-0.045379009</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 9. Maximum and minimum values encountered for each function in the NFG computation. Last column is the number of bits required for the integer portion.
2. 16, 32, 64 Bit Accuracy vs. 16, 32, 64 Bit Architecture

The accuracy and architecture can be built to match each other. Consider a set of values of 16 bit accuracy. Based on the number system, we would need 16 bits for integer and 16 bits for fraction (which is the accuracy). An architecture that matches these needs has to have 32 bit words; the architecture would be 32 bits. One implementation in the NFG was designed this way. Another design was built with 32 bit accuracy (32 bits fraction and 32 bits integer) and therefore the width of the architecture is 64 bits.

Another way to build the NFG is to use 64 bit architecture for all accuracies. This means that all values will be represented in 64 bits. Consider a value that is accurate to 16 bits. In this case, 32 bits are available to represent the fraction, but the fraction will only be accurate to 16 bits. The rest of the bits are irrelevant, but the hardware operates on all 64 bits. The architecture, in this case, does not match the accuracy.

B. CIRCUIT COMPONENTS

1. Segment Index Encoder

The segment index encoder accepts input (x) values (within the domain of the NFG) as inputs and outputs a number used to obtain the quadratic coefficients. The number is an index to the segment that x belongs. This only applies to the non-uniform segmentation.

User callable macros available in the SRC are used to implement a priority selector in the NFG. The prioritized selectors work as an “if-else-if” sequence. A wide number of options are available for 8, 16, 32 and 64 bit wide values. Each of these bit widths options can be implemented with 4, 8, 16, 32, 64, 128 or 256 elements. For example, choosing 64 bits and 256 elements, is equivalent to a priority encoder of 256 64 bit words.

The prioritized selector requires a Boolean condition and an assignment for a true condition. In the NFG, the Boolean condition is the comparison of the segment endpoint to the input value (numeric function argument; x). If x is less than the segment endpoint,
then \( x \) belongs to that segment and the corresponding assignment value is the index of the segment. Since \( x \) lies in the chosen segment, the index of the segment is used to access the polynomial coefficients that approximate the numeric function in that segment.

The types of selectors for a given segmentation are carefully chosen so as not to use more FPGA area than necessary. For example, consider a numeric function that has been segmented into 48 segments. The only selector that would accommodate this number of segments is the 64 element selector or greater. The 64 element selector can handle another 16 elements. However, since we do not need them, the whole selector wastes 48 elements. A better approach is to make two smaller selectors out of one 16 element selector and one 32 element selector. This saves FPGA area and allows us to build the selector we need. An example of the described code is provided in Table 10.

```c
//--Select Which Switch Statement will be executed---//
if ( varx <= 0.333333333333333310)
    sel =  1;
else if ( varx <= 0.500000000000000000)
    sel =  2;

//---------------Switch Statement-------------------//
switch (sel)
{
    case 1:
        select_pri_64bit_32val( varx <= 0.010351035103510351,  0,
                                 varx <= 0.020802080208020803,  1,
                                 varx <= 0.031231231231231204,  2,
                                 .
                                 .
                                 .
                                 varx <= 0.3228822822822822870, 30,
                                 31, &indx);
        break;
    case 2:
        select_pri_64bit_16val( varx <= 0.343734373437343750,  32,
                                 varx <= 0.354135413541354140,  33,
                                 varx <= 0.364586458645864590,  34,
                                 .
                                 .
                                 .
                                 varx <= 0.479147914791479170, 45,
                                 varx <= 0.489598959895989620, 46,
                                 47, &indx);
        break;
}
```

Table 10. Code that uses two selectors to implement 48 segments.
To implement a larger than 256 selector, a combination of available selectors can be used. In the .mc file, an if-else-if statement precedes the set of selectors and selects which one of the selectors will be used to encode the index.

More detail on the various selectors available in the SRC, is provided in Appendix A.10 of [17].

2. Indexing in Uniform Segmentation

In uniform segmentation, a number that is multiplied by the input value, $x$, is used to compute the appropriate segment; essentially, a segment number density. It represents the number of segments per unit length. Instead of a segment index encoder, $x$ is multiplied by the segment density number and the integer result is the index that is applied to the coefficients’ arrays to access the coefficients for the quadratic approximation.

The segment density number is obtained by dividing the entire interval by the number of segments and inverting the result.

For example, consider an interval, $[0, 0.5]$ with uniform segmentation. If 100 segments are realized, then the number used to multiply all inputs is $\left(\frac{0.5 - 1}{100}\right)^{-1} = 200$. If the input is 0.3356, then the coefficients will be extracted from the OBM array using the index $67\left(\text{floor}(0.3356 \times 200 = 67.12) = 67\right)$.

If the interval of the domain starts at a non-zero value, then the index obtained from the above method will be offset. Simply subtract the offset from the index obtained to get the true index into the array. This extra step increases the pipeline depth of the NFG. The effect is greater in floating point implementation compared to fixed point implementation.
a. Floating Point Implementation

The uniform segmentation of the NFG in floating point requires three files; main.c, <subroutine>.mc and memoryFile. An array containing floating point values of the endpoints and coefficients of the uniform segmentation are passed into the OBM, via a DMA call. The sample points for testing the NFG are placed in a separate array and passed into OBM via a second DMA call. The memory file contains three numbers at the beginning of the file:

- The number of segments (which is also the number of sets of coefficients in the memory file). Stored as an int.
- The segment density number that is used to determine the segment that any $x$ input belongs to. Stored as a double.
- The offset value (needed for functions that have an interval with a non-zero begin point)

b. Fixed Point Implementation

The uniform segmentation, fixed point implementation, works similar to the floating point implementation. Three files are needed; main.c, <subroutine>.mc and memoryFile. The coefficients in the memory file and in the computation are two’s complement hexadecimal numbers, as described in the section on number systems. The memory file contains three numbers at the beginning of the file:

- The number of segments (which is also the number of sets of coefficients in the memory file). Stored as an int.
- The segment density number that is used to determine the segment to which any $x$ input belongs. Stored as an int64_t.
- The offset value (needed for functions that have an interval with a non-zero begin point)

The computation of the index, and therefore, the segment, is accomplished in two’s complement. One major problem exists in this multiplication; the product is 128 bits, but the architecture only allows 64 bits to be stored. This means the upper 64 bits are truncated. In addition, since the decimal point in the operands is 32 bits from the LSB, the decimal point in the product is between bit 63 and bit 64 (when LSB is
considered to be bit 0). This means we lose all integer values and the entire product that is stored is only the fraction portion of the true 128 bit product.

To represent the full range of numbers in the numeric functions, we need to retrieve some of the upper bits. The segment density number is normally a whole number (without value in the fraction); occasionally the segment density number may have a small but negligible fraction. We can perform a 16 bit logical shift right to the segment density number without a large loss. This opens up 16 bits in the integer part of the product; which is really the index into the array of coefficients. 16 bits is enough to represent over 65,000 segments\(^2\). The product is then shifted 48 bits to the right to give an index number (index numbers must be whole numbers). This method is prone to rounding errors which occasionally result in the wrong index.

Other schemes have to be implemented when both operands have a significant amount of data in the fraction. The section on the two’s complement multiplier discusses other schemes in more detail.

3. **Coefficients Table**

The coefficients to the quadratic equation for each segment are stored in an array in the OBM banks on the MAP® board. The segment index encoder provides an index into the array. The coefficients are accessed and applied to the quadratic equation along with the x value that is being evaluated.

4. **Multiplier**

The three multipliers shown in Figure 1 are either implemented in two’s complement or floating point. Floating point operations increase the pipeline depth, but are easier to code.

---

\(^2\) The largest number of segments is 34,483, which is the uniform segmentation of \(\sqrt{-\ln(x)}\), when \(x = 2^{-38}\). Table 12 shows the number of segments for various functions when using uniform segmentation.
a. **Floating Point Multiplier**

The floating point multipliers implemented in the NFG are implicitly instantiated. The operands are declared as doubles and when the multiplier operator in the `.mc` file was applied, the MAP® compiler builds the floating point multiplier.

b. **Two’s Complement Fixed Point Multiplier**

The three main categories of interest are:

- Fixed point two’s complement multiplier
- Floating point multiplier
- Signed Magnitude multiplier

The signed magnitude multiplier was not built. The fixed point multipliers implemented in the NFG are either implicitly instantiated or explicitly built in HDL. The two’s complement fixed point multiplier was built in Verilog, VHDL and implicitly instantiated by the SRC MAP® with various levels of success.

To implicitly instantiate the two’s complement multiplier, the operands are declared as integer values (`int64_t`) and when the multiplier operator in the `<subroutine.mc>` file is applied, the MAP® compiler builds the appropriate multiplier.

This method has two major problems; (1) The SRC 64bitx64bit multiplier does not result in a 128 bit product. Instead, it results in a 64 bit product that is composed of only the lower 64 bits. (2) If the MSB at the cutoff is a binary 1, the number appears as a negative number, even though it is really a positive number.

Because of the number system chosen, i.e. 32 bits of integer and 32 bits of fraction, multiplication results in a product that represents only the fraction portion of the multiply; the integer portion, bits 65 through 128, are truncated.

One way to overcome this limitation is to choose a different number system that has fewer bits to represent the fraction, but this reduces the accuracy of the NFG and it still limits the size of the integer. The integer must be at least 16 bits to provide full coverage of the values encountered in the suite of functions in Table 1. One
The implementation of the NFG was built by shifting the operands right 8 bits, before the multiply. This allowed for 16 bits to be represented in the integer portion of the product. In this case, the best accuracy that one would expect to attain is 24 bits, i.e. $2^{24}$. Due to truncating the operands, error is propagated to the output and the accuracy is not reliable. Shifting values presents another problem, because, if the MSB is a binary 1, then the right shift operation will sign extends the number. This has unwanted effects. A product may be positive, but if the bit right before the cutoff point is a binary 1, the shifted values will be sign extended and we have to zero out the leading bits. More detail on the results of this method can be found in section V where the implementation results are covered.

The best solution is to build an HDL multiplier that can compute the result in the number system chosen and therefore keep the desired accuracy and the best range for the integer without any sacrifices to accuracy. The problem with this method is that it requires a long carry chain.

Verilog or VHDL can be used to explicitly build the multipliers. Several multipliers were built in VHDL and Verilog. The HDL files do not meet the timing requirements while running the NFG, although the program compiles without any errors. Simulation using Modelsim and Xilinx ISE showed that the design for the multipliers was correct. The problem appears to be the carry chain that is required to add all the partial products.

Further investigation is needed to determine if indeed the problem is in the carry chain and if a carry save adder (CSA) followed by a carry lookahead adder (CLAH) are required. (Which were not built)

5. Adder

The NFG required a 3-input adder. As in the case with the multipliers, floating point and fixed point adders are instantiated by the MAP® compiler.
C. SUMMARY

The NFG circuit requires three multipliers and one 3-input adder. Floating point implementation is easier than the fixed point implementation, but requires more hardware. The multipliers can be instantiated implicitly or in the case of fixed point, the user has the option to explicitly build the multiplier in HDL.

Fixed point arithmetic presents some challenges with rounding and truncating of the operands and results.

The circuit design was built on the SRC-6E reconfigurable hardware. Section IV provides a background on the SRC-6E system to give a better understanding of the hardware and software system.
IV. SRC BACKGROUND

A. INTRODUCTION

The late Seymour Cray established SRC Computers Incorporated in Colorado Springs, Colorado in 1996. SRC developed the IMPLICIT+EXPLICIT™ architecture that is designed to provide increased performance over conventional processors [16].

1. IMPLICIT+EXPLICIT™ Architecture

The IMPLICIT+EXPLICIT™ architecture allows the full integration of Dense Logic Device (DLD) technology such as ASIC devices or microprocessors with reconfigurable Direct Execution Logic (DEL). SRC’s Carte™ Programming Environment lets the programmer choose that part of code that executes in the fixed logic (i.e. microprocessor - implicit) and that part that executes in the reconfigurable hardware (explicit) [16]. Figure 13 is an overview of the SRC IMPLICIT+EXPLICIT™ architecture.

Figure 13. IMPLICIT+EXPLICIT™ architecture [16].
The user can program in the Carte™ Programming Environment in C or FORTRAN instead of designing logic. A single executable is generated that specifies which operations execute on which parts of the system. If the programmer desires to design the logic, he/she can design in a schematic capture program and generate VHDL or Verilog files that are used as macros. The user can also code the Verilog and VHDL files and use them as macros. More information on what is needed to implement macros is provided in the section on software code [16].

B. HARDWARE

Figure 14 shows 3 Xilinx XC2V6000 FPGAs on the MAP®, 2 sets of memory and some ROM.

![MAP® Hardware overview diagram](image)

Figure 14. MAP® Hardware overview diagram [18].
There are three FPGAs. The user can program two of the FPGAs, while the third is used as a controller. The FPGAs are Xilinx Virtex II’s, XC2v6000 with a -4 speed grade. There are 6 banks of dual ported On-board Memory (OBM) with a total of 24 MB (high-speed local memory). The OBM RAM is connected to the two user logic FPGAs via a 4800 MB/s (OBM RAMs is also connected to third FPGA via another 4800 MB/s bus).

The two FPGAs are connected directly to each other with access to a 4 MB dual ported memory bank for inter-chip data exchange on a 4800 MB/s bus. The two FPGAs have two General purpose I/O (GPIO) ports for direct data off the MAP® that is connected via a 2400 MB/s bus.

Internal to each user FPGA is an additional 144 BRAM 18KB blocks [19] for a total of 2,592 KBs of BRAM. BRAM is fast since it is on the FPGA chip.

C. SOFTWARE CODE

A user program consists of two C programs, main.c and <subroutine>.mc as well as “helper” files.

1. main.c

The main routine is a C program that runs on the SRC’s Intel processor. The main routine contains the declarations for the subroutine functions and makes the subroutine functions visible to the Intel processor.

To effectively use the MAP® hardware, we need to partition the code and select the portions that will provide improved overall performance when executed on the MAP® processor. These include loops that can be pipelined, or manipulation of bits that are in a long bit stream of data [20]. They are placed in a C program described in the next section.
2. \textless subroutine\textgreater .mc

These are the files that contain the function subroutine that is called from the main routine to execute on the MAP® boards. The code in the .mc files should not contain any external calls outside the MAP® with the exception of SRC-defined or user-defined macros.

The .mc file does not allow any system calls or runtime functions that require intervention from the operating system. The only exception is the \texttt{printf} statement which is ignored during compile time except in debug mode; the \texttt{printf} statement is very handy in the debug mode. This means that .mc cannot contain any additional system header files besides the \texttt{libmap.h} header file, which is the only runtime library allowed in the MAP® [16].

3. Makefile

Many files are used during compilation. The Makefile identifies the files and commands that are used by the compiler. The Makefile allows the programmer to set the source code preprocessing environment variables, C compiler flags, MAP® compiler flags and simulation compiler flags [16]. SRC provides a template that can be tailored for the specific needs of the program.

4. Macros

Macros allow the programmer to design in HDL. It is more flexible than just the \textless subroutine\textgreater .mc file alone. Macros allow the programmer the flexibility of creating specific and unique hardware that can manipulate wide bit values and all the way down to single bits.

To implement a macro, the Makefile needs to know where to find the HDL files and the macro support files. The following are required for macros:
a. info

The info file provides the MAP® compiler with the name of the macro and the relationship between the call and the macro instantiation. The info file defines the name, characteristics (such as whether the macro is pipelined), whether it interacts with external systems (outside the code block), the latency of the circuit specified by the macro, the number of inputs and outputs. The signal names and macros in the Verilog code that is generated by the MAP® compiler requires the info file in order to correctly map the operators and calls in the source program [16].

The info file can also be used to define the behavior, in C, that the hardware is expected to perform. This feature is available for the debug mode and uses the Intel processor to emulate the hardware that the programmer intends to design on the MAP®.

If multiple macros are used, the user only needs one info file. The information associated with the different macros must be put into the one info file.

b. blk.v

The black box file, blk.v, describes the macros interface. It is a simple file that tells the number of bits for each input and output and is described in a Verilog-style.

If multiple macros are used, the user must add the interface information into a single blk.v file.

c. HDL Files

The HDL files can be written in VHDL or Verilog. They are specified in the Makefile.

d. Location for NGO Directory

This location must be specified in the Makefile to identify the directory that will contain all the NGO files. The recommended practice is to put the NGO
directory in the same directory with all the macro information, and include the *info*, *blk.v* and *HDL* files.

The macros describe the logical design at a high level. The NGO files are used by NGDbuild to create an NGD file. The NGD file describes the logical design in terms of Xilinx primitives (basic elements in the FPGA).

D. SUMMARY

The SRC system provides flexibility, and a user-friendly interface for designing specialized hardware.

Various implementations of the NFG were built on the SRC system. The results are documented in section V.
V. IMPLEMENTATION RESULTS

A. UNIFORM SEGMENTATION

Uniform segmentation is easier to implement in terms of programming the <subroutine>.mc file. Appendix C shows the code main.c and subr.mc for uniform and non-uniform segmentation.

1. Floating Point Implementation

Two major advantages of the uniform segmentation floating point implementation are (1) the multiplier does all the work of moving the decimal point and (2) once the file is compiled, any function can be computed without having to recompile. The only requirement is to change the memory file.

The disadvantage is that floating point operations require much hardware. The complexity of using floating point is hidden from the user, but is evident in the amount of multipliers consumed and the pipeline depth required. Figure 15 shows the summary report after the compile process is completed; (i.e. after the user types make hw).

<table>
<thead>
<tr>
<th>loop on line 55:</th>
</tr>
</thead>
<tbody>
<tr>
<td>clocks per iteration: 1</td>
</tr>
<tr>
<td>pipeline depth: 84</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>inner loop summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>Place and Route Summary</td>
</tr>
</tbody>
</table>

| Number of Slice Flip Flops: 17,647 out of 67,584 26% |
| Number of 4 input LUTs: 9,299 out of 67,584 13% |
| Number of occupied Slices: 11,390 out of 33,792 33% |
| Number of MULT18X18s: 64 out of 144 44% |
| freq = 100.2 MHz |

Figure 15. NFG Pipeline depth and place and route summary.
The SRC has user callable macros that are summarized in Appendix A of [20]. Figure 16 shows the difference between the pipeline depth of the NFG and the SRC user callable macro. The pipeline depth for the NFG is a 20% less than that of the user callable macro.

Figure 16 also shows the place and route information associated with mapping both the NFG and SRC’s user callable cosine macro. Comparing Figure 15 with Figure 16, one can see the hardware requirements have increased due to adding SRC’s user callable macro.

Figure 16. Pipeline depth (NFG and SRC Cosine Macro). Place and route summary.

Table 11 shows a comparison of the hardware used to build the NFG, the macro and both on the same FPGA. The comparison shows that the NFG approximation is close to the macro in terms of hardware needed; with the exception of the multiplier. The NFG requires a slightly more than double the multipliers that the macro requires.
Table 11. Comparison of NFG uniform segmentation and macros: NFG alone, Macro alone and both (function is \( \cos(\pi x) \)). Implementations without offset.

<table>
<thead>
<tr>
<th></th>
<th>NFG Alone</th>
<th>Macro Alone</th>
<th>NFG &amp; Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Slice Flip Flops</td>
<td>26%</td>
<td>21%</td>
<td>40%</td>
</tr>
<tr>
<td># of 4 input LUTs</td>
<td>13%</td>
<td>14%</td>
<td>25%</td>
</tr>
<tr>
<td># of occupied Slices</td>
<td>33%</td>
<td>27%</td>
<td>52%</td>
</tr>
<tr>
<td># of Block RAMs</td>
<td>0%</td>
<td>1%</td>
<td>1%</td>
</tr>
<tr>
<td># of MULT18X18s</td>
<td>44%</td>
<td>19%</td>
<td>63%</td>
</tr>
<tr>
<td>Freq</td>
<td>100.2 MHz</td>
<td>100.1 MHz</td>
<td>100.0 MHz</td>
</tr>
</tbody>
</table>

The implementation described above applies to functions that have a domain interval that starts at zero. If the interval starts at a non-zero value, then the index computed needs to be adjusted by an offset value. Figure 17 shows the hardware requirements when the offset is applied.

Figure 17. Pipeline depth (NFG and SRC \( \sqrt{-\ln(x)} \) implemented in macros). Place and route summary with subtraction hardware included for computing offset (when finding the index. of coefficients).

The adjustment is a subtraction operation. In the floating point number system, the hardware required to perform arithmetic computations is large and by adding a
subtraction computation, the NFG pipeline depth increases from 84, as shown in Figure 15 and Figure 16, to 98 as shown in Figure 17.

Figure 18 shows the comparison between the output of the macro and the NFG. The macro computes using float values, while the NFG can compute higher precision values. Therefore, a user can achieve a shorter pipeline depth and higher accuracy by using the NFG. The cost of using the NFG is that the user must have a memory file to load the coefficients of the quadratic approximation into OBM.

Figure 18 shows the comparison of the results from the NFG that uses a memory file with the coefficients computed with an accuracy of $\varepsilon = 2^{-33}$. This implementation has 459 segments and an accuracy of 32 bits.

The first labeled column in Figure 18 is, $x$ values, which shows the values of $x$, which in this case are the endpoints. Based on the Remez algorithm, the end points, begin points and two other points in the middle of each segment have the worst case approximation error. Therefore, we expect to see the error of these points to be very close to the maximum error allowed for the segmentation i.e. $\varepsilon = 2^{-33} = 1.1641532... \times 10^{-10}$ (essentially at the 10th decimal place).

Excel and MATLAB are used to compute $\cos(\pi x)$. The results for Excel and MATLAB are exactly the same as shown in Figure 18, in the column labeled Excel-MATLAB (difference of the results is zero). The NFG output and the SRC cosine macro are compared to Excel and the results are shown in the last two columns. Figure 18 shows that SRC’s macro is accurate to $\varepsilon = 2^{-23}$, which is the correct accuracy for floating point values. The NFG is accurate to within $2^{-33}$. This accuracy can be increased without an increase in FPGA hardware, if desired. The cost is OBM memory to store a larger coefficients table.
<table>
<thead>
<tr>
<th>x Values</th>
<th>NFG OUTPUT</th>
<th>SRC MACRO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00089400089400090</td>
<td>ySRCMacro: 0.999996055923</td>
<td>Excel Cosine: 0.999996055923</td>
</tr>
<tr>
<td>0.001788501788501780</td>
<td>ySRCMacro: 0.9999984214899</td>
<td>MATLAB: 0.9999984214899</td>
</tr>
<tr>
<td>0.002683002683002680</td>
<td>ySRCMacro: 0.999964477019</td>
<td>Excel-MATLAB: 0.0000000001017031</td>
</tr>
<tr>
<td>0.003577503577503570</td>
<td>ySRCMacro: 0.9999999842442</td>
<td>ySRCMacro - Excel: 0.0000000001012988</td>
</tr>
<tr>
<td>0.004472004472004470</td>
<td>ySRCMacro: 0.99990131383</td>
<td>-0.000000000138820</td>
</tr>
<tr>
<td>0.005365005365005360</td>
<td>ySRCMacro: 0.99996591900</td>
<td>-0.000000000178917</td>
</tr>
<tr>
<td>0.006260506260506260</td>
<td>ySRCMacro: 0.99998058208</td>
<td>-0.000000000211574</td>
</tr>
<tr>
<td>0.007155007155007150</td>
<td>ySRCMacro: 0.999974377745</td>
<td>-0.000000000276480</td>
</tr>
<tr>
<td>0.008049508049508050</td>
<td>ySRCMacro: 0.9999680268602</td>
<td>-0.000000000307531</td>
</tr>
<tr>
<td>0.008944008944008940</td>
<td>ySRCMacro: 0.999960525006</td>
<td>-0.000000000339733</td>
</tr>
<tr>
<td>0.009838509838509830</td>
<td>ySRCMacro: 0.999922367549</td>
<td>-0.000000000372570</td>
</tr>
<tr>
<td>0.010730107301073010</td>
<td>ySRCMacro: 0.999943157688</td>
<td>0.000000000405187</td>
</tr>
<tr>
<td>0.011627511627511620</td>
<td>ySRCMacro: 0.9999323893727</td>
<td>0.000000000500591</td>
</tr>
<tr>
<td>0.012520212520212520</td>
<td>ySRCMacro: 0.999926318659</td>
<td>0.000000000510538</td>
</tr>
<tr>
<td>0.013416513416513410</td>
<td>ySRCMacro: 0.999911853121</td>
<td>0.000000000535635</td>
</tr>
<tr>
<td>0.014311014311014310</td>
<td>ySRCMacro: 0.998898974718</td>
<td>0.000000000550152</td>
</tr>
<tr>
<td>0.015205015205015200</td>
<td>ySRCMacro: 0.998859327721</td>
<td>0.000000000573486</td>
</tr>
<tr>
<td>0.016099516099516090</td>
<td>ySRCMacro: 0.998812199455</td>
<td>0.000000000576733</td>
</tr>
<tr>
<td>0.016994016994016990</td>
<td>ySRCMacro: 0.998875183430</td>
<td>0.000000000588137</td>
</tr>
<tr>
<td>0.017988571988571980</td>
<td>ySRCMacro: 0.9988421293434</td>
<td>0.000000000601571</td>
</tr>
<tr>
<td>0.01878301878301880</td>
<td>ySRCMacro: 0.9988254980847</td>
<td>0.000000000626865</td>
</tr>
<tr>
<td>0.019677519677519670</td>
<td>ySRCMacro: 0.998808929425</td>
<td>0.000000000649836</td>
</tr>
<tr>
<td>0.020572020572020570</td>
<td>ySRCMacro: 0.997912278907</td>
<td>0.000000000669832</td>
</tr>
<tr>
<td>0.021466521466521465</td>
<td>ySRCMacro: 0.997726847897</td>
<td>0.000000000702730</td>
</tr>
<tr>
<td>0.022361022361022360</td>
<td>ySRCMacro: 0.997533535789</td>
<td>0.000000000726736</td>
</tr>
<tr>
<td>0.023255523255523255</td>
<td>ySRCMacro: 0.997332350318</td>
<td>0.000000000746732</td>
</tr>
<tr>
<td>0.024150024150024150</td>
<td>ySRCMacro: 0.997123286416</td>
<td>0.000000000767728</td>
</tr>
<tr>
<td>0.025044025044025044</td>
<td>ySRCMacro: 0.99690472609</td>
<td>0.000000000790724</td>
</tr>
<tr>
<td>0.025938525938525938</td>
<td>ySRCMacro: 0.996681866744</td>
<td>0.000000000808220</td>
</tr>
<tr>
<td>0.026833026833026830</td>
<td>ySRCMacro: 0.996448990104</td>
<td>0.000000000825716</td>
</tr>
</tbody>
</table>

Figure 18. Results from Uniform Segmentation NFG compared with SRC Cosine Macro, MATLAB and Excel.
The results in Figure 18 show that the accuracy in the NFG can be increased to 33 bits. To take advantage of the uniform segmentation, we need to know the number of segments required in uniform segmentation. The quadratic coefficients for the numeric functions are stored in OBM memory. Table 12 shows the number of segments required for each of the accuracies. All the segments shown can be implemented in the NFG, even when the number of segments is as large as 34483; as in the numeric function: $\sqrt{-\ln(x)}$

<table>
<thead>
<tr>
<th>Numeric Function</th>
<th>$\varepsilon = 2^{-17}$</th>
<th>$\varepsilon = 2^{-24}$</th>
<th>$\varepsilon = 2^{-33}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^x$</td>
<td>8</td>
<td>39</td>
<td>311</td>
</tr>
<tr>
<td>$1/x$</td>
<td>17</td>
<td>81</td>
<td>646</td>
</tr>
<tr>
<td>$\sqrt{x}$</td>
<td>7</td>
<td>33</td>
<td>257</td>
</tr>
<tr>
<td>$1/\sqrt{x}$</td>
<td>11</td>
<td>55</td>
<td>439</td>
</tr>
<tr>
<td>$\log_2(x)$</td>
<td>13</td>
<td>64</td>
<td>506</td>
</tr>
<tr>
<td>$\ln(x)$</td>
<td>12</td>
<td>56</td>
<td>448</td>
</tr>
<tr>
<td>$\sin(\pi x)$</td>
<td>14</td>
<td>70</td>
<td>559</td>
</tr>
<tr>
<td>$\cos(\pi x)$</td>
<td>14</td>
<td>70</td>
<td>559</td>
</tr>
<tr>
<td>$\tan(\pi x)$</td>
<td>18</td>
<td>88</td>
<td>704</td>
</tr>
<tr>
<td>$\sqrt{-\ln(x)}$</td>
<td>794</td>
<td>4017</td>
<td>34483</td>
</tr>
<tr>
<td>$\tan^2(\pi x)+1$</td>
<td>30</td>
<td>151</td>
<td>1204</td>
</tr>
<tr>
<td>$-(x \log_2 x + (1-x) \log_2 (1-x))$</td>
<td>399</td>
<td>2013</td>
<td>16667</td>
</tr>
<tr>
<td>$\frac{1}{1+e^{-x}}$</td>
<td>5</td>
<td>23</td>
<td>178</td>
</tr>
<tr>
<td>$\frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}}$</td>
<td>11</td>
<td>52</td>
<td>412</td>
</tr>
<tr>
<td>$\sin(e^x)$</td>
<td>125</td>
<td>627</td>
<td>5103</td>
</tr>
</tbody>
</table>

Table 12. Number of segments required for Uniform Segmentation computed with $N=1,000,000$ for various values of $\varepsilon$.

2. Fixed Point Implementation

The fixed point implementation has a shorter pipeline depth. Numeric function $2^x$ has a pipeline depth of 31 in fixed point and 84 in floating point uniform...
segmentation. The multiplier inferred by the SRC accepts 64 bit operands and outputs a 64 bit product that contains only the lower 64 bits of the computed 128 bit product. This presents a challenge when computing in fixed point number system as discussed in section III.B.4.a.

Table 13 shows the fixed point implementation without any special adjustments to the bits. The function is $2^x$. The green portion of the table did not require any adjustment. In the yellow section, adjustments are required to eliminate the unintended sign extension of shifted values. The last two columns show the accuracy of the NFG. The very last column shows the accuracy when rounding is performed (rounding performed only in the final result, not at any intermediate points).

<table>
<thead>
<tr>
<th>Index</th>
<th>x Values</th>
<th>Excel $2^x$</th>
<th>NFG Approx</th>
<th>Accurate to x Bits</th>
<th>If rounding were performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6905840</td>
<td>104972342</td>
<td>1049722c3</td>
<td>23 bits</td>
<td>24 bits</td>
</tr>
<tr>
<td>1</td>
<td>d20c146</td>
<td>1094364e6</td>
<td>109436464</td>
<td>24 bits</td>
<td>24 bits</td>
</tr>
<tr>
<td>2</td>
<td>13b12a4d</td>
<td>10e051a07</td>
<td>10e051983</td>
<td>22 bits</td>
<td>24 bits</td>
</tr>
<tr>
<td>3</td>
<td>1a419353</td>
<td>112dca51f</td>
<td>112dca498</td>
<td>23 bits</td>
<td>24 bits</td>
</tr>
<tr>
<td>4</td>
<td>20d1fc5a</td>
<td>117ca6a6a</td>
<td>117ca69e0</td>
<td>22 bits</td>
<td>24 bits</td>
</tr>
<tr>
<td>5</td>
<td>27626560</td>
<td>11ccecff2</td>
<td>11ccecff66</td>
<td>24 bits</td>
<td>24 bits</td>
</tr>
<tr>
<td>6</td>
<td>2df2ce67</td>
<td>121ea3d94</td>
<td>121ea3d06</td>
<td>24 bits</td>
<td>24 bits</td>
</tr>
<tr>
<td>7</td>
<td>3483376e</td>
<td>1271d1d0a</td>
<td>1271d1c7b</td>
<td>23 bits</td>
<td>23 bits</td>
</tr>
<tr>
<td>8</td>
<td>3b13a074</td>
<td>12c67d9f5</td>
<td>12c67d962</td>
<td>24 bits</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>a41a41a4</td>
<td>18f374a1d</td>
<td>18f374959</td>
<td>22 bits</td>
<td>23 bits</td>
</tr>
<tr>
<td>25</td>
<td>aaaaaaab</td>
<td>1965fea54</td>
<td>1965feb1d</td>
<td>23 bits</td>
<td>23 bits</td>
</tr>
<tr>
<td>26</td>
<td>b13b13b1</td>
<td>19da96753</td>
<td>19da96689</td>
<td>22 bits</td>
<td>24 bits</td>
</tr>
<tr>
<td>27</td>
<td>b7cb7cb8</td>
<td>1a51457f7</td>
<td>1a51458c6</td>
<td>20 bits</td>
<td>21 bits</td>
</tr>
<tr>
<td>28</td>
<td>be5be5be</td>
<td>1aca155ce</td>
<td>1aca154fc</td>
<td>23 bits</td>
<td>24 bits</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>f2df2df3</td>
<td>1ee1ebf39</td>
<td>1ee1ec02c</td>
<td>21 bits</td>
<td>21 bits</td>
</tr>
<tr>
<td>37</td>
<td>f96f96f9</td>
<td>1f6fb0940</td>
<td>1f6fb084a</td>
<td>23 bits</td>
<td>23 bits</td>
</tr>
<tr>
<td>38</td>
<td>100000000</td>
<td>200000000</td>
<td>0</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 13. Fixed point implementation of $2^x$, no bit shifts, $N=1,000,000$ and $\epsilon = 2^{-24}$.  

61
Table 13 shows that the accuracy\(^3\) degrades in segments of higher index. This is expected because uniform segmentation results in segments that have varied accuracy. Figure 19 shows the error expected for uniform segmentation of \(2^x\), which is consistent with the results in Table 13. When implemented in hardware, this design does not meet the accuracy because the values are truncated at various intermediate points in the computation. The error propagates and magnifies the error in the result.

A bigger problem exists in indexing. In Table 13, the coefficients used to compute the NFG output for index 24, were actually coefficients intended for segment 25. The segment indexing failed to give the correct index. These problems contributed to the lower output accuracy as is seen in the second from last column in Table 13.

The advantage of using \(2^x\) is that all values are less than 1.0 except for the last value; \(x\) is 1.0. No integers to deal with in this example.

\[\text{Max Error} = 5.7966 \times 10^{-8} = 2^{-24.0402}.\]

\[\text{Error for UNIFORM } f(x)=2^x \text{ segmentation. No. of segs} = 39.\]

\[\text{Figure 19. Uniform Segmentation of } 2^x, N=1,000,000 \text{ and } \epsilon = 2^{-24}.\]

\(^3\) The endpoints of the segments are used as the \(x\) input values to test the numeric factions. The endpoints have the worst case approximation error. Table 13 shows the worst case scenario.
This implementation works for only a few functions. To make it work for the rest of the functions, a better method is required to handle integers and rounding.

Table 14 shows the implementation adjusted to accommodate the integers. As described in III.B.4.b, an arithmetic shift right (8 bits) is performed on the multiplication operands before multiplication. The product now has 16 bits to represent integer portion of the product. This is enough for all the values that will be encountered in the suite of functions investigated.

The worst case function is $\sqrt{-\ln(x)}$ of large coefficients. Whenever the coefficients are very large, the impact of small numbers is larger and therefore a greater room for errors exists. When the operands are shifted, the values are truncated which causes propagation of error to the product. Last column shows the accuracy.

<table>
<thead>
<tr>
<th>INDEX</th>
<th>x</th>
<th>x^2</th>
<th>a</th>
<th>ax^2</th>
<th>b</th>
<th>bx</th>
<th>c</th>
<th>fx</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1959</td>
<td>1f7bb7</td>
<td>3d32</td>
<td>82c6b81</td>
<td>21ad89ef</td>
<td>ffffffff8d5a05</td>
<td>ffffffff6439c766</td>
<td>1ebc291c2</td>
<td>17299e2df</td>
<td>25 bits</td>
</tr>
<tr>
<td>1959</td>
<td>1f7bb7c</td>
<td>3d323a</td>
<td>8b2b8218</td>
<td>21ad85ba</td>
<td>ffffffff8d5a05a1</td>
<td>ffffffff6439c516</td>
<td>1ebc291c2</td>
<td>17299e284</td>
<td>16 bits</td>
</tr>
<tr>
<td>1960</td>
<td>1f7fc3</td>
<td>3e30</td>
<td>8b9553</td>
<td>21ade5bd</td>
<td>ffffffff8d0de803</td>
<td>ffffffff6436ad30</td>
<td>1ecaa459b</td>
<td>1728e6e28</td>
<td>20 bits</td>
</tr>
<tr>
<td>1961</td>
<td>1f7fc3b5</td>
<td>3e0312a</td>
<td>8b09553aa</td>
<td>21ade6ddda</td>
<td>ffffffff8d0ddd03</td>
<td>ffffffff6436aa70</td>
<td>1ecaa459b</td>
<td>1728e6e66</td>
<td>20 bits</td>
</tr>
<tr>
<td>1961</td>
<td>1f83cf</td>
<td>3e12f</td>
<td>8ae7350</td>
<td>21af4950</td>
<td>ffffffff8d066eae1</td>
<td>ffffffff6432d489</td>
<td>1eca20868</td>
<td>172832241</td>
<td>21 bits</td>
</tr>
<tr>
<td>1962</td>
<td>1f83cfeee</td>
<td>3e1033a</td>
<td>8ae7350dd</td>
<td>21af49fbb</td>
<td>ffffffff8d066e1ad</td>
<td>ffffffff6432d005</td>
<td>1eca20868</td>
<td>172832868</td>
<td>21 bits</td>
</tr>
<tr>
<td>1962</td>
<td>1f87dc</td>
<td>3e22f6b</td>
<td>8ac5215bd</td>
<td>21af5200</td>
<td>ffffffff8d0edd1fd</td>
<td>ffffffff642f56a0</td>
<td>1ec9a4c9b</td>
<td>172831846</td>
<td>21 bits</td>
</tr>
<tr>
<td>1963</td>
<td>1f8be8</td>
<td>3e32e</td>
<td>8a517a7</td>
<td>21af09d</td>
<td>ffffffff8d07f33c</td>
<td>ffffffff642bd6d6</td>
<td>1ec9a82c8</td>
<td>172836e2f</td>
<td>19 bits</td>
</tr>
<tr>
<td>1963</td>
<td>1f8be86e1</td>
<td>3e32ebe</td>
<td>8a3a702</td>
<td>21af13f3c</td>
<td>ffffffff8d07f32c23</td>
<td>ffffffff642bd6f0d</td>
<td>1ec9a82c8</td>
<td>1726c721c</td>
<td>22 bits</td>
</tr>
<tr>
<td>1964</td>
<td>1f8e84</td>
<td>3e42e2</td>
<td>8a91f2c</td>
<td>21af742b</td>
<td>ffffffff8d0193f9</td>
<td>ffffffff642b8557</td>
<td>1ec9a61a5</td>
<td>17261a5d5</td>
<td>22 bits</td>
</tr>
<tr>
<td>1964</td>
<td>1f8ef4a</td>
<td>3e42e0</td>
<td>8a61f0</td>
<td>21af75d3</td>
<td>ffffffff8d0193990</td>
<td>ffffffff642b8672</td>
<td>1ec9a61a5</td>
<td>172611990</td>
<td>22 bits</td>
</tr>
<tr>
<td>1965</td>
<td>1f9400</td>
<td>3e52d</td>
<td>8af531f</td>
<td>21af1d03</td>
<td>ffffffff8d107f15</td>
<td>ffffffff6424ed03</td>
<td>1ec9100da</td>
<td>17255be00</td>
<td>17 bits</td>
</tr>
<tr>
<td>1965</td>
<td>1f9400d3</td>
<td>3e52dc4</td>
<td>8af531f3f</td>
<td>21af75a4</td>
<td>ffffffff8d107f15ca</td>
<td>ffffffff6424e90a</td>
<td>1ec9100da</td>
<td>17255c189</td>
<td>22 bits</td>
</tr>
<tr>
<td>1966</td>
<td>1f980d</td>
<td>3e62d</td>
<td>8a3d508</td>
<td>21b03549</td>
<td>ffffffff8d1104d2</td>
<td>ffffffff64217027</td>
<td>1ec914c</td>
<td>1724a68bc</td>
<td>23 bits</td>
</tr>
<tr>
<td>1966</td>
<td>1f980dca</td>
<td>3e62d</td>
<td>8a5d508a1</td>
<td>21b0395e</td>
<td>ffffffff8d1104d205</td>
<td>ffffffff642416f0c</td>
<td>1ec914c</td>
<td>1724a69e6</td>
<td>23 bits</td>
</tr>
<tr>
<td>1967</td>
<td>1f9c19</td>
<td>3e72d</td>
<td>8b7b9</td>
<td>21b09860</td>
<td>ffffffff8d12a4a</td>
<td>ffffffff641df863</td>
<td>1ec702a9</td>
<td>1723f136c</td>
<td>21 bits</td>
</tr>
<tr>
<td>1967</td>
<td>1f9c19</td>
<td>3e72d4</td>
<td>8a7b78d</td>
<td>21b0969d</td>
<td>ffffffff8d12a5e</td>
<td>ffffffff641df6f4</td>
<td>1ec702a9</td>
<td>1723f14a6</td>
<td>21 bits</td>
</tr>
<tr>
<td>1968</td>
<td>1fa025</td>
<td>3e82d</td>
<td>8f9b37</td>
<td>21b0a5d</td>
<td>ffffffff8d1020e89f</td>
<td>ffffffff641a80a5</td>
<td>1ec884590</td>
<td>17233c000</td>
<td>28 bits</td>
</tr>
<tr>
<td>1968</td>
<td>1faa52f7</td>
<td>3e82d44</td>
<td>8f9b3716</td>
<td>21b0ca5</td>
<td>ffffffff8d1020e5f7</td>
<td>ffffffff641a8e5f</td>
<td>1ec884590</td>
<td>17233c000</td>
<td>28 bits</td>
</tr>
<tr>
<td>1969</td>
<td>1fa431</td>
<td>3e92d</td>
<td>8d7f75</td>
<td>21b150f</td>
<td>ffffffff8d129545</td>
<td>ffffffff64179989</td>
<td>1ec60083c</td>
<td>172286cd4</td>
<td>24 bits</td>
</tr>
<tr>
<td>1969</td>
<td>1fa431b</td>
<td>3e92da5</td>
<td>8d7f754</td>
<td>21b151eb</td>
<td>ffffffff8d129543d</td>
<td>ffffffff64170607</td>
<td>1ec60083c</td>
<td>172286c5e</td>
<td>24 bits</td>
</tr>
<tr>
<td>1970</td>
<td>1fa834d</td>
<td>3e932d</td>
<td>8b647f</td>
<td>21b1ba9</td>
<td>ffffffff8d131a80</td>
<td>ffffffff64132921</td>
<td>1ec57cc71</td>
<td>1727d880</td>
<td>25 bits</td>
</tr>
<tr>
<td>1970</td>
<td>1fa834d</td>
<td>3e932d</td>
<td>8b647f6b</td>
<td>21b1bf29</td>
<td>ffffffff8d131a8026</td>
<td>ffffffff64138dd2</td>
<td>1ec57cc71</td>
<td>1727d9d6</td>
<td>25 bits</td>
</tr>
</tbody>
</table>

Table 14. Fixed point, uniform segmentation of $\sqrt{-\ln(x)}$, multiplier operands shifted by 8 bits, $N=1,000,000$ and $e=2^{-24}$. 

63
In Table 14, the first column is the index into the array. The rows show two computations; the NFG is in the colored row and the row below shows the correct values which have been computed in MATLAB and converted to the number representation. Coefficients \( a, b \) the input \( x \) and \( x^2 \) are shifted 8 bits in the NFG (colored rows). The intermediate products show the error in the intermediate steps. The two products; \( ax^2 \) and \( bx \) have been realigned before the final addition step. Table 14 shows the effect of the error as it propagates from the intermediate steps to the final answer. The last column shows the number of bits that match between the NFG output (in the colored row) and the desired output. This is basically telling how accurate the NFG has performed. As can be seen, there are instances where the error is large.

Table 15 shows the pipeline depth is 32. It also shows the summary of place and route and hardware resource requirements to implement uniform segmentation using fixed point numbers. This data is the same for all the numeric functions. The memory file determines which numeric function will be implemented.

<table>
<thead>
<tr>
<th>INNER LOOP SUMMARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop on line 54:</td>
</tr>
<tr>
<td>clocks per iteration: 1</td>
</tr>
<tr>
<td>pipeline depth: 32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PLACE AND ROUTE SUMMARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Slice Flip Flops: 8,751 out of 67,584 12%</td>
</tr>
<tr>
<td>Number of 4 input LUTs: 3,282 out of 67,584 4%</td>
</tr>
<tr>
<td>Number of occupied Slices: 5,226 out of 33,792 15%</td>
</tr>
<tr>
<td>Number of MULT18X18s: 40 out of 144 27%</td>
</tr>
<tr>
<td>freq = 100.0 MHz</td>
</tr>
</tbody>
</table>

Table 15. Pipeline depth and hardware resources for uniform implementation with no adjustments.

Table 16 is a comparison of uniform segmentation between the floating point and fixed point NFG implementations. They both require the same size memory files, but the floating point hardware can handle a larger range of values than the fixed point implementation.
### Table 16. Comparison of uniform segmentation NFG between fixed point and floating point.

<table>
<thead>
<tr>
<th></th>
<th>Floating Point</th>
<th>Fixed Point</th>
<th>Fixed Point / Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline Depth</td>
<td>84</td>
<td>32</td>
<td>38 %</td>
</tr>
<tr>
<td># of Slice Flip Flops</td>
<td>26%</td>
<td>12%</td>
<td>46 %</td>
</tr>
<tr>
<td># of 4 input LUTs</td>
<td>13%</td>
<td>4%</td>
<td>31 %</td>
</tr>
<tr>
<td># of occupied Slices</td>
<td>33%</td>
<td>15%</td>
<td>45 %</td>
</tr>
<tr>
<td># of Block RAMs</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td># of MULT18X18s</td>
<td>44%</td>
<td>27%</td>
<td>61 %</td>
</tr>
<tr>
<td>Freq</td>
<td>100.2 MHz</td>
<td>100.1 MHz</td>
<td>0%</td>
</tr>
</tbody>
</table>

#### B. NON-UNIFORM SEGMENTATION

Non-uniform segmentation requires a segment index encoder. The SRC programming environment has a priority selector macro that is used as the segment index encoder for the NFG.

1. **Floating Point Implementation**

The priority selector macro in the SRC, is used as the segment index encoder. The priority selector has a limit (approximately 150 elements) when used in the NFG with three 64 bit multipliers. The non-uniform segmentation NFG, in floating point, has a pipeline depth of 74.

The math macros available in the SRC have pipeline depths that vary. For example, \( \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \) implemented using the math macros has a pipeline depth of 274 as shown in Table 17. Table 17 summarizes the hardware pipeline depth for the suite of numeric functions. The table shows side by side comparisons of the pipeline depth for the NFG and the SRC math macros. In 10 of the 15 functions, the pipeline depth is smaller. For one function the pipeline depths are the same and for 4 of the functions the NFG pipeline depth is larger. Regardless of the size of the function, the NFG has the same pipeline depth; the only exception is \( \sin(e^x) \). It is only one clock longer.
Three functions in Table 17 are limited by the number of segments required. In the floating point implementation with 3 multipliers and the other hardware requirements, the FPGA runs out of resources to build large priority selectors. The priority selectors were limited to approximately 150 segments. Implementations requiring larger selectors did not compile on the MAP. The data was obtained by compiling in debug mode. Some of the implementations were built in hardware, for example: $\frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$.

| Numeric Function | Macro Pipeline Depth | NFG Pipeline Depth | Number of Segments
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^x$</td>
<td>132</td>
<td>74</td>
<td>35</td>
</tr>
<tr>
<td>$1/x$</td>
<td>70</td>
<td>74</td>
<td>50</td>
</tr>
<tr>
<td>$\sqrt{x}$</td>
<td>43</td>
<td>74</td>
<td>24</td>
</tr>
<tr>
<td>$1/\sqrt{x}$</td>
<td>74</td>
<td>74</td>
<td>36</td>
</tr>
<tr>
<td>$\log_2(x)$</td>
<td>73</td>
<td>74</td>
<td>44</td>
</tr>
<tr>
<td>$\ln(x)$</td>
<td>61</td>
<td>74</td>
<td>39</td>
</tr>
<tr>
<td>$\sin(\pi x)$</td>
<td>105</td>
<td>74</td>
<td>58</td>
</tr>
<tr>
<td>$\cos(\pi x)$</td>
<td>105</td>
<td>74</td>
<td>58</td>
</tr>
<tr>
<td>$\tan(\pi x)$</td>
<td>135</td>
<td>74</td>
<td>58</td>
</tr>
<tr>
<td>$\sqrt{-\ln(x)}$</td>
<td>127</td>
<td>74</td>
<td>163$^4$</td>
</tr>
<tr>
<td>$\tan^2(\pi x) + 1$</td>
<td>254</td>
<td>74</td>
<td>79</td>
</tr>
<tr>
<td>$-(x \log_2 x + (1-x) \log_2 (1-x))$</td>
<td>114</td>
<td>74</td>
<td>183$^4$</td>
</tr>
<tr>
<td>$\frac{1}{1+e^{-x}}$</td>
<td>185</td>
<td>74</td>
<td>20</td>
</tr>
<tr>
<td>$\frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$</td>
<td>274</td>
<td>74</td>
<td>45</td>
</tr>
<tr>
<td>$\sin(e^x)$</td>
<td>212</td>
<td>75</td>
<td>265$^4$</td>
</tr>
</tbody>
</table>

Table 17. Pipeline depth for various implementations of using the available macros or the NFG in floating point number system.

$^4$ Note that these numbers (number of segments) are larger than 150, and cannot be realized in priority selector in the floating point implementation.
When both the NFG and the macro are built on the FPGA, a large amount of resources are consumed and the frequency may be affected due to place and route difficulties and increased delay in the wiring. Figure 20 shows the summary of the place and route when numeric function \( \frac{1}{\sqrt{2\pi}} e^{\frac{x^2}{2}} \) is implemented with the macros and the NFG both on the same FPGA. The frequency is 77.2MHz.

\[
\frac{1}{\sqrt{2\pi}} e^{\frac{x^2}{2}}
\]

---

**Figure 20.** NFG and macro both built on the FPGA for numeric function; \( \frac{1}{\sqrt{2\pi}} e^{\frac{x^2}{2}} \).
The performance improves if only one is built at a time. Figure 21 shows the same function built on the FPGA using the NFG only. The frequency is 100.0MHz.

```
loop on line 53:
clocks per iteration: 1
pipeline depth: 74
```

**Place and Route Summary**

- Number of Slice Flip Flops: 26,377 out of 67,584 (39%)
- Number of 4 input LUTs: 16,386 out of 67,584 (24%)
- Number of occupied Slices: 17,473 out of 33,792 (51%)
- Number of MULT18X18s: 48 out of 144 (33%)
- freq = 100.0 MHz

Figure 21. NFG built on the FPGA for numeric function: \( \frac{1}{\sqrt{2\pi}} e^{\frac{x^2}{2}} \).

Table 18 shows the results from computing \( \frac{1}{\sqrt{2\pi}} e^{\frac{x^2}{2}} \), with \( N=1,000,000 \) and \( \epsilon = 2^{-24} \). The values are displayed to twelve decimal places. This function requires 45 segments. The values of \( x \) that are tested in Table 18 are the endpoints of the segment and therefore have the worst case approximation error. At the very bottom of Table 18 is \( \epsilon = 2^{-24} \) in decimal. The last column shows the approximation error is consistently smaller than \( \epsilon \); per the design.

5 If the \( x \) input to the NFG were somewhere in the middle of the segment, the approximation error would be smaller. There are four points in a segment with worst case approximation error. Figure 10 is a good example to see the distribution of the approximation error on a non-uniform segment.
Table 18. Comparison between SRC macro and NFG; numeric function $e^{-\frac{x^2}{2}}$, \(N=1,000,000\) and $\epsilon = 2^{-24}$.

2. Fixed Point Implementation

As mentioned before, the advantage of using fixed point is the reduction in hardware and the reduced pipeline depth. The disadvantage is that is takes more work to program.

Macros may be used to define certain behavior that is easier to describe in HDL or to provide special functionality that is not available in regular programming. In the NFG, the multiplier is limited by the 64 bit architecture. The product of two 64 bit
numbers does not give the user access to all 128 bits in the product. HDL can be used to manipulate and access the desired bits.

### a. No Macro Multiplier (non-uniform)

The fixed point implementation without a macro is exactly the same as the fixed point implementation with only one exception; the indexing in non-uniform segmentation is accomplished using the user callable macro, priority selector, available in the SRC.

```plaintext
loop on line 46:
clocks per iteration: 1
pipeline depth: 28
```

Table 19. Pipeline depth, place and route summary for \(\sqrt{-\ln(x)}\), N=1,000,000 and \(e = 2^{-24}\).

Non-uniform segmentation using priority selector macro.

### b. Macro Multiplier Implementation

The goal is to build a multiplier in VHDL or Verilog that can successfully multiply in two’s complement and provide a result that is already shifted into the number system chosen for fixed point. Specifically, we want a product that is 32 bits integer and 32 bits fraction.

Several multipliers were built. The multipliers function correctly in simulation on PC’s using Xilinx ISE, Project Navigator and Modelsim simulating software. However, when the VHDL or Verilog files were compiled on the SRC, the products were not correct. This version was implemented, but it did not produce correct products.
Appendix B shows the VHDL code for a 32x32 bit multiplier with a 32 bit product. The design instantiates the 18x18 signed multiplier primitive. The design makes use of a modified I/O pipeline design from a Xilinx application note [22].

Appendix B also shows the Verilog file for a 64x64 bit multiplier with a 64 bit product. The 64x64 bit multiplier makes use of the source code for the 64x64 bit multiplier macro designed by SRC.

C. SOURCES OF ERROR

The floating point implementation has only errors associated with the MATLAB computed values and the restrictions placed on \( \epsilon \). When implemented in the SRC, double precision accurately represents what is expected from the values fed into the NFG and the coefficients table.

The fixed point implementation had errors due to several reasons. We explore some of those reasons for error in the NFG as a whole.

1. Function Approximation

Both floating point and fixed point have to work with approximation error. This is discussed in detail in section II B (Segmentation).

2. Absence of Rounding in the Multiplier

The fixed point implementation of the NFG shifts binary bits and truncates lower and upper bits. This introduces error in computing the products and these errors propagate to the final answer.

3. Insufficient Bits

Insufficient bits to represent the full product means that the numbers have to be shifted and truncated. This limits the ability for the NFG.
D. SUMMARY

The NFG implementation of the uniform segmentation using floating point number system has a pipeline depth of 84 or 98 depending on whether the begin point of the domain interval is zero or non-zero (zero is preferred). This implementation must read a memory file containing the polynomial coefficients into OBM. Aside from these requirements, the NFG implemented in uniform segmentation and floating point number systems, provides advantages over using the available user callable macros and the math operators. It can be implemented in very high precision, shorter pipeline depth and in some cases less hardware.

Another advantage of the uniform segmentation is that once compiled, the NFG can compute any of the 15 functions. The memory file with the coefficients must be available.

The NFG non-uniform implementation has a shorter pipeline depth, but requires much hardware to implement the segment index encoder. The segment index encoder is limited to approximately 150 segments in this design. Depending on the function, the precision can be increased as long as the number of segments does not exceed approximately 150.

The fixed point implementation requires a rounding macro and a good macro multiplier to provide the desired product bits and make it effective. However, it provides a significantly smaller pipeline depth than the floating point implementation.

A real advantage of the NFG is when very complicated numeric functions need to be implemented; the NFG has a constant pipeline depth unlike the more complicated functions that have long pipeline depths.

More research is required to realize a complete NFG design. Section VI discusses some suggestions for future work.
VI. CONCLUSION

A. SUMMARY OF WORK

An efficient and fast segmentation of numeric functions was accomplished in MATLAB. Table 20 shows the number of tests (calls to `chebyRemz`) required to segment the suite of 15 functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Interval</th>
<th>% Of tests</th>
<th># of Segments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^x$</td>
<td>[0,1]</td>
<td>0.00910</td>
<td>35</td>
</tr>
<tr>
<td>$1/x$</td>
<td>[1,2]</td>
<td>0.01020</td>
<td>50</td>
</tr>
<tr>
<td>sqrt(x)</td>
<td>[1,2]</td>
<td>0.00750</td>
<td>24</td>
</tr>
<tr>
<td>1/sqrt(x)</td>
<td>[1,2]</td>
<td>0.00720</td>
<td>36</td>
</tr>
<tr>
<td>log2(x)</td>
<td>[1,2]</td>
<td>0.00900</td>
<td>44</td>
</tr>
<tr>
<td>log(x)</td>
<td>[1,2]</td>
<td>0.00780</td>
<td>39</td>
</tr>
<tr>
<td>sin(pi*x)</td>
<td>[0,1/2]</td>
<td>0.01990</td>
<td>58</td>
</tr>
<tr>
<td>cos(pi*x)</td>
<td>[0,1/2]</td>
<td>0.01740</td>
<td>58</td>
</tr>
<tr>
<td>tan(pi*x)</td>
<td>[0,1/4]</td>
<td>0.01240</td>
<td>58</td>
</tr>
<tr>
<td>sqrt(-log(x)</td>
<td>[1/512,1/4]</td>
<td>0.04070</td>
<td>163</td>
</tr>
<tr>
<td>tan(pi*x).^</td>
<td>[0,1/4]</td>
<td>0.02180</td>
<td>79</td>
</tr>
<tr>
<td>-(x*log2(x))</td>
<td>[1/256,1-1/256]</td>
<td>0.04710</td>
<td>183</td>
</tr>
<tr>
<td>1/(1+exp(-x)</td>
<td>[0,1]</td>
<td>0.00920</td>
<td>20</td>
</tr>
<tr>
<td>(1/sqrt(2*p)</td>
<td>[0,sqrt(2)]</td>
<td>0.01670</td>
<td>45</td>
</tr>
<tr>
<td>sin(exp(x))</td>
<td>[0,2]</td>
<td>0.07810</td>
<td>265</td>
</tr>
</tbody>
</table>

Table 20. Speed-up in computation time for 15 functions (expressed as a percentage of the time needed when the domain is divided into 1,000,000 points) for $\epsilon = 2^{-24}$

The NFG circuit built in the SRC was very effective in floating point. The computation of numeric functions in the NFG was shown to obtain accuracy of up to 33 bits. Higher accuracy is possible at the cost of increasing the size of the memory files required to store the coefficients.

Floating point implementation was easier to build on the SRC than the fixed point implementation. However, floating point implementation takes up a large amount of FPGA resources.
The NFG is a useful technique to compute complicated numeric functions that would otherwise require a combination of several other arithmetic operations. The more demanding the numeric function the more reason to use the NFG instead. The NFG is more efficient in 10 out of the 15 functions that were investigated in this thesis (when using the non-uniform segmentation).

The fixed point implementation did not produce all of the desired results. The multiplication required more programming than the floating point implementation required, but the results had errors due to rounding and truncating the intermediate and final results. This area needs more research to improve. The advantage of fixed point implementation is that it requires much less hardware than floating point and therefore can reduce the pipeline depth to about 30% of the pipeline depth required by the floating point implementation.

B. SUGGESTED FUTURE WORK

1. Hybrid of Uniform and Non-Uniform Segmentation

Uniform segmentation is much faster and less complicated than non-uniform segmentation. Although non-uniform segmentation may not be practical on its own, a hybrid of non-uniform and uniform segmentation would take advantage of the strengths of each.

Consider a numeric function that is not suitable for uniform segmentation, such as $\sqrt{-\ln(x)}$, which appears in Figure 4 to demonstrate this fact. In the non-uniform segmentation of the same function; such as Figure 2, the restricting portion is the beginning of the segment. Therefore to capture the most restricting part of the numeric function, segment the numeric function into a few non-uniform segments.

A good starting point is to determine an upper limit for the total number of constant segments. Let us decide on 400 segments. If we dedicate 100 constant segments to the first portion of the numeric function $\sqrt{-\ln(x)}$, then change the segment size for another 100 constant segments and repeat this process four or five times, we will have five non-uniform segments each containing a set of uniform segments.
This method would provide three advantages:

1. Relieve the segmentation constraint from the most restricting segment.
2. The segment index encoder would be small (5 groups of segments) and save FPGA space.
3. The indexing would be less complex once the input has been mapped to the correct group of segments.

2. **Expand the Domain of the NFG via Mapping**

The functions investigated in this thesis have a limited domain interval. To make the functions useful for a wide range of applications, the domain interval should be increased. Theoretical research is being conducted in this field [21].

3. **Build an HDL Multiplier Macro and Tap of Desired Bits**

If the multiplier in fixed point were built in a macro, the desired bits could be tapped off. This implementation would be both fast and accurate.

3. **Build a Rounding Macro**

A macro can be built to round off shifted values in the fixed point implementation instead of truncating the values. This would improve the accuracy in the output of the products and the final result of the NFG.

4. **Efficient Segment Index Encoder vice Priority Selector Macros**

The priority selectors are fast and work well, but take up a lot of hardware. Combined with the other hardware in the NFG, the priority selectors take up all the resources and limit the accuracy and flexibility of the NFG to handle all the functions. An implementation that uses a more efficient method for the segment index encoder would benefit the NFG.

Sasao, Butler have three suggestions; (1) LUT cascade, (2) Content addressable memory and (3) EVBDD.
5. Different Architecture

If FPGA resources became scarcer and one wanted to implement a larger coefficients table, the only way to make room is to remove the major consumers of real estate. In the NFG, it would be the segment index encoder that is implemented as a priority selector macro and the multipliers. We have already discussed possible solutions to removing large selectors.

Using Horner’s rule, a multiplier can be eliminated from the NFG. Equation (0.5) shows how to apply Horner’s rule to the NFG.

\[
f(x) = c_2 x^2 + c_1 x + c_0 = (c_2 x + c_1) x + c_0 \tag{0.5}
\]

The NFG hardware would add one more adder stage, however if the segment index encoder were able to work in one or two clocks, this would be a speed-up from the previous architecture as long as the adder stages take fewer clocks than the multipliers. Floating point adders can take as many clocks as the multipliers, but in two’s Complement or signed magnitude, the adders are faster than the multiplier.

In the previous architecture, \(x^2\) takes many more clocks than the segment index encoder and adds to the pipeline depth.

Figure 22 shows an overview of the NFG architecture when Horner’s rule has been applied.
Figure 22. Horner’s rule NFG architecture overview.

\[ f(x) = C_2 x^2 + C_1' x + C_0' \]
APPENDIX A. MATLAB ALGORITHMS

The following MATLAB Code generates the segmentation for any function; however a user interface has been added for convenience. The user simply picks a number instead of re-typing the entire function or the interval for evaluation. The interface limits the MATLAB Code to the suite of functions found in Table 1.

A.1 QUADRATIC APPROXIMATION USING POLYFIT

This code implements the quadratic approximation using the MATLAB function Polyfit. There are 6 files needed to run the non-uniform and uniform segmentation: QuadAppxPfit.m, multipleQuadApprox.m, varQuadApprox.m, dec2binfp.m, constantQuadApprox.m, and constQuadAppxWErr.m.

QuadAppxPfit.m is the top function where the program starts and ends. All the other files are child functions that provide the segmentation data back to this file for presentation / file storage.

multipleQuadApprox.m calls the non-uniform segmentation algorithms to collect the data for the segment endpoints and coefficients.

varQuadApprox.m tests proposed segments and reduces finds the optimum width of the segment by testing the approximation error to \( \epsilon \).

dec2binfp.m is the file that converts decimal numbers into binary. This is limited to converting one integer value and only up to 9 binary bits of accuracy.

constantQuadApprox.m is used for uniform segmentation when the number of segment is known before hand. The key requirement is to input the number of segments desired, the approximation error is unspecified.

constQuadAppxWErr.m needs to have \( \epsilon \) specified, then this file will compute the uniform segmentation of the numeric function that meets the constraint \( \epsilon \).
% Arbitrary_PW_Quadratic_Approx.m
% Created: January 6, 2006 (from Arbitrary_PW_Linear_Approx.m)
% Last modified: October 20, 2006
% Produced by: Tom Mack & Jon Butler
% Modified by: NJuguna Macaria for quadratic approximation

% This program produces a segmentation of a given function using either:
% 1. Uniform piecewise Quadratic approximation
% 2. Non-uniform piecewise Quadratic approximation
% 3. Both

% It is based on the algorithm:
% 1. For non-uniform, the MATLAB polyfit function
% 2. For uniform, dividing the range of the input into
%    equal, user-defined segments
%    or by using max error to determine max segment length
%    at the greatest curvature and then dividing the range
%    up into equal segments.
%    All with intercept shifting to balance the positive
%    and negative error

% Inputs
% N - number of elements on which function is expressed
% f(x) - function to be evaluated
% x_low - low end of interval over which f(x) is evaluated
% x_high - high end of interval over which f(x) is evaluated
% epsilon - precision of approximation (for variable only)
% consegs - number of segments to use to approximate (constant only)

% Outputs
% Segment info - Segment #, Begin Pt, End Pt, Coefficients, & Error
% Plot showing the approximation
% Text file used to initialize memory in SRC (both Binary & Decimal)

%%%%%%%%%%%%%%%%%%%%% INPUT OF USER-SPECIFIED PARAMETERS %%%%%%%%%%%%%%%%%%
clear
close all
format long g
fprintf('
'                                                        )
fprintf('
**************************************************************')
fprintf('
'                                                        )
fprintf('
     QUADRATIC APPROXIMATION OF A FUNCTION USING POLYFIT ')  
fprintf('
'                                                        )

%% Get FUNCTION to be approximated (user input)
func = input( 'Input the Function, func[sqrt(-1*log(x))]:  ','s');
if isempty(func)
    func = 'sqrt(-1*log(x))';   %% default
end

%% Get LOW range (user input)
x_low = input( 'Input the Lower Range of x - LOW value, x(low)[1/256]:  ');  
if isempty(x_low)
    x_low = 1/256;                  %% default
% Get HIGH range (user input)
x_high = input('Input the Higher Range of x - HIGH value, x(high)[1/4]:');
if isempty(x_high)
    x_high = 1/4;                 %% default
end

% Get CONSTANT OF VARIABLE segmentation (User input)
vari_or_const = 0;
while vari_or_const ~= 1 && vari_or_const ~= 2 && vari_or_const ~= 3
    vari_or_const = ...
    Input( '(1)Non-uniform (2)Uniform Segmentation or (3)Both [1]:');
    if isempty(vari_or_const)
        vari_or_const = 1;      %% default Non-uniform
    end
end

% If non-uniform segmentation, then enter ERROR parameters
if vari_or_const ~= 2
    epsilon = input('Input the Desired Error, epsilon[0.0001]:  ');
    if isempty(epsilon)
        epsilon = 0.0001;        %% default
    end
end

% If uniform segmentation, find how the user will restrict # of segments
if vari_or_const == 2
    err_or_segs = ...
    Input( 'Constrain by (1)Number of Segments or (2)Error [1]:   ');
    if isempty(err_or_segs)
        err_or_segs = 1;        %% default
    end
    if err_or_segs == 1
        consegs = input('Input the number of Desired Segments[100]:  ');
        if isempty(consegs)
            consegs = 200;       %% default
        end
    end
    if err_or_segs == 2
        epsilon = input('Input the Desired Error, epsilon[0.0001]:  ');
        if isempty(epsilon)
            epsilon = 0.0001;    %% default
        end
    end
end

N = input('Input the no. of pts the fct is to be evaluated; N[10000]:  ');
if isempty(N)
    N = 10000;                  %% default
end

% eqn = input( 'Input the equation to use:
% (1)F(x)=ax^2+bx+c or (2)F(x)=a(x-p)^2+b(x-p)+c, [1]:  ');
% if isempty(eqn)
eqn = 1;  

%%% Based on the number of points to be used for the curve, find the
%%% x values to calculate and spread over the approximating function
N = N * (x_high - x_low);

x = linspace(x_low, x_high, N);

eval(['F = ', func, ';'])  % Evaluate the function and place values in F

% Print demarcation line
fprintf('
**************************************************************
')

%%%%%%%%%%%%%%%%%%%%%%%%  Segmentation Algorithm %%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%   REPEAT FOR EACH i   %%%%%%%%%%%%%%%%%%%%%%%%%%
repeat = 1;
while repeat == 1
  if (mod(vari_or_const,2) == 1)
    [endpt,seg_end_point,c_2,c_1,c_0] = multipleQuadApprox(x,F,epsilon);
  end
  if (vari_or_const == 2) && (err_or_segs == 1)
    [endpt,seg_end_point,c_2,c_1,c_0] = constantQuadApprox(x,F,consegs);
  end
  if ((vari_or_const == 2) && (err_or_segs == 2)) || (vari_or_const == 4)
    [endpt,seg_end_point,c_2,c_1,c_0] = constQuadAppxWErr(x,F,epsilon);
  end

% Compute and plot function, approximate function and error
ind = 1;                             % Index for each segment
for i = 1:length(seg_end_point);
  m     = 1;                       % Index within each segment
  XP    = [];
  FP    = [];
  Error = [];
  while (ind < seg_end_point(i))
    XP(m)    = x(ind);
    FNC(m)   = F(ind);           % Actual function (Fct No correction)
    FP(m)    = c_2(i)*((x(ind)).^2)+c_1(i)*x(ind) + c_0(i); % Approx
    Error(m) = FNC(m) - FP(m);
    ind      = ind + 1;
    m        = m + 1;
  end %while
  MaxError(i) = max(abs(Error));          % Keep track of all errors
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if (mod(i,2) == 0)  % Plot every other segment a different color
    figure(mod(vari_or_const,2)+1)  %% Blue
    plot(XP,FP)
    figure(mod(vari_or_const,2)+3)  %% Blue
    plot(XP,Error)
else
    figure(mod(vari_or_const,2)+1)
    plot(XP,FP,'r','LineWidth',2)  %% Red
    figure(mod(vari_or_const,2)+3)
    plot(XP,Error,'r','LineWidth',2)  %% Red
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 1
% integer bit and 8 fraction bits and print results table.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if (mod(vari_or_const,2) == 1)
    fprintf(\'NON-UNIFORM Segmentation\'
if eqn == 1
    fprintf(\'Segment   End Point   End Point           c_2     \','...
    'c_2                  c_1        c_1                 c_0  ',...
    'c_0')
    fprintf(\'Number    (Decimal)   (Binary)            (Decimal)',...
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_2bin(i) = dec2binfp(c_2(i)); 
    c_1bin(i) = dec2binfp(c_1(i)); 
    c_0bin(i) = dec2binfp(c_0(i)); 
    if eqn == 1
        % Print Remaining Results Table 
        fprintf('n

'); 
        fprintf('%3d %8.6f %019.9f %10.5f %019.9f %10.5f %019.9f %10.5f %019.9f', i-1, x(seg_end_point(i)), xbin(i), c_2(i), c_2bin(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 of Binary values to initialize memory 
memBin = [c_2bin .* 10^9; c_1bin .* 10^9 ; c_0bin .* 10^9]; % Memory with bin 
fid = fopen('memory.mem','w'); 
fprintf (fid,'n%018.0f%018.0f%018.0f',memBin); 
fclose (fid); 

% Create text file of Decimal Values to initialize memory 
fid = fopen('memDEC.mem','w'); 
format long g; 
fprintf(fid,'%5d', length(seg_end_point)); % Number of Segments 
memDEC = [segment(2:end); c_2; c_1; c_0] 
fprintf(fid,'n%18.12f %18.12f %18.12f %18.12f',memDEC); 
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.  Squared term coefficient 
% 2.  Linear term coefficient 
% 3.  Constant 
% 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('
')
segment(i) = 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.
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 end of segment.
end %for i = 2:length(segment)
fprintf('
DECIMAL values for Approx = slope*(x - pivot) + intercept.')
fprintf('
seg no.  [s,  e]         slope   intercept     pivot  approx_error    e-s    (e-s)*slope (e-s)*sl+intercept exact f(x)
')
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
', 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('
BINARY values')
fprintf('
seg no.      [s,  e]             slope      intercept',...
    'approx_error e-s    (e-s)*slope (e-s)*sl+intercept exact f(x)
')
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
% End file: QuadAppxPfit.m
function [endpt, indx, c2, c1, c0] = multipleQuAdapprox(x, fct, max_error)

% This function will produce multiple Quadratic-line approximations of a
given function to within the bounds of max error provided.
% Created by Tom Mack for linear approximations
% Created: Mar 31, 2006
% Modified for Quadratic approximations by Njuguna Macaria
% Modified: Dec 30, 2006

i      = 1;
indx   = 1;
seg_no = 1;
endpt  = [];
c2     = [];
c1     = [];
c0     = [];

while i < length(fct)
    [endpt(seg_no), indx(seg_no), c2(seg_no), c1(seg_no), c0(seg_no)] =
        varQuadApprox(x, fct, max_error, i);
    i      = indx(seg_no) + 1;
    seg_no = seg_no + 1;
end
function [endpt,i,c2,c1,c0] = varQuadApprox(x,fct,max_error,indx)
% This function creates a 2nd Order approximation of a given function
% using the polyfit function. It continues to calculate polyfits until
% maximum error is exceeded.
% Linear approximation Created by Tom Mack >> Mar 31, 2006
% Modified for Quadratic approximation by Njuguna Macaria
% Modified: Dec 29, 2006

for i=indx:length(fct);
    p = polyfit(x(indx:i),fct(indx:i),2); % Fit equ to 2nd order poly
    c_2(i) = p(1);                        % Coefficient of X^2
    c_1(i) = p(2);                        % Coefficient of X
    c_0(i) = p(3);                        % Intercept of polynomial
    approx(indx:i) = p(1)*(x(indx:i)).^2 + p(2)*x(indx:i) + p(3);
    errors         = approx(indx:i) - fct(indx:i);
    %    maxposerror    = max(errors);
    %    maxnegerror    = min(errors);
    %    c_0delta(i)    = abs((abs(maxposerror) - abs(maxnegerror))/2);
    % %    % If the negative error is bigger, then the delta should be negative
    % %    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 exceeded the max error, then go back to the previous endpoint
    if error > max_error
        i     = i-1;
        endpt = x(i);
        c2    = c_2(i);
        c1    = c_1(i);
        c0    = c_0(i);
        return
    end % if error > max
end % for i=indx+1:length(fct)

endpt = x(i);
c2    = c_2(i);    % Removed i = i-1;
c1    = c_1(i);
c0    = c_0(i);
function [binfp] = dec2binfp(x,n)
% This 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) = bit resolution to the right and left of decimal pt
% Outputs
%   binfp = binary floating point representation
%   Negative inputs are output in 18-bit (9.9) 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);
    x = dec2bin(x,18);
    x = str2double(x);
    x = x / 10^n;
    binfp = x;
    return
else
    x = x * 2^n;
    x = dec2bin(x,18);
    x = str2double(x);
    x = x / 10^n;
    binfp = x;
end
function [endpt, indx, c2, c1, c0] = constQuadAppxWErr(x, fct, max_error)

% This function will produce multiple Quadratic-line approximations of a constant size of a given function to within the bounds of the max error provided. Coefficients & intercept calculated using polyfit. Intercept adjusted to balance max positive and negative errors.
% Created by Tom Mack for linear approximations
% Created: July 10, 2006
% Modified: July 11, 2006
% Modified again by Njuguna Macaria for Quadratic approximations
% Modified: Dec 30, 2006
%
% Compute # of segs

firstderiv = diff(fct)./diff(x);
secndderiv = diff(firstderiv)./diff(x(1:length(firstderiv)));
dermax = 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), 2);
    approx(i_low:i_high) = p(1)*(x(i_low:i_high)).^2 + p(2)*x(i_low:i_high) + p(3);
    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);
% Figure out if the error is positive or negative and move the function
% to compensate and balance the error of the approximated function
if abs(maxnegerror) > abs(maxposerror)
    c_0delta = -1 * c_0delta;
end

% Re-check the error and find the max error
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 is larger than should be
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 Coefficients 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)),2);
    approx(idx:indx(i)) = p(1)*(x(idx:indx(i))).^2 + ...
                          p(2)*x(idx:indx(i)) + p(3);
    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
    c2(i) = p(1);
    c1(i) = p(2);
    c0(i) = p(3) - c_0delta;  % Constant shift to balance pos & neg error
    idx   = indx(i)+1;
    i     = i+1;
end
function [endpt, indx, c2, c1, c0] = constantQuadApprox(x, fct, constsegs)

% This function will produce multiple Quadratic line approximations of a
% given function to within the bounds of the number of segments provided.
% Coefficients calculated by polyfit. Intercept adjusted to balance
% maximum positive and negative errors.
% Created by Tom Mack for linear approximations
% Created: June 4, 2006
% Modified for Quadratic approximations by Njuguna Macaria
% Modified: July 11, 2006
%

idx = 1;

for i = 1:constsegs
    indx(i) = round((length(x)/constsegs)*i);
    if i == constsegs
        indx(i) = length(x);
    end
    endpt(i) = x(indx(i));
    p = polyfit(x(idx:indx(i)), fct(idx:indx(i)), 2);

    approx(idx:indx(i)) = p(1)*(x(idx:indx(i))).^2 + p(2)*x(idx:indx(i)) + p(3);
    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
    c2(i) = p(1);
    c1(i) = p(2);
    c0(i) = p(3) - c_0delta;  % Intercept shift to balance pos & neg error
    idx = indx(i)+1;
    i = i+1;
end
A.2 QUADRATIC APPROXIMATION USING REMEZ ALGORITHM

The thesis was designed using the Remez algorithm. The following files were developed to compute the segmentation. The top level file is QuadAppxRemz.m, which calls a set of user written MATLAB functions to display and request the user input (UserInput.m), obtain the numeric functions selected by the user and their respective domain intervals (getF.m) and then compute the segmentation.

Non-uniform segmentation was performed by multipleQuadApprox.m in conjunction with varQuadApproxHyb3AvgThird.m and chebyRemz.m. chebyRemz.m takes place of Polifit.m that is an optimized user callable MATLAB function shown in A.1 above.

Uniform segmentation is performed by two other files. If the number of segments is known without explicit input of \( \varepsilon \), then constantQuadApprox.m is the file that is used. If on the other hand, \( \varepsilon \) is defined and uniform segmentation is desired, then constQuadAppxWErr.m is the file that is used.

The file twosComp.m was developed to convert the data to a two’s complement, fixed point binary, hexadecimal or decimal number. Note the two’s complement decimal number is not the same as a float or double data type.
This program produces a segmentation of a given function using either:
1. Uniform Quadratic approximation
2. Non-uniform piecewise Quadratic approximation
3. Both

It is based on the algorithm:
1. For non-uniform, the MATLAB Remez algorithm
2. For uniform, dividing the range of the input into equal, user-defined segments
   or by using max error to determine max segment length at the greatest curvature and then dividing the range up into equal segments.

Inputs
Inputs are taken from an input function; "userInput();"
N - number of elements on which function is expressed
eqn - (1)F(x)=ax^2+bx+c OR PIVOT: (2)F(x)=a(x-p)^2+b(x-p)+c
x_low - low end of interval over which f(x) is evaluated
x_high - high end of interval over which f(x) is evaluated
func(x) - function to be evaluated
epsilon - precision of approximation (for variable only)
consegs - number of segments to use to approximate (constant only)
err_or_segs - Constant segmentation; decide # of segments or err bound
vari_or_const - Variable or constant segmentation

Outputs
Segment info - Segment #, Begin Pt, End Pt, Coefficients, & Error
Plot showing the approximation
Text file used to initialize memory in SRC (both Binary & Decimal)

%%%%%%%%%%%%%%%%%%%%% INPUT OF USER-SPECIFIED PARAMETERS %%%%%%%%%%%%%%%%%%%
clear
clc
close all
format long g;

% Get user input
% profile on % For use when debugging. Find runtimes
sel = UserInput();
[f,interval,vari_or_const,err_or_segs,consegs,epsilon,N]=getF(sel);

%%% Based on the number of points to be used for the curve, find the
%%% x values to calculate and spread over the approximating function
syms x
eval(['func = ', f, ';'])
eval(['intv = ', interval, ';'])
x_pts = linspace(intv(1), intv(2), N);
vecFunc = inline(vectorize(func)); % Vectorized version of func.
y_actual = vecFunc(x_pts); % Evaluate the function with x_pts

% The segments in this program overlap (i.e. the first element of % the NEXT segment IS the last element of the LAST segment. %

% Print demarcation line
fprintf('
********************************************************
')
fprintf('
')
fprintf('%%%%%%%%%%%%%%%%%%%%%%%% Segmentation Algorithm %%%%%%%%%%%%%%%%%%%
')
repeat = 1;
while repeat == 1
  if (mod(vari_or_const,2) == 1)
    [endpt,seg_end_point,c_2,c_1,c_0] = ...
    multipleQuadApprox(x_pts,func,epsilon);
  end
  if (vari_or_const == 2) && (err_or_segs == 1)
    [endpt,seg_end_point,c_2,c_1,c_0] = ...
    constantQuadApprox(x_pts,vecFunc,consegs);
  end
  if ((vari_or_const == 2) && (err_or_segs == 2)) || (vari_or_const == 4)
    [endpt,seg_end_point,c_2,c_1,c_0] = ...
    constQuadAppxWErr(x_pts,func,epsilon);
  end
  fprintf('
********************************************************
')
  fprintf('

Back from all the Segmentation

')
  fprintf('
********************************************************
')
  fprintf('%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Compute and plot function, approximate function and error %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% for i = 1:length(seg_end_point)-1;

% looking at each segment find the approximate and actual points
XP = x_pts(seg_end_point(i):seg_end_point(i+1));
c = [c_2(i),c_1(i),c_0(i)];
FNC = vecFunc(XP);
FP = polyval(c,XP);
Error = FP - FNC;
MaxError(i) = max(abs(Error));
if (mod(i,100)==0) % Only used when trying to limit graphing
  if (mod(i,2) == 0) % Plot every other segment a different color
    figure(mod(vari_or_const,2)+1) % Blue
    plot(XP,FP)
    figure(mod(vari_or_const,2)+3) % Blue
    plot(XP,Error)
else
    figure(mod(vari_or_const,2)+1)
    plot(XP,FP,'r','LineWidth',2)  % Red
    figure(mod(vari_or_const,2)+3)
    plot(XP,Error,'r','LineWidth',2)  % Red
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)=',f,...
        ' segmentation. No. of segments = ',...
        num2str(length(seg_end_point)-1),'.'],'FontSize',10)
elseif (mod(vari_or_const,2) == 0)
    title(['UNIFORM f(x)=',f,...
        ' segmentation. No. of segments = ',...
        num2str(length(seg_end_point)-1),'.'],'FontSize',10)
end
figure(mod(vari_or_const,2)+3)
hold on
xlabel('x','FontSize',14)
errPwr2 = log2(max(MaxError));
ylabel(['Max Error = ',num2str(max(MaxError)),' = 2^',...
    num2str(errPwr2),'.'],'FontSize',10)
if (mod(vari_or_const,2) == 1)
    title(['Error for NON-UNIFORM f(x)=',f,...
        ' segmentation. No. of segs = ',...
        num2str(length(seg_end_point)-1),'.'],'FontSize',10)
elseif (mod(vari_or_const,2) == 0)
    title(['Error for UNIFORM f(x)=',f,...
        ' segmentation. No. of segs = ',...
        num2str(length(seg_end_point)-1),'.'],'FontSize',10)
end
end % if (mod(i,100)==0) Graphing STOP/START
end %for i = 1:length(seg_endpt)
figure(mod(vari_or_const,2)+1)
plot(x_pts,y_actual)  % Plot func on same fig as piecewise approx
stem(x_pts(seg_end_point),y_actual(seg_end_point))
hold off

%Decimal to Binary Conversion Algorithm

%%%%%%%%%%%%%%%% Decimal to Binary Conversion Algorithm

% Print whether Uniform or Non-uniform %
if (mod(vari_or_const,2) == 1)
fprintf('\n NON-UNIFORM Segmentation')
elseif (mod(vari_or_const,2) == 0)
fprintf('\n UNIFORM Segmentation')

96
% Convert to Twos Complement (32.32) and save in a file.
fractLen = 32; % 32 bits to represent the fraction
intLen = 64-fractLen; % 32 bits to represent the integer

% Convert to Twos Complement (16.16) and save in a file.
fractLen = 16; % 16 bits to represent the fraction
intLen = 32 - fractLen; % 16 bits to represent the integer

% BINARY FILE
% Create text file of Binary values to initialize memory
fid = fopen('memBIN.mem','w');
fprintf(fid,'%d', length(seg_end_point)-1); % Number of Segments

% Convert the values to binary and save in the file
for i = 1:length(seg_end_point)-1
    xbin(i,:) = twosComp(x_pts(seg_end_point(i+1)),intLen, fractLen);
    segmnt(i) = x_pts(seg_end_point(i+1)); % Used in next program
    c_2bin(i,:) = twosComp(c_2(i),intLen, fractLen);
    c_1bin(i,:) = twosComp(c_1(i),intLen, fractLen);
    c_0bin(i,:) = twosComp(c_0(i),intLen, fractLen);
    memBin = [xbin(i,:), c_2bin(i,:), c_1bin(i,:), c_0bin(i,:)];
    fprintf (fid,'
%s',memBin);
end
fclose (fid);

% HEXADECIMAL FILE
% Create text file of Binary values to initialize memory
fid = fopen('memHEX0x.mem','w');
Num_of_Segments = length(seg_end_point)-1;
fprintf(fid,'%6d', Num_of_Segments); % Number of Segments

% for uniform segmentation, store a step size
if (vari_or_const == 2) || (vari_or_const == 4) %
    step_len = Num_of_Segments/(intv(2) - intv(1)); %
    fprintf(fid,'%s', twosComp(step_len,intLen, fractLen));
end
% Convert the values to binary and save in the file
for i = 1:length(seg_end_point)-1
    xbin(i,:) = twosComp(x_pts(seg_end_point(i+1)),intLen, fractLen);
    segmnt(i) = x_pts(seg_end_point(i+1));    % Used in next program
    c_2bin(i,:) = twosComp(c_2(i),intLen, fractLen);
    c_lbin(i,:) = twosComp(c_l(i),intLen, fractLen);
    c_0bin(i,:) = twosComp(c_0(i),intLen, fractLen);
    memBin = [['0x',xbin(i,:)],' ',
               ['0x',c_2bin(i,:)]', ' ',
               ['0x',c_lbin(i,:)]', ' ',
               ['0x',c_0bin(i,:)]]
    fprintf (fid,'
%s',memBin);
end %for i = 1:length(seg_end_point)
fclose (fid);

%======================================%
% DECIMAL FILE                         %
%======================================%
% Create text file of Decimal Values to initialize memory
fid = fopen('memDEC.mem','w');
fprintf(fid,'%6d', Num_of_Segments);       % Number of Segments
% for uniform segmentation, store a step size
if (vari_or_const == 2) || (vari_or_const == 4)
    step_len = Num_of_Segments/(intv(2) - intv(1));  %
    fprintf(fid,'
%26.18f', step_len);    % Step size in Decimal
end
memDEC = [segmnt(1:end); c_2; c_l; c_0]
maxCoef = max(memDEC);
minCoef = min(memDEC);
fprintf(fid,'
%26.18f %26.18f %26.18f %26.18f',memDEC);
fclose (fid);
%End text file creation

fprintf('
')
fprintf('********************************************************
')
if vari_or Const ~= 3
    repeat = 0;
end
if vari_or Const == 3
    vari_or Const = 4;
end
% % profile viewer
% pr = profile('info');
% profsave(pr,'profile_results')
end % End while repeat == 1
% % maxCoef = max(maxCoef)        % for debugging to find number range
% % minCoef = min(minCoef)
% End file: QuadAppxRemz.m
A.2.1 Remez Algorithm With Chebyshev Initial Points

FILE: chebyRemz.m

function [poly_coeff, oscil, snd_Err] = chebyRemz(fun,interval,order)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% chebyRemz.m
% Get chebyshev polynomial on the first iteration. Repeat for Remez
% application; User specifies the function to approximate.
% This program turns the function provided into an inline function.

% INPUT:
% f: function entered by user (want to approximate this)
% However this function cannot be a constant. f must
% be only one variable. Must use the variable 'x'.
% order: order of approximation, e.g. 2nd order polynomial
% interval: range on which to get the coefficients will be
% approximated on the users function.

% OUTPUT:
% errRemz: error points for the range given
% poly_coeff: These are the coefficients of the polynomial that
% approximates the function.
% oscil: Oscillations on interval, for second order poly, we
% want only 2 oscillations. In this case oscillations
% are the zeroes of the first derivative.

% Author: Njuguna Macaria
% Created: 20 February 2007 Last Modified: 26 March 2007
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%=====================================%
a = interval(1);
b = interval(2);
N = 500; % Number of elements per segment
x_pts = linspace(a,b,N); % x axis sample points
y_act = fun(x_pts); % Evaluate actual function
eps = (-1).^[0:order+1]; % Epsilon for coefficients calculation
p_track = []; % For tracking result with error
%=====================================%
% Estimate with Polyfit and get data
% % % pp       = polyfit(x_pts,y_act,order); % get polyfit coefficients
% % % y_pfit   = polyval(pp,x_pts); % evaluate with polyfit coefficients
% % % errPfit  = y_pfit - y_act; % get polyfit error values to compare
% % % Repeat Powers of the polynomial in
% % in (order +2) rows and get the

99
% initial x points
%======================================%
set = ones(order+2,1)*([0:order+1]);
x_i = (a+b)/2 + (b-a)/2*cos((set*pi)/(order+1));

% Entering conditions for the loop. First loop is the chebyshev polynomial
j = 1;
max_loops = 10; % Max loops for Remez function
% % % % % ratio_error = 2;
%=========================================================================%
% Remez loop, however first set of coefficients are chebyshev coefficients%
% Exit on these conditions: 1) Convergence 2) Greater than 9 iterations %
% 3) If we have an exact quadratic to approx..
%=========================================================================%
%% while (ratio_error > 1.00000001 || ratio_error < 0.9999999) && j<max_loops
while j<max_loops
    % Extract set of initial points for evaluation (we'll use 4th column)
    % Next, evaluate the points on the actual function
    N_p = [x_i(1,1); x_i(1,2); x_i(1,3); x_i(1,4)];
    F' = fun(N_p);

    % Raise x0, x1, x2, x3, to the respective powers
    A = (x_i').^(set);
    A(:,4) = eps';
    %======================================%
    % Find Polynomial Coefficients         %
    %======================================%
p = A\F; % 1st time = chebyshev coefficients
    p_track = [p_track,p]; % Records error
    %======================================%
    % Remove err term; flip coefficients   %
    %======================================%
pflip = fliplr(p(1:end-1)');
poly_coeff = pflip;
    %======================================%
    % Calculate Plot Values                %
    %======================================%
y_apprx = polyval(pflip,x_pts); % evaluate with poly coefficients
    %======================================%
    % Calculate the Errors, break loop if %
    % 1. function is already a Quadratic    %
    % 2. If convergence has been reached    %
    %======================================%
    errRemz = y_apprx - y_act;
    max_Err = max (errRemz(2:end-1)); % Max error (exclude ends)
    min_Err = min (errRemz(2:end-1)); % Min error (exclude ends)
    if abs(max_Err)>abs(min_Err) % Set the return value of error
        snd_Err = abs(max_Err);
    else
        snd_Err = abs(min_Err);
    end
end
% (3) Exit loop if function == quadratic (very very small error)
if abs(max_Err) < 2^-40 && abs(min_Err) < 2^-40
    oscil = 0;
% % % % %         plot_cheby(x_pts,y_apprx,y_act,y_pfit,errRemz,errPfit);
    break;                          % if exact polynomial is found!!!
end

% (1) Exit loop on convergence (previous error equal to present)
if j>4
    compl=p_track(4,j);
    comp2=p_track(4,j-1);
    if compl == comp2
% %         plot_cheby(x_pts,y_apprx,y_act,y_pfit,errRemz,errPfit);
        break;
    end % if compl == comp2
end % if j>1

%%%%%%%%%%%%%%%%%%%%%%
% Finding zeroes (Max & Min of error)  %
%%%%%%%%%%%%%%%%%%%%%%
err_der  = diff(errRemz);           % Find difference between adjacent
err_sign = sign(err_der);           % points and determine the signs.
err_sign = diff(err_sign);          % Find difference between signs
errZer1  = find(err_sign == -2);    % Yields either 2 or -2 where the
errZer2  = find(err_sign ==  2);    % original function changed sign
errZeros = [errZer1,errZer2];       % Matrix of where sign changed

%%%%%%%%%%%%%%%%%%%%%%
% Exit Remez if too many Oscillations  %
% Provide Chebyshev Coefficients.      %
%%%%%%%%%%%%%%%%%%%%%%
oscil = length (errZeros);
if oscil>order
    fprintf('.

% %         warning(''Too many oscillations; Chebyshev Coefficients provided.'
% %         break;
end

%%%%%%%%%%%%%%%%%%%%%%
% Use max errors and replace x values  %
%%%%%%%%%%%%%%%%%%%%%%
new_x2  = find(errRemz == max_Err); % Index of max error point
new_x3  = find(errRemz == min_Err); % Index of min error point
% Make sure to replace into the correct order on the range
new_x2  = new_x2(1);                % Incase there are multiple
new_x3  = new_x3(1);                % pick the first element
if new_x2 > new_x3
    xi(:,2) = a+new_x2/N*(b-a);
    xi(:,3) = a+new_x3/N*(b-a);
elseif new_x2 < new_x3
    xi(:,2) = a+new_x3/N*(b-a);
    xi(:,3) = a+new_x2/N*(b-a);
end % end if new_x2 > new_x3 statement
\[ \text{ratio\_error} = \frac{\text{abs(max\_Err)}}{\text{abs(min\_Err)}}; \]
\[ \text{ratio\_err\_track} = [\text{ratio\_err\_track}, \text{ratio\_error}]; \]

\[ \text{======================================} \%
\text{ Plot actual vs the approx functions } \%
\text{======================================} \%
\]

\[ \% \text{if mod(j,3)==1 || j==max\_loops} \%
\% \text{plot\_cheby(x\_pts,y\_apprx,y\_act,y\_pfit,\_Remz,\_Pfit);} \%
\% \text{figure} \%
\% \text{plot(x\_pts,\_FuncP)} \%
\% \text{end % end if mod(j,3)==1 || j==max\_loops statement} \%
\% \% \% \text{trackj} = [\text{trackj}, j]; \%
\% \% \% \text{j=j+1; \%
\% \% \% \text{end %while loop} \%
\]

\[ \% \text{format long;} \%
\% \% \text{ratio\_err\_track} \%
\% \% \text{p\_track} \%
\% \% \text{trackj} \%
\% \% \text{format short;} \]
A.2.1 Variable Length Approximation Speed-Up Algorithms

The following files are the programs used to speed up the segmentation. 6 are presented here. The first file is the file that is used for segmentation. The others are available for the purpose of comparison. Only the first file is complete, the other files only show the code that is different from the first one i.e. the middle of the file that searches out the width for segmentation.

a. Hybrid of 3 estimates, average and thirds

FILE: varQuadApproxHyb3AvgThird.m

```matlab
function [endpt,i,p,data_] = ...
    varQuadApproxHyb3AvgThird(x_pts,f3der,est_max_len,fct,epsilon,indx)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                         
% varQuadApproxHyb3AvgThird.m                                             
%                                                                         
% This function creates a 2nd Order polynomial approximation of a given   
% function using the Remez algorithm. It continues to calculate Remez     
% approximations until epsilon is exceeded.                               
%                                                                         
% Remez approximations (with first approximation being a chebychev       
% polynomial approximation).                                             
%                                                                         
% To reduce the loop time, we first approximate the length of the          
% proposed segment. We take 3 estimates, at the beginning, end and        
% middle. Take the average of these 3. Then evaluate all the points       
% on the proposed length and get set of estimated lengths.               
% Take the average of all these estimates. This is the proposed length     
% to be used.                                                           
%                                                                         
% INPUT:                                                                
%    fct: function entered by user (want to approximate this)            
%           However this function cannot be a constant. f must be only one 
%           variable. Must use the variable 'x'.                         
%    x_pts: All the x-axis points on which to evaluate the                
%           function.                                                   
%    indx: index at which to start the interval of x values              
%    epsilon: maximum error that the user wants to limit the             
%           approximated function.                                      
%                                                                         
% OUTPUT:                                                               
%    endpt: end point of the segment                                      
%    i: Index at which we stopped the function approximated             
%    p: coefficient for polynomial approximation                         
%           p(1) is the x^2 coeff, p(2) is the x coeff and                
%           p(3) is the constant term in the 2nd order poly              
%                                                                         
% Modified by Njuguna Macaria                                           
```
syms x
order = 2; % Set the order of the polynomial
errStop = 0; % To to see if we exceeded epsilon
loopt = 1; % track times Remez is called
data_ = []; % Final loop count accumulated
x_ptsRange = x_pts(end)-x_pts(1); % Basically (b-a)
start_interval = x_pts(indx); % Start of this segment interval

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%       ESITMATION      %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%     Using Average after 3 Est    %%%%%%%%%%%%%%%%%%%%%%%%%%
abs_f3der = abs(f3der(start_interval));
if abs_f3der == 0
    len = round(.086*length(x_pts)); % Close, but ends up being increased
else
    x_range1 = 4*(epsilon*3/abs_f3der)^(1/3);
    len1 = round(x_range1/(x_ptsRange)*length(x_pts));
    if len1+indx > length(x_pts)
        len = length(x_pts) - indx;
    end
    abs_f3der = abs(f3der(x_pts(indx+len1)));
    if abs_f3der == 0
        len = round(.086*length(x_pts));
    end
else
    x_range2 = 4*(epsilon*3/abs_f3der)^(1/3);
    len2 = round(x_range2/(x_ptsRange)*length(x_pts));
    len_mid = round((len1+len2)/4);
    abs_f3der = abs(f3der(x_pts(indx+len_mid)));
    if abs_f3der == 0
        len = round(.086*length(x_pts));
    end
else
    x_range3 = 4*(epsilon*3/abs_f3der)^(1/3);
    len3 = round(x_range3/(x_ptsRange)*length(x_pts));
    len = round((len1+len2+len3)/3);
end
if len+indx > length(x_pts)
    len = length(x_pts) - indx;
end
Der3Intr = f3der(x_pts(index:indx+len)); % Get third derivatives
AV3DER = mean(Der3Intr); % Average them all
x_range = 4*(epsilon*3/abs(AV3DER))^(1/3); % Get new X_range value
len = round(x_range/(x_ptsRange)*length(x_pts)); % Best len
if len+indx > length(x_pts)
    len = length(x_pts) - indx;
elseif len > est_max_len*10 % When 3rd Derivative is small
    len = est_max_len;
end
interval = [start_interval,x_pts(len+indx)];
[p,oscil,errP] = chebyRemz(fct,interval,order);
max_Perr = errP;

LOOK = max_Perr/epsilon;
if abs_f3der == 0  ||  LOOK < 0.9 ||  LOOK > 1.002

%=====================================%
% Find a good place to start indexing %
%=====================================%
if abs_f3der == 0
while (max_Perr > epsilon) && len > 2
    len = ceil (len/3);
    if len+indx > length(x_pts)
        len = length(x_pts) - indx;
        break;
    end
    interval = [start_interval,x_pts(indx+len)];
    [p,oscil,errP] = chebyRemz(fct,interval,order);
    max_Perr = errP;
    loopt = loopt +1;
end % while max_Perr > epsilon
incrementLen = len;
else
    incrementLen = ceil(len*.05);
end % if abs_f3der == 0

while incrementLen > 2
    incrementLen = ceil(incrementLen/3);
    while (max_Perr < epsilon) && len > 2
        len = len + incrementLen;
        if len+indx > length(x_pts)
            len = length(x_pts) - indx;
            break;
        end
        interval = [start_interval,x_pts(indx+len)];
        [p,oscil,errP] = chebyRemz(fct,interval,order);
        max_Perr = errP;
        loopt = loopt +1;
    end % while max_Perr > epsilon
    incrementLen = ceil(incrementLen/3);
end % while incrementLen > 2

while (max_Perr > epsilon) && len > 2
    len = len - incrementLen;
    interval = [start_interval,x_pts(indx+len)];
    [p,oscil,errP] = chebyRemz(fct,interval,order);
    max_Perr = errP;
    loopt = loopt +1;
    if incrementLen < 2
        break;
    end
end % max_Perr > epsilon
end % end while incrementLen > 2
end % if

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%       PINPOINT        %%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%=====================================%
% Step from indx + len                %
%=====================================%
if max_Perr > epsilon               % Since we exceeded, go backwards
    i       = indx+len;             % Jump to the estimated length
    errStop = 2*epsilon;            % Increase to prevent premature stop
while i < length(x_pts)
    if errStop < epsilon
        i     = i+1;            % This was the point evaluated before
        endpt = x_pts(i);       % the decrement at the end of this
                                 % while loop. Restore index i and all
                                 % associated data.
    fid   = fopen('CompareLoop.txt','a');
data_ = [data_ loopt];
Der3Intr = f3der(x_pts(indx:indx+len));
AV3DER   = mean(Der3Intr);
    fprintf(fid,'
%4d   %4d     len: %5d    i: %5d   ',...
        'avg:%10.5f  LOOK: %8.6f  MORE',...
        i,loopt, len, i-indx, AV3DER, LOOK);
    fclose (fid);
    return
end
    loopt          = loopt + 1;
    interval       = [start_interval, x_pts(i)];
    [p,oscil,errP] = chebyRemz(fct,interval,order);
    errStop        = errP;
    i              = i -1;
end
else
    for i=indx+len:length(x_pts)    % Since we were short, go forward
        % First time thru, skip this if statement
        % If exceeded the max error, then go back to the previous endpoint
        if errStop > epsilon
            i     = i-2; % Get back to within Error
            endpt = x_pts(i);
            interval = [start_interval, x_pts(i)];
            [p,oscil,errP] = chebyRemz(fct,interval,order);
            fid   = fopen('CompareLoop.txt','a');
data_ = [data_ loopt];
Der3Intr = f3der(x_pts(indx:indx+len));
AV3DER   = mean(Der3Intr);
            fprintf(fid,'
%4d   %4d     len: %5d    i: %5d   ',...
                'avg:%10.5f  LOOK: %8.6f  LESS',...
                i,loopt, len, i-indx, AV3DER, LOOK);
fclose (fid);
return
end % if error > max
loopt = loopt + 1;
interval = [start_interval, x_pts(i)];
[p,oscil,errP] = chebyRemz(fct,interval,order);
errStop = errP*1.05; % reduces the iterations
end
end % max_Perr > epsilon...... % for i=indx+1:length(fct)

fid = fopen('CompareLoop.txt','a');
data_ = [data_ loopt];
fprintf(fid,'\n%4d   %4d',i, loopt);
close (fid);
endpt = x_pts(i);
% END OF FILE: varQuadApproxHyb3AvgThird.m
FILE: varQuadApproxBinSearch.m

while (max_Perr > epsilon) && len > 2
    len = round (len/2);
    interval = [start_interval,x_pts(indx+len)];
    [p,oscil,errP] = chebyRemz(fct,interval,order);
    max_Perr = errP;
    loopt = loopt +1;
end % while max_Perr > epsilon

incrementLen = len;

while incrementLen > 2
    incrementLen = round(incrementLen/2);
    while (max_Perr < epsilon) && len > 1
        len = len + incrementLen;
        if len+indx > length(x_pts)
            len = length(x_pts) - indx;
            break;
        end
        interval = [start_interval,x_pts(indx+len)];
        [p,oscil,errP] = chebyRemz(fct,interval,order);
        max_Perr = errP;
        loopt = loopt +1;
    end % while max_Perr > epsilon

    incrementLen = round(incrementLen/2);

    while (max_Perr > epsilon) && len > 1
        len = len - incrementLen;
        interval = [start_interval,x_pts(indx+len)];
        [p,oscil,errP] = chebyRemz(fct,interval,order);
        max_Perr = errP;
        loopt = loopt +1;
        if incrementLen == 1
            break;
        end
    end % max_Perr > epsilon
end % end while incrementLen > 2

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%% PINPOINT %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
c. Thirds

```matlab
while (max_Perr > epsilon) && len > 2
    len = round (len/3);
    if len+indx > length(x_pts)
        len = length(x_pts) - indx;
        break;
    end
    interval = [start_interval,x_pts(index+len)];
    [p,oscil,errP] = chebyRemez(fct,interval,order);
    max_Perr = errP;
    loopt = loopt +1;
end % while max_Perr > epsilon

incrementLen = len;

while incrementLen > 2
    incrementLen = round(incrementLen/3);
    while (max_Perr < epsilon) && len > 2
        len = len + incrementLen;
        if len+indx > length(x_pts)
            len = length(x_pts) - indx;
            break;
        end
        interval = [start_interval,x_pts(index+len)];
        [p,oscil,errP] = chebyRemez(fct,interval,order);
        max_Perr = errP;
        loopt = loopt +1;
    end % while max_Perr > epsilon

    incrementLen = round(incrementLen/3);

    while (max_Perr > epsilon) && len > 2
        len = len - incrementLen;
        interval = [start_interval,x_pts(index+len)];
        [p,oscil,errP] = chebyRemez(fct,interval,order);
        max_Perr = errP;
        loopt = loopt +1;
        if incrementLen < 3
            break;
        end
    end % max_Perr > epsilon
end % end while incrementLen > 2
```
d. **Ratios**

FILE: varQuadApproxRatio.m

```matlab
len = length(x_pts) - indx;

max_Perr = 100;
LOOK = 0;

%=====================================%
% Find a good place to start indexing %
%=====================================%
while (max_Perr > epsilon) && len > 2
    len = floor(len/3);
    interval = [start_interval, x_pts(indx+len)];
    [p, oscil, errP] = chebyRemz(fct, interval, order);
    max_Perr = errP;
    loopt = loopt + 1;
end

while (max_Perr < epsilon) && len > 2
    len = ceil (len*1.2);
    if len+indx > length(x_pts)
        len = length(x_pts) - indx;
        break;
    end
    interval = [start_interval, x_pts(indx+len)];
    [p, oscil, errP] = chebyRemz(fct, interval, order);
    max_Perr = errP;
    loopt = loopt + 1;
end % max_Perr > epsilon

while (max_Perr > epsilon) && len > 2
    len = floor(len*.95);
    interval = [start_interval, x_pts(indx+len)];
    [p, oscil, errP] = chebyRemz(fct, interval, order);
    max_Perr = errP;
    loopt = loopt + 1;
end % max_Perr > epsilon

while (max_Perr < epsilon) && len > 2
    len = ceil (len*1.01);
    if len+indx > length(x_pts)
        len = length(x_pts) - indx;
        break;
    end
    interval = [start_interval, x_pts(indx+len)];
    [p, oscil, errP] = chebyRemz(fct, interval, order);
    max_Perr = errP;
    loopt = loopt + 1;
end % while max_Perr > epsilon

while (max_Perr > epsilon) && len > 2
    len = floor (len*.999);
```

110
if len+indx > length(x_pts)
    len = length(x_pts) - indx;
    break;
end
interval = [start_interval, x_pts(indx+len)];
[p, oscil, errP] = chebyRemz(fct, interval, order);
max_Perr = errP;
loopt = loopt + 1;
end % while max_Perr > epsilon
% end % if
interval = [start_interval, x_pts(len+indx)];
[p, oscil, errP] = chebyRemz(fct, interval, order);
max_Perr = errP;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%       PINPOINT        %%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

e. 1 estimate

FILE: varQuadApprox1.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%       ESITMATION      %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%..     Using 1 Est     %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
abs_f3der = abs(f3der(start_interval));
if abs_f3der == 0
    len = round(.086*length(x_pts));  % Close, but ends up being increased
else
    x_range1 = 4*(epsilon*3/abs_f3der)^(1/3);
    len = round(x_range1/(x_ptsRange)*length(x_pts));
    if len+indx > length(x_pts)
        len = length(x_pts) - indx;
    end
end
interval = [start_interval, x_pts(len+indx)];
[p, oscil, errP] = chebyRemz(fct, interval, order);
max_Perr = errP;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%       PINPOINT        %%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
abs_f3der = abs(f3der(start_interval));
if abs_f3der == 0
    len = round(.086*length(x_pts));  % Close, but ends up being increased
else
    x_range1 = 4*(epsilon*3/abs_f3der)^(1/3);
    len1 = round(x_range1/(x_ptsRange)*length(x_pts));
    if len1+indx > length(x_pts)
        len = length(x_pts) - indx;
    else
        abs_f3der = abs(f3der(x_pts(indx+len1)));
        if abs_f3der == 0
            len = est_max_len;
        else
            x_range2 = 4*(epsilon*3/abs_f3der)^(1/3);
            len2 = round(x_range2/(x_ptsRange)*length(x_pts));
            len = round((len1+len2)/2);
        end
    end
    if len+indx > length(x_pts)
        len = length(x_pts) - indx;
    end
end
interval = [start_interval,x_pts(len+indx)];
[p,oscil,errP] = chebyRemz(fct,interval,order);
max_Perr = errP;

 FILE: varQuadApprox2.m

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%      ESTIMATION      %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%..     Using 2 Est     %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
FILE: varQuadApprox3.m

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%       ESTIMATION      %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%..     Using 3 Est     %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

abs_f3der = abs(f3der(start_interval));
if abs_f3der == 0
    len = round(.086*length(x_pts));  % Close, but ends up being increased
else
    x_range1 = 4*(epsilon*3/abs_f3der)^(1/3);
    len1    = round(x_range1/(x_ptsRange)*length(x_pts));
    if len1+indx > length(x_pts)
        len = length(x_pts) - indx;
    else
        abs_f3der= abs(f3der(x_pts(indx+len1)));
        if abs_f3der == 0
            len = est_max_len;
        else
            x_range2 = 4*(epsilon*3/abs_f3der)^(1/3);
            len2    = round(x_range2/(x_ptsRange)*length(x_pts));
            len_mid = round((len1+len2)/4);
            abs_f3der= abs(f3der(x_pts(indx+len_mid)));
            if abs_f3der == 0
                len = est_max_len;
            else
                x_range3 = 4*(epsilon*3/abs_f3der)^(1/3);
                len3    = round(x_range3/(x_ptsRange)*length(x_pts));
                len     = round((len1+len2+len3)/3);
            end
        end
    end
    if len+indx > length(x_pts)
        len = length(x_pts) - indx;
    elseif len > est_max_len*10     % When 3rd Derivative is small
        len = est_max_len;
    end
end

interval       = [start_interval,x_pts(len+indx)];
[p,oscil,errP] = chebyRemz(fct,interval,order);
max_Perr       = errP;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%       PINPOINT        %%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
A.2.2 Non-Uniform Quadratic Approximation

This is the file that keeps track of the segments computed and the associated endpoints and coefficients. The data is sent back to the main function, QuadAppxRemz.m. From we call varQuadApproxHyb3AvgThird.m or any of the other varQuadApprox* files depending on which one we want to use.

FILE: multipleQuadApprox.m

function [endpt,indx,c2,c1,c0] = multipleQuadApprox(xpts,fct,epsilon)

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This function will produce multiple Quadratic-line approximations of a %
% given function to within the bounds of max error provided.              
% Created: January, 2007                                                  
%                                                                         
%                                                                         
%   INPUT:                                                                
%                fct: function entered by user (want to approximate this) %
%                However this function cannot be a constant. f must %
%                be only one variable. Must use the variable 'x'.       %
%                xpts: All the x-axis points on which to evaluate the %
%                function.                                              %
%                epsilon: maximum error that the user wants to limit the %
%                approximated function.                                %
%   OUTPUT:                                                               
%              endpt: end point of the segment                            
%               indx: Array of all the index endpoints                    
%               c2: Array of the x^2 polynomial coefficients            
%               c1: Array of the x polynomial coefficients              
%               c0: Array of the constant terms in the 2nd order poly    
%                                                                         
% Modified: July 2, 2007                                                  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

syms x
format compact
i = 1;
seg_no = 1;
endpt = [ ];
c2 = [ ];
c1 = [ ];
c0 = [ ];

%========================================================================%
% Find Max length Estimate. Will be %
% used if third derivative = 0, or if %
% it's really small (NOT YET IMPLEMENTED) %
%========================================================================%
fct_vec = inline(vectorize(fct));
abs_f3der = abs(diff(diff(diff(fct))));
abs_f3der_vec = inline(vectorize(abs_f3der));
f3der_pts = abs(abs_f3der_vec(xpts));
abs_f3der_max = max(f3der_pts); % Absolute Max 3rd derivative
x_ptsRange = xpts(end) - xpts(1);

xpts_min_seg = 4*(epsilon^3/abs_f3der_max)^(1/3); % smallest seg width
min_seg_len = round(xpts_min_seg/x_ptsRange*length(xpts));
xpts_avg_seg = 4*(epsilon^3*x_ptsRange/quadl(abs_f3der_vec,xpts(1),xpts(end)))^(1/3);
avg_seg_len = round(xpts_avg_seg/x_ptsRange*length(xpts));
est_max_len = 2*avg_seg_len - min_seg_len;

% If the function is sqrt(-log(x)), then make est_max_len the max size.
% est_max_len calculated is not as large as the larger segments and will
% slow down the program because of small estimates...Therefore:
if fct == sqrt(-log(x))
est_max_len = length(xpts);
end

% Sometimes the estimates are short. To prevent this from affecting the
% program... est_max_len is increased * 10
% est_max_len = 10*est_max_len;

indx(i)= 1; % To include the first element, offset length by 1
while i < length(xpts)
    [endpt(seg_no),indx(seg_no+1),polyCoeff] = ...
        varQuadApproxHyb3AvgThird (xpts,abs_f3der_vec,...
            est_max_len,fct_vec,epsilon,i);
    c2(seg_no) = polyCoeff(1);
c1(seg_no) = polyCoeff(2);
c0(seg_no) = polyCoeff(3);
i = indx(seg_no+1);
    seg_no = seg_no + 1;
end

fprintf('

******************End of Segmentation******************

');

avg_seg_len
min_seg_len
est_max_len
Seg_lengths = diff(indx)
A.2.3 Uniform Quadratic Approximation

FILE: constantQuadApprox.m

function [endpt, indx, c2, c1, c0] = constantQuadApprox(x_pts, fct, constsegs)

% This function produces multiple Quadratic approximations of a
% given function to within the bounds of the number of segments provided.
% Coefficients calculated by Remez.
%
% Created by Tom Mack for linear approximations, using polyfit
% Created: June 4, 2006
% Modified for Quadratic approximations using Remez by Njuguna Macaria
% Modified: July 11, 2006
%
% syms x
order   = 2;
indx(1) = 1;

for i = 1:constsegs
    indx(i+1) = round((length(x_pts)/constsegs)*i);  % each iteration set seg size
    if i==constsegs
        indx(i+1) = length(x_pts);
    end
    endpt(i) = x_pts(indx(i+1));
    interval = [x_pts(indx(i)), endpt(i)];
    [p, oscil, errP] = chebyRemz(fct, interval, order);
    c2(i) = p(1);
    c1(i) = p(2);
    c0(i) = p(3);
    i = i+1;
end
A.2.4 Uniform Quadratic Approximation with Constraints

FILE: constQuadAppxWErr.m

function [endpt, indx, c2, c1, c0] = constQuadAppxWErr(xpts, fct, epsilon)

% This function will produce multiple Quadratic-line approximations of a %
% constant size of a given function to within the bounds of the max error %
% provided. Coefficients & intercept calculated using Chebychev and %
% algorithm.

% INPUT:
% fct: function entered by user (want to approximate this) %
% However this function cannot be a constant. f must %
% be only one variable. Must use the variable 'x'. %
% x_pts: All the x-axis points on which to evaluate the %
%        function.
% indx: index at which to start the interval of x values %
% epsilon: maximum error that the user wants to limit the %
%          approximated function.

% OUTPUT:
% endpt: end point of the segment %
% c2: Coefficients of x^2 in quadratic polynomial %
% c1: Coefficients of x in quadratic polynomial %
% c0: Constant of quadratic polynomial %

% Compute # of seg
% Author: Njuguna Macaria                              Date: 5 July 2007 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

syms x

% Find Min length Estimate. Will be %
% the limiting length for uniform %
% implementation %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

fct_vec = inline(vectorize(fct)); % vectorize fct (for eval)
abs_f3der = abs(diff(diff(diff(fct)))); % symbolic 3rd derivative
abs_f3der_vec = inline(vectorize(abs_f3der)); % vectorize for evaluation
f3der_pts = abs(abs_f3der_vec(xpts)); % evaluate to form vector
abs_f3der_max = max(f3der_pts); % abs (Max 3rd derivative)
x_ptsRange = xpts(end) - xpts(1); % Find length of x-domain

xpts_min_seg = 4*(epsilon^3/abs_f3der_max)^0.33; % smallest domain len
est_min_seglen = floor(xpts_min_seg/x_ptsRange*length(xpts)) %in index pts

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Find where this happens in the domain %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

IndxofMax = find(f3der_pts == abs_f3der_max); % Find min len on domain
numoftimes = length(IndxofMax); % How many are there?

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Test begin, End and Middle %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
est_max_len = length(xpts); % dummy variable
if numoftimes > 1 % more than 1 Max point
    i = IndxofMax(1); % i at begin of est seg
else
    i = IndxofMax; % default
end
IndMaxtmp = i; % The new IndexofMax

if i > length(xpts) - est_min_seglen % Check if truncated
    lenBegin = est_min_seglen; % segment then fix
    lenMid = est_min_seglen; % both these estimates
else
    % Begin with the index of the highest 3rd derivative
    [endpt, indx, p] = varQuadApproxHyb3AvgThird( xpts,...
        abs_f3der_vec, ...
        est_max_len, ...
        fct_vec, ...
        epsilon,...
        i);

    lenBegin = indx - i;

    % index of the highest 3rd derivative in the middle
    i = IndMaxtmp - floor(est_min_seglen/2); % is at end of est seg
    if i < 1 % Check to make sure not indexing before begin
        i = 1; % of interval, if so start at begin of interval
    end

    [endpt, indx, p] = varQuadApproxHyb3AvgThird( xpts,...
        abs_f3der_vec, ...
        est_max_len, ...
        fct_vec, ...
        epsilon,...
        i);

    lenMid = indx - i;
end

% end with the index of the highest 3rd derivative
i = IndMaxtmp - est_min_seglen; % is at end of est seg
if i < 1 % Check to make sure not indexing before begin
    i = 1; % of interval, if so start at begin of interval
end

[endpt, indx, p] = varQuadApproxHyb3AvgThird( xpts,...
    abs_f3der_vec, ...
    est_max_len, ...
    fct_vec, ...
    epsilon,...
    i);

lenEnd = indx - i;
% Find # of required segments on domain 
%========================================%
min_seglen = min([lenBegin,lenMid,lenEnd,est_min_seglen]);
numberOfSegs = ceil(length(xpts)/(min_seglen-1));  % Go large, figure # segs
%========================================%

% Reuse the function to calculate data 
%========================================%
[endpt,indx,c2,c1,c0] = constantQuadApprox(xpts,fct_vec,numberOfSegs);
FILE: twosComp.m

function [hexX, decX, binX] = twosComp(x, intLen, mantisaLen)
function hexX = twosComp(x, intLen, mantisaLen)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% twosComp.m
%                                                                         
% This function converts any decimal number to a two's complement binary %  
% fi object.
%                                                                         
% function [hex, decX, binX] = twosComp(x, intLen, mantisaLen)            
%                                                                         
% Input:  x:  The value to be converted                                
%          intLen:  User desired length of the integer portion of        
%                       the number.  How many bits are in the integer.   
%          mantisaLen:  The length of the mantissa.  The number of bits 
%                               in the fraction section, the precision.   
%  Output:   decX:  Decimal value as fi object.  Integer and            
%               fraction as decimal representation.                   
%          binX:  Two's Complement of the input x.  With integer        
%                           portion represented with "intLen" bits and the 
%                           fraction portion represented with "mantisaLen" 
%                           bits.                                       
%          hexX:  Two's Complement of the input x. Represented          
%                               as a Hexadecimal value.                 
%                                                                         
% This function auto-aligns the decimal point.                           
%                                                                         
% Created by:  Njuguna Macaria                                            
%       Date:  10 May 2007                                               
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

totalLen = intLen + mantisaLen;  % Total bits desired to represent the nbr.
if totalLen > 128
    warning('Max Precision: 128bits. You have requested > 128 bits');
end

% fi Object: two's complement  
%============================================================================
decX = fi(x,1,totalLen,mantisaLen);  % Create fi object, display decimal
binX = bin(decX);  % Save and return a binary form
hexX = hex(decX);
deciM = dec(decX);

% Quantizer: two's complement  
%============================================================================
q = quantizer('fixed', 'nearest', 'saturate', [totalLen mantisaLen])
[a,b] = range(q)
binX = num2bin(q,x)
decX = bin2num(q,b)
A.2.6 User Interface and Function Information Files

FILE: UserInput.m

function select = UserInput()

format long g
fprintf('\n
')
fprintf('***************************************************************
')
fprintf('\\n QUADRATIC APPROXIMATION OF A FUNCTION USING CHEBYSHEV' )
fprintf(' AND REMEZ ALGORITHM' )
fprintf('\\n')
fprintf('\\n')
disp('***************************************************************')
disp('Functions to be compared')
disp('Interval')
disp(' 1.  2^x')
disp('[0,1]')
disp(' 2.  1/x')
disp('[1,2]')
disp(' 3.  sqrt(x)')
disp('[1,2]')
disp(' 4.  1/sqrt(x)')
disp('[1,2]')
disp(' 5.  log2(x)')
disp('[1,2]')
disp(' 6.  log(x) = ln(x)')
disp('[1,2]')
disp(' 7.  sin(pi*x)')
disp('[0,1/2]')
disp(' 8.  cos(pi*x)')
disp('[0,1/2]')
disp(' 9.  tan(pi*x)')
disp('[0,1/4]')
disp(' 10.  sqrt(-log(x)) = sqrt(-ln(x))')
disp('[1/512,1/4]')
disp(' 11.  tan(pi*x)^2 + 1')
disp('[0,1/4]')
disp(' 12.  -(x*log2(x) + (1-x)*log2(1-x))')
disp('[1/256,1-1/256]')
disp(' 13.  1/(1+exp(-x)) = 1/(1+e^(-x))')
disp('[0,1]')
disp(' 14.  (1/sqrt(2*pi))*exp(-x^2/2)')
disp('[0,sqrt(2)])')
disp(' 15.  sin(exp(x))')
disp('[0,2]')
end

% Get FUNCTION to be approximated (user input)
select = input('Input the Function, func[sqrt(-1*log(x))]: ');
if isempty(select)
    select = 10; % default
end
FILE: getF.m

function [func,interval,vari_or_const,err_or_segs,consegs,epsilon,N]=...
getF(fnc_choice);

syms x
interval = '[1/256, 1/4]';           %% default
err_or_segs = 0;                        %% default
consegs = 200;                        %% default
epsilon = 0.0001;                   %% default

switch fnc_choice
    case 1
        func = '2^x';
        interval = '[0,1]';
    case 2
        func = '1./x';
        interval = '[1,2]';
    case 3
        func = 'sqrt(x)';
        interval = '[1,2]';
    case 4
        func = '1/sqrt(x)';
        interval = '[1,2]';
    case 5
        func = 'log2(x)';
        interval = '[1,2]';
    case 6
        func = 'log(x)';
        interval = '[1,2]';
    case 7
        func = 'sin(pi*x)';
        interval = '[0,1/2]';
    case 8
        func = 'cos(pi*x)';
        interval = '[0,1/2]';
    case 9
        func = 'tan(pi*x)';
        interval = '[0,1/4]';
    case 10
        func = 'sqrt(-log(x))';
        interval = '[1/512,1/4]';
    case 11
        func = 'tan(pi*x).^2 + 1';
        interval = '[0,1/4]';
    case 12
        func = '-(x*log2(x) + (1-x)*log2(1-x))';
        interval = '[1/256,1-1/256]';
    case 13
        func = '1/(1+exp(-x))';
        interval = '[0,1]';
    case 14
        func = '(1/sqrt(2*pi))*exp(-x^2/2)';
        interval = '[0,sqrt(2)]';
    case 15
        func = 'sin(exp(x))';
interval = '[0,2]';
end %switch fnc_choice

% Get CONSTANT OF VARIABLE segmentation (User input)
vari_or_const = 0;
while vari_or_const ~= 1 && vari_or_const ~= 2 && vari_or_const ~= 3
    vari_or_const = input( '(1)Non-uniform (2)Uniform Segmentation [1]: ');
    if isempty(vari_or_const)
        vari_or_const = 1;              %% default Non-uniform
    end
end

% If non-uniform segmentation, then enter ERROR parameters
if vari_or_const ~= 2
    epsilon = input( 'Input the Desired Error, epsilon[2^-33]: ');
    if isempty(epsilon)
        epsilon = 2^-33;             %% default
    end
end

% If uniform segmentation, find how the user will restrict # of segments
if vari_or_const == 2
    err_or_segs = input( 'Constrain by (1)Number of Segments or (2)Error [1]: ');
    if isempty(err_or_segs)
        err_or_segs = 1;                %% default
    end
    if err_or_segs == 1
        consegs = input( 'Input the number of Desired Segments[20]: ');
        if isempty(consegs)
            consegs = 20;               %% default
        end
    end
    if err_or_segs == 2
        epsilon = input( 'Input the given error; epsilon[2^-16]: ');
        if isempty(epsilon)
            epsilon = 2^-16;            %% default
        end
    end
end

N = input( 'Input the no. of pts the fct is to be evaluated, N[1000000]: ');
if isempty(N)
    N = 1000000;                         %% default
end
APPENDIX B. HDL CODE

B.1 MULTIPLIER CODE

The VHDL code was adapted from Xilinx’s application note on pipelining a multiplier in the Virtex II family of chips[22]. The code is for 32 bit inputs and one 32 bit product with the decimal point in the middle; 16 bit integer and 16 bit fraction.

1. VHDL

```
library ieee;
use ieee.std_logic_1164.all;
Library UNISIM;
use UNISIM.vcomponents.all;

-- Entity: Description of pins (PORTS)
entity mult16_32 is
  port( au, bu: in std_logic_vector (15 downto 0);
        clk : in std_logic;
        prod: out std_logic_vector(31 downto 0));
end mult16_32;

architecture mult16_32_beh of mult16_32 is
```
component FDR
  port(
    Q : out STD_ULOGIC;
    D : in STD_ULOGIC;
    C : in STD_ULOGIC;
    R : in STD_ULOGIC);
end component;

component MULT18X18S
  port (A   : in  STD_LOGIC_VECTOR (17 downto 0);
    B   : in  STD_LOGIC_VECTOR (17 downto 0);
    C   : in  STD_ULOGIC ;
    CE  : in  STD_ULOGIC ;
    P   : out STD_LOGIC_VECTOR (35 downto 0);
    R   : in  STD_ULOGIC );
end component;

signal  a_wire, b_wire: std_logic_vector(15 downto 0);
signal          p_wire: std_logic_vector(31 downto 0);
signal         discard: std_logic_vector( 3 downto 0);

attribute RLOC : string;

attribute RLOC of REG_A0 : label is "X0Y0" ;
attribute RLOC of REG_A1 : label is "X0Y0" ;
attribute RLOC of REG_A2 : label is "X0Y1" ;
attribute RLOC of REG_A3 : label is "X0Y1" ;
attribute RLOC of REG_A4 : label is "X0Y2" ;
attribute RLOC of REG_A5 : label is "X0Y2" ;
attribute RLOC of REG_A6 : label is "X0Y3" ;
attribute RLOC of REG_A7 : label is "X0Y3" ;
attribute RLOC of REG_A8 : label is "X0Y4" ;
attribute RLOC of REG_A9 : label is "X0Y4" ;
attribute RLOC of REG_A10: label is "X0Y5" ;
attribute RLOC of REG_A11: label is "X0Y5" ;
attribute RLOC of REG_A12: label is "X0Y6" ;
attribute RLOC of REG_A13: label is "X0Y6" ;
attribute RLOC of REG_A14: label is "X0Y7" ;
attribute RLOC of REG_A15: label is "X0Y7" ;
--    attribute RLOC of REG_A16: label is "X-1Y7";
--    attribute RLOC of REG_A17: label is "X-1Y7";

attribute RLOC of REG_B0 : label is "X2Y0" ;
attribute RLOC of REG_B1 : label is "X2Y0" ;
attribute RLOC of REG_B2 : label is "X2Y1" ;
attribute RLOC of REG_B3 : label is "X2Y1" ;
attribute RLOC of REG_B4 : label is "X2Y2" ;
attribute RLOC of REG_B5 : label is "X2Y2" ;
attribute RLOC of REG_B6 : label is "X2Y3" ;
attribute RLOC of REG_B7 : label is "X2Y3" ;
attribute RLOC of REG_B8 : label is "X2Y4" ;
attribute RLOC of REG_B9 : label is "X2Y4" ;
attribute RLOC of REG_B10: label is "X2Y5" ;
attribute RLOC of REG_B11: label is "X2Y5" ;
attribute RLOC of REG_B12: label is "X2Y6" ;
attribute RLOC of REG_B13: label is "X2Y6";
attribute RLOC of REG_B14: label is "X2Y7";
attribute RLOC of REG_B15: label is "X2Y7";
-- attribute RLOC of REG_B16: label is "X-1Y6";
-- attribute RLOC of REG_B17: label is "X-1Y6";

attribute RLOC of REG_P0 : label is "X-2Y0";
attribute RLOC of REG_P1 : label is "X1Y0";
attribute RLOC of REG_P2 : label is "X1Y0";
attribute RLOC of REG_P3 : label is "X1Y1";
attribute RLOC of REG_P4 : label is "X1Y1";
attribute RLOC of REG_P5 : label is "X3Y0";
attribute RLOC of REG_P6 : label is "X3Y0";
attribute RLOC of REG_P7 : label is "X3Y1";
attribute RLOC of REG_P8 : label is "X-2Y2";
attribute RLOC of REG_P9 : label is "X1Y2";
attribute RLOC of REG_P10: label is "X1Y2";
attribute RLOC of REG_P11: label is "X1Y3";
attribute RLOC of REG_P12: label is "X1Y3";
attribute RLOC of REG_P13: label is "X3Y2";
attribute RLOC of REG_P14: label is "X3Y2";
attribute RLOC of REG_P15: label is "X3Y3";
attribute RLOC of REG_P16: label is "X-2Y4";
attribute RLOC of REG_P17: label is "X1Y4";
attribute RLOC of REG_P18: label is "X1Y4";
attribute RLOC of REG_P19: label is "X1Y5";
attribute RLOC of REG_P20: label is "X1Y5";
attribute RLOC of REG_P21: label is "X3Y4";
attribute RLOC of REG_P22: label is "X3Y4";
attribute RLOC of REG_P23: label is "X3Y5";
attribute RLOC of REG_P24: label is "X-2Y6";
attribute RLOC of REG_P25: label is "X1Y6";
attribute RLOC of REG_P26: label is "X1Y6";
attribute RLOC of REG_P27: label is "X1Y7";
attribute RLOC of REG_P28: label is "X1Y7";
attribute RLOC of REG_P29: label is "X3Y6";
attribute RLOC of REG_P30: label is "X3Y6";
attribute RLOC of REG_P31: label is "X3Y7";
-- attribute RLOC of REG_P32: label is "X3Y1";
-- attribute RLOC of REG_P33: label is "X3Y3";
-- attribute RLOC of REG_P34: label is "X3Y5";
-- attribute RLOC of REG_P35: label is "X3Y7";

attribute BEL : string;

attribute BEL of REG_A0 : label is "FFX";
attribute BEL of REG_A1 : label is "FFY";
attribute BEL of REG_A2 : label is "FFX";
attribute BEL of REG_A3 : label is "FFY";
attribute BEL of REG_A4 : label is "FFX";
attribute BEL of REG_A5 : label is "FFY";
attribute BEL of REG_A6 : label is "FFX";
attribute BEL of REG_A7 : label is "FFY";
attribute BEL of REG_A8 : label is "FFX";
attribute BEL of REG_A9 : label is "FFY";
attribute BEL of REG_A10: label is "FFX";
attribute BEL of REG_A11: label is "FFY";
attribute BEL of REG_A12: label is "FFX";
attribute BEL of REG_A13: label is "FFY";
attribute BEL of REG_A14: label is "FFX";
attribute BEL of REG_A15: label is "FFY";
--  attribute BEL of REG_A16: label is "FFX";
--  attribute BEL of REG_A17: label is "FFY";

attribute BEL of REG_B0 : label is "FFX";
attribute BEL of REG_B1 : label is "FFY";
attribute BEL of REG_B2 : label is "FFX";
attribute BEL of REG_B3 : label is "FFY";
attribute BEL of REG_B4 : label is "FFX";
attribute BEL of REG_B5 : label is "FFY";
attribute BEL of REG_B6 : label is "FFX";
attribute BEL of REG_B7 : label is "FFY";
attribute BEL of REG_B8 : label is "FFX";
attribute BEL of REG_B9 : label is "FFY";
attribute BEL of REG_B10: label is "FFX";
attribute BEL of REG_B11: label is "FFY";
attribute BEL of REG_B12: label is "FFX";
attribute BEL of REG_B13: label is "FFY";
attribute BEL of REG_B14: label is "FFX";
attribute BEL of REG_B15: label is "FFY";
--  attribute BEL of REG_B16: label is "FFX";
--  attribute BEL of REG_B17: label is "FFY";

attribute BEL of REG_P0 : label is "FFY";
attribute BEL of REG_P1 : label is "FFX";
attribute BEL of REG_P2 : label is "FFY";
attribute BEL of REG_P3 : label is "FFX";
attribute BEL of REG_P4 : label is "FFY";
attribute BEL of REG_P5 : label is "FFX";
attribute BEL of REG_P6 : label is "FFY";
attribute BEL of REG_P7 : label is "FFX";
attribute BEL of REG_P8 : label is "FFY";
attribute BEL of REG_P9 : label is "FFX";
attribute BEL of REG_P10: label is "FFY";
attribute BEL of REG_P11: label is "FFX";
attribute BEL of REG_P12: label is "FFY";
attribute BEL of REG_P13: label is "FFX";
attribute BEL of REG_P14: label is "FFY";
attribute BEL of REG_P15: label is "FFX";
attribute BEL of REG_P16: label is "FFY";
attribute BEL of REG_P17: label is "FFX";
attribute BEL of REG_P18: label is "FFY";
attribute BEL of REG_P19: label is "FFX";
attribute BEL of REG_P20: label is "FFY";
attribute BEL of REG_P21: label is "FFX";
attribute BEL of REG_P22: label is "FFY";
attribute BEL of REG_P23: label is "FFX";
attribute BEL of REG_P24: label is "FFY";
attribute BEL of REG_P25: label is "FFX";
attribute BEL of REG_P26: label is "FFY";

128
attribute BEL of REG_P27: label is "FFX";
attribute BEL of REG_P28: label is "FFY";
attribute BEL of REG_P29: label is "FFX";
attribute BEL of REG_P30: label is "FFY";
attribute BEL of REG_P31: label is "FFX";
-- attribute BEL of REG_P32: label is "FFY";
-- attribute BEL of REG_P33: label is "FFY";
-- attribute BEL of REG_P34: label is "FFY";
-- attribute BEL of REG_P35: label is "FFY";

begin

REG_A0  : FDR port map(Q => a_wire(0) , C => CLK, D => au(0) , R => '0');
REG_A1  : FDR port map(Q => a_wire(1) , C => CLK, D => au(1) , R => '0');
REG_A2  : FDR port map(Q => a_wire(2) , C => CLK, D => au(2) , R => '0');
REG_A3  : FDR port map(Q => a_wire(3) , C => CLK, D => au(3) , R => '0');
REG_A4  : FDR port map(Q => a_wire(4) , C => CLK, D => au(4) , R => '0');
REG_A5  : FDR port map(Q => a_wire(5) , C => CLK, D => au(5) , R => '0');
REG_A6  : FDR port map(Q => a_wire(6) , C => CLK, D => au(6) , R => '0');
REG_A7  : FDR port map(Q => a_wire(7) , C => CLK, D => au(7) , R => '0');
REG_A8  : FDR port map(Q => a_wire(8) , C => CLK, D => au(8) , R => '0');
REG_A9  : FDR port map(Q => a_wire(9) , C => CLK, D => au(9) , R => '0');
REG_A10 : FDR port map(Q => a_wire(10), C => CLK, D => au(10), R => '0');
REG_A11 : FDR port map(Q => a_wire(11), C => CLK, D => au(11), R => '0');
REG_A12 : FDR port map(Q => a_wire(12), C => CLK, D => au(12), R => '0');
REG_A13 : FDR port map(Q => a_wire(13), C => CLK, D => au(13), R => '0');
REG_A14 : FDR port map(Q => a_wire(14), C => CLK, D => au(14), R => '0');
REG_A15 : FDR port map(Q => a_wire(15), C => CLK, D => au(15), R => '0');
-- REG_A16 : FDR port map(Q => a_wire(16), C => CLK, D => '0', R => '0');
-- REG_A17 : FDR port map(Q => a_wire(17), C => CLK, D => '0', R => '0');

REG_B0  : FDR port map(Q => b_wire(0) , C => CLK, D => bu(0) , R => '0');
REG_B1  : FDR port map(Q => b_wire(1) , C => CLK, D => bu(1) , R => '0');
REG_B2  : FDR port map(Q => b_wire(2) , C => CLK, D => bu(2) , R => '0');
REG_B3 : FDR port map(Q => b_wire(3) , C => CLK, D => bu(3) , R => '0');
REG_B4 : FDR port map(Q => b_wire(4) , C => CLK, D => bu(4) , R => '0');
REG_B5 : FDR port map(Q => b_wire(5) , C => CLK, D => bu(5) , R => '0');
REG_B6 : FDR port map(Q => b_wire(6) , C => CLK, D => bu(6) , R => '0');
REG_B7 : FDR port map(Q => b_wire(7) , C => CLK, D => bu(7) , R => '0');
REG_B8 : FDR port map(Q => b_wire(8) , C => CLK, D => bu(8) , R => '0');
REG_B9 : FDR port map(Q => b_wire(9) , C => CLK, D => bu(9) , R => '0');
REG_B10 : FDR port map(Q => b_wire(10), C => CLK, D => bu(10), R => '0');
REG_B11 : FDR port map(Q => b_wire(11) , C => CLK, D => bu(11) , R => '0');
REG_B12 : FDR port map(Q => b_wire(12) , C => CLK, D => bu(12) , R => '0');
REG_B13 : FDR port map(Q => b_wire(13) , C => CLK, D => bu(13) , R => '0');
REG_B14 : FDR port map(Q => b_wire(14) , C => CLK, D => bu(14) , R => '0');
REG_B15 : FDR port map(Q => b_wire(15) , C => CLK, D => bu(15) , R => '0');
-- REG_B16 : FDR port map(Q => b_wire(16) , C => CLK, D => '0', R => '0');
-- REG_B17 : FDR port map(Q => b_wire(17) , C => CLK, D => '0', R => '0');
Mult1 : MULT18X18S
  port map(P(31 downto 0) => p_wire, P(35 downto 32) => discard(3 downto 0),
          A(17 downto 16) => "00", A(15 downto 0) => a_wire,
          B(17 downto 16) => "00", B(15 downto 0) => b_wire,
          C => CLK,
          CE => '1',
          R => '0');
REG_P0 : FDR port map(Q => produ(0) , C => CLK, D => p_wire(0) , R => '0');
REG_P1 : FDR port map(Q => produ(1) , C => CLK, D => p_wire(1) , R => '0');
REG_P2 : FDR port map(Q => produ(2) , C => CLK, D => p_wire(2) , R => '0');
REG_P3 : FDR port map(Q => produ(3) , C => CLK, D => p_wire(3) , R => '0');
REG_P4 : FDR port map(Q => produ(4) , C => CLK, D => p_wire(4) , R => '0');
REG_P5 : FDR port map(Q => produ(5) , C => CLK, D => p_wire(5) , R => '0');
REG_P6 : FDR port map(Q => produ(6) , C => CLK, D => p_wire(6) , R => '0');
REG_P7 : FDR port map(Q => produ(7) , C => CLK, D => p_wire(7) , R => '0');
R => '0');
  REG_P8  : FDR port map(Q => produ(8) , C => CLK, D => p_wire(8) ,
R => '0');
  REG_P9  : FDR port map(Q => produ(9) , C => CLK, D => p_wire(9) ,
R => '0');
  REG_P10 : FDR port map(Q => produ(10) , C => CLK, D => p_wire(10) ,
R => '0');
  REG_P11 : FDR port map(Q => produ(11) , C => CLK, D => p_wire(11) ,
R => '0');
  REG_P12 : FDR port map(Q => produ(12) , C => CLK, D => p_wire(12) ,
R => '0');
  REG_P13 : FDR port map(Q => produ(13) , C => CLK, D => p_wire(13) ,
R => '0');
  REG_P14 : FDR port map(Q => produ(14) , C => CLK, D => p_wire(14) ,
R => '0');
  REG_P15 : FDR port map(Q => produ(15) , C => CLK, D => p_wire(15) ,
R => '0');
  REG_P16 : FDR port map(Q => produ(16) , C => CLK, D => p_wire(16) ,
R => '0');
  REG_P17 : FDR port map(Q => produ(17) , C => CLK, D => p_wire(17) ,
R => '0');
  REG_P18 : FDR port map(Q => produ(18) , C => CLK, D => p_wire(18) ,
R => '0');
  REG_P19 : FDR port map(Q => produ(19) , C => CLK, D => p_wire(19) ,
R => '0');
  REG_P20 : FDR port map(Q => produ(20) , C => CLK, D => p_wire(20) ,
R => '0');
  REG_P21 : FDR port map(Q => produ(21) , C => CLK, D => p_wire(21) ,
R => '0');
  REG_P22 : FDR port map(Q => produ(22) , C => CLK, D => p_wire(22) ,
R => '0');
  REG_P23 : FDR port map(Q => produ(23) , C => CLK, D => p_wire(23) ,
R => '0');
  REG_P24 : FDR port map(Q => produ(24) , C => CLK, D => p_wire(24) ,
R => '0');
  REG_P25 : FDR port map(Q => produ(25) , C => CLK, D => p_wire(25) ,
R => '0');
  REG_P26 : FDR port map(Q => produ(26) , C => CLK, D => p_wire(26) ,
R => '0');
  REG_P27 : FDR port map(Q => produ(27) , C => CLK, D => p_wire(27) ,
R => '0');
  REG_P28 : FDR port map(Q => produ(28) , C => CLK, D => p_wire(28) ,
R => '0');
  REG_P29 : FDR port map(Q => produ(29) , C => CLK, D => p_wire(29) ,
R => '0');
  REG_P30 : FDR port map(Q => produ(30) , C => CLK, D => p_wire(30) ,
R => '0');
  REG_P31 : FDR port map(Q => produ(31) , C => CLK, D => p_wire(31) ,
R => '0');
--  REG_P32 : FDR port map(Q => discard( 3) , C => CLK, D =>
p_wire(32) , R => '0');
--  REG_P33 : FDR port map(Q => discard( 2) , C => CLK, D =>
p_wire(33) , R => '0');
--  REG_P34 : FDR port map(Q => discard( 1) , C => CLK, D =>
p_wire(34) , R => '0');
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

discard(0)
-- Signal to hold value (synplify pro will not work -- if the width is not matched, Xilinx will)
SIGNAL prdt1 : std_logic_vector(49 downto 0);
SIGNAL prdt2 : std_logic_vector(65 downto 0);
SIGNAL prdt3 : std_logic_vector(65 downto 0);
SIGNAL prdt4 : std_logic_vector(65 downto 0);
SIGNAL prdt5 : std_logic_vector(65 downto 0);
SIGNAL prdt6 : std_logic_vector(65 downto 0);
-- SIGNAL b : std_logic_vector(31 downto 0);

BEGIN
-- BEGIN the 32 bit Multiplier
--------------------------------------------------------
BEGIN

PROCESS(clk)
VARIABLE zer   : std_logic_vector(15 downto 0) := X"0000";
-- zeros
VARIABLE ones  : std_logic_vector(15 downto 0) := X"FFFF";
-- ones
BEGIN

IF clk'event and clk = '1' THEN

IF (a(15) = '1') THEN
    ae(15 downto 0) <= ones;
ELSE
    ae(15 downto 0) <= zer;
END IF;

IF (b(15) = '1') THEN
    be(15 downto 0) <= ones;
ELSE
    be(15 downto 0) <= zer;
END IF;

END IF;
END PROCESS;

-- Apply the Multiplies
U00 : mult16_32
PORT MAP (au (15 downto 0) => a (15 downto 0),
bu (15 downto 0) => b (15 downto 0),
clk => clk,
produ(31 downto 0) => M00 (31 downto 0));

U01 : mult16_32
PORT MAP (au (15 downto 0) => a (15 downto 0),
bu (15 downto 0) => b (31 downto 16),
clk => clk,
produ(31 downto 0) => M01 (31 downto 0));

U10 : mult16_32
PORT MAP (au (15 downto 0) => a (31 downto 16),
bu (15 downto 0) => b (15 downto 0),
U02 : mult16_32
PORT MAP (au (15 downto 0)=> a (15 downto 0),
bu (15 downto 0)=> be (15 downto 0),
clk => clk,
produ(31 downto 0)=> M02 (31 downto 0) );

U11 : mult16_32
PORT MAP (au (15 downto 0)=> a (31 downto 16),
bu (15 downto 0)=> b (31 downto 16),
clk => clk,
produ(31 downto 0)=> M11 (31 downto 0) );

U20 : mult16_32
PORT MAP (au (15 downto 0)=> ae (15 downto 0),
bu (15 downto 0)=> b (15 downto 0),
clk => clk,
produ(31 downto 0)=> M20 (31 downto 0) );

-- shift the values appropriately for addition
PROCESS(clk)
BEGIN
  IF clk'event and clk = '1' then
    A00(33 downto 32) <= "00";
    A00(31 downto 0) <= M00(31 downto 0);
    A01(49 downto 48) <= "00";
    A01(47 downto 16) <= M01(31 downto 0);
    A01(15 downto 0) <= X"0000";
    A10(49 downto 48) <= "00";
    A10(47 downto 16) <= M10(31 downto 0);
    A10(15 downto 0) <= X"0000";
    A02(65 downto 64) <= "00";
    A02(63 downto 32) <= M02(31 downto 0);
    A02(31 downto 0) <= X"00000000";
    A11(65 downto 64) <= "00";
    A11(63 downto 32) <= M11(31 downto 0);
    A11(31 downto 0) <= X"00000000";
    A20(65 downto 64) <= "00";
    A20(63 downto 32) <= M20(31 downto 0);
    A20(31 downto 0) <= X"00000000";
  END if;
END PROCESS;

PROCESS(clk)
BEGIN
  IF clk'event and clk = '1' then
prdt1 <= unsigned(A00) + unsigned(A01) + unsigned(A10);
prdt2 <= unsigned(A02) + unsigned(A11) + unsigned(A20);
prdt3 <= unsigned(prdt2) + unsigned(prdt1);
prod <= prdt3(47 downto 16);
END IF;
END PROCESS;
END structural;

2. Verilog

// $Id: S_MULT_64TO64_SRC6.v,v 1.1 2007/06/25 18:20:29 pvg Exp $

//
// Copyright 2007 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.
//
// DESCRIPTION: This module performs 64 bit signed integer
// multiplication
// and provides a 64 bit result.
// This module instantiates Xilinx components.
// ---------------------------------------------------------------
// This file was modified by Njuguna Macaria to make a 64 bit by 64 bit
// Multiplier with a 64 bit result that is shifted to the appropriate
// decimal point for a 32 bit integer and 32 bit fraction.
//
//---------------------------------------------------------------

//---------------------------------------------------------------

// `timescale 1ns/1ns

module mult32_64s (A, B, Q, CLK, CLR);
  input [31:0] A;
  input [31:0] B;
  output [63:0] Q;

input CLK /* synthesis syn_noclockbuf=1 syn_maxfan=100000 */ ;
input CLR;

reg [63:0] Q;
reg [31:0] AR;
reg [31:0] BR;
wire [35:0] R0;
wire [35:0] R1;
wire [35:0] R2;
wire [35:0] R3;

reg [31:0] R0_R;
reg [31:0] R1_R;
reg [31:0] R2_R;
reg [31:0] R3_R;

always @ (posedge CLK or posedge CLR)
begin
  if (CLR) begin
    AR <= 0;
    BR <= 0;
  end
  else begin
    AR <= A;
    BR <= B;
  end
end

MULT18X18S X0 ( .A ((2'b0, AR[15:0])),
  .B ((2'b0, BR[15:0])),
  .C (CLK),
  .R (CLR),
  .CE (1'b1),
  .P (R0) );

MULT18X18S X1 ( .A ((2'b0, AR[31:16])),
  .B ((2'b0, BR[15:0])),
  .C (CLK),
  .R (CLR),
  .CE (1'b1),
  .P (R1) );

MULT18X18S X2 ( .A ((2'b0, AR[15:0])),
  .B ((2'b0, BR[31:16])),
  .C (CLK),
  .R (CLR),
  .CE (1'b1),
  .P (R2) );
MULT18X18S X3 (  
  .A      ((2'b0, AR[31:16])),  
  .B      ((2'b0, BR[31:16])),  
  .C      (CLK),  
  .R      (CLR),  
  .CE     (1'b1),  
  .P      (R3)  
);  

always @ (posedge CLK or posedge CLR)  
begin  
  if (CLR) begin  
    R0_R <= 0;  
    R1_R <= 0;  
    R2_R <= 0;  
    R3_R <= 0;  
  end  
  else begin  
    R0_R <= R0;  
    R1_R <= R1;  
    R2_R <= R2;  
    R3_R <= R3;  
  end  
end  

always @ (posedge CLK or posedge CLR)  
begin  
  if (CLR) begin  
    Q <= 0;  
  end  
  else begin  
    // add and shift  
    Q <= R0_R + {R1_R,16'b0} + {R2_R,16'b0} + {R3_R,32'b0};  
  end  
end  
endmodule

`timescale 1ns/1ns  
module mult_64s (A, B, Q, CLK, CLR);  
  input  [63:0] A;  
  input  [63:0] B;  
  output [63:0] Q;  
  input CLK /* synthesis syn_noclockbuf=1 syn_maxfan=100000 */ ;  
  input CLR;
always @ (posedge CLK or posedge CLR)
begin
  if (CLR) begin
    AR <= 0;
    BR <= 0;
  end
  else begin
    AR <= A;
    BR <= B;
  end
end

mult32_64s X0 (  
.A (AR[31:0]),  
.B (BR[31:0]),  
.Q (R0),  
.CLK (CLK),  
.CLR (CLR)  );
mult32_64s X1 (  
.A (AR[63:32]),  
.B (BR[31:0 ]),  
.Q (R1),  
.CLK (CLK),  
.CLR (CLR)  );
mult32_64s X2 (  
.A (AR[31:0]),  
.B (BR[63:32]),  
.Q (R2),  
.CLK (CLK),  
.CLR (CLR)  );
mult32_64s X3 (  
.A (AR[63:32]),  
.B (BR[63:32]),  
.Q (R3),  
.CLK (CLK),  
.CLR (CLR)  );
```verilog
always @ (posedge CLK or posedge CLR)
begin
    if (CLR) begin
        R0_R <= 0;
        R1_R <= 0;
        R2_R <= 0;
        R3_R <= 0;
    end
    else begin
        R0_R <= R0;
        R1_R <= R1;
        R2_R <= R2;
        R3_R <= R3;
    end
end

always @ (posedge CLK or posedge CLR)
begin
    if (CLR) begin
        Q <= 0;
    end
    else begin
        // add and shift
        Q_R <= R0_R + {R1_R,32'b0} + {R2_R,32'b0} + {R3_R,64'b0};
        // Only take 64 bits from the middle for a 32.32 number
        Q   <= Q_R[95:32];
    end
end
endmodule
```
APPENDIX C. SRC C CODE

C.1 UNIFORM SEGMENTATION

1. Floating Point

   a. Main.c

```
#include<stdio.h>
#include<stdlib.h>
#include<strings.h>
#include<libmap.h>

// Subroutine initialization in Main
void  subr_map( double  acoef[],
    int     ncoef,
    double  incre,
    double  offsetV,
    double  x[],
    double  y[],
    double  ys[],
    int     npts,
    int64_t *time0,
    int64_t *time1,
    int     mapnum);

// MAIN
main ()  {

   // Initialize Variables
   FILE    *fp1;
   double  *array, *x, *y, *ys, incre,val,offsetV;
   int     i,ir,nc,npts, mapnum,nmap, ncoef,arr_indx,inNum;
   int64_t tm0, tm1;

   // Start NFG and select map number
   printf ("\n***START UP THE NFG ***\n");
   mapnum = 0;
   nmap   = 1;

   // ! allocate map to this problem
   map_allocate (nmap);

   // User interface
   printf("----------------------------------------\n");
   printf("Function  1.  2^x                           :  1\n");
   printf("Function  2.  1/x                           :  2\n");
   printf("Function  3.  sqrt(x)                       :  3\n");
   printf("Function  4.  1/sqrt(x)                     :  4\n");
   printf("Function  5.  log2(x)                       :  5\n");
   printf("Function  6.  ln(x)                         :  6\n");
   printf("Function  7.  sin(pi*x)                     :  7\n");
   printf("Function  8.  cos(pi*x)                     :  8\n");
   printf("Function  9.  tan(pi*x)                     :  9\n");
```

`printf("Function 10. sqrt(-ln(x)) : 10\n"));
printf("Function 11. tan(pi*x)^2 + 1 : 11\n"));
printf("Function 12. -(x*log2(x) + (1-x)*log2(1-x)): 12\n"));
printf("Function 13. 1/(1+e^(-x)) : 13\n"));
printf("Function 14. (1/sqrt(2*pi))*exp(-x^2/2) : 14\n"));
printf("Function 15. sin(exp(x)) : 15\n"));
printf("-----------------------------------------------------------------------------------\n"));

printf("\nSelect which function to implement: ");
scanf("%i", &inNum);
printf("What value did I enter: %i \n ",inNum);

// Open the Hex data file to read
switch (inNum) {
  case 1: fp1 = fopen("Data/memD1.mem","r");
          break;
  case 2: fp1 = fopen("Data/memD2.mem","r");
          break;
  case 3: fp1 = fopen("Data/memD3.mem","r");
          break;
  case 4: fp1 = fopen("Data/memD4.mem","r");
          break;
  case 5: fp1 = fopen("Data/memD5.mem","r");
          break;
  case 6: fp1 = fopen("Data/memD6.mem","r");
          break;
  case 7: fp1 = fopen("Data/memD7.mem","r");
          break;
  case 8: fp1 = fopen("Data/memD8.mem","r");
          break;
  case 9: fp1 = fopen("Data/memD9.mem","r");
          break;
  case 10: fp1 = fopen("Data/memD10.mem","r");
           break;
  case 11: fp1 = fopen("Data/memD11.mem","r");
           break;
  case 12: fp1 = fopen("Data/memD12.mem","r");
           break;
  case 13: fp1 = fopen("Data/memD13.mem","r");
           break;
  case 14: fp1 = fopen("Data/memD14.mem","r");
           break;
  default: fp1 = fopen("Data/memD15.mem","r");
           break;
}
printf ("fp1 %i\n",fp1);

// Read in the values from the file
fscanf (fp1, "%i", &ncoef);
fscanf (fp1, "%lf", &incre);
fscanf (fp1, "%lf", &offsetV);

// Depending on number segments
//nc = 50;        // For 16 bit accuracy
//nc = 600;       // For 23 bits
//nc = 1500;      // For 32 bits
nc = 35000;       // For 40 bits
array     = (double*)Cache_Aligned_Allocate (4*nc*8);
x         = (double*)Cache_Aligned_Allocate (nc*8  );
y         = (double*)Cache_Aligned_Allocate (nc*8  );
y         = (double*)Cache_Aligned_Allocate (nc*8  );

// check if the right thing was read
printf("  ncoef %i\n",ncoef);

// read_file
for (i=0;i<ncoef;i++)  {
    fscanf (fp1, "%lf", &val);
    array[i*4] = val;
    fscanf (fp1, "%lf", &val);
    array[i*4+1] = val;
    fscanf (fp1, "%lf", &val);
    array[i*4+2] = val;
    fscanf (fp1, "%lf", &val);
    array[i*4+3] = val;
} // end read_file
fclose(fp1);
npts = 30;
// create_samples
for (ir=0;ir<npts;ir++) {
    arr_indx = ir % ncoef;
    x[ir]    = array[arr_indx*4];
    printf("ir %3i x_values are: %lf\n",ir,x[ir]);
} //end create_samples
printf ("main ncoef %i npts %i\n",ncoef,npts);

subr_map (array, ncoef, incre, offsetV, x, y, ys, npts, &tm0, &tm1,
mapnum);

printf ("\n************ BACK FROM MAP **********\n");
printf("%lld clocks for NFG\n", tm0);
printf("%lld clocks for SRC Macro\n", tm1);

for (i=0;i<npts;i++) {
    printf ("x: %.16lf  ysubr: %.16lf  ySRCMacro: %.16lf\n", 
            x[i],y[i],ys[i] );
}

// ! release the map resources
map_free (nmap);
#include <libmap.h>

void subr_map ( double ac[],
               int ncoef,
               double incre,
               double offsetV,
               double xc[],
               double yc[],
               double ys[],
               int npts,
               int64_t *time0,
               int64_t *time1,
               int mapno)
{
    /******************************************************************
    * Declarations
    ******************************************************************/
    OBM_BANK_A (ysmap, double, MAX_OBM_SIZE)
    OBM_BANK_B (a, double, MAX_OBM_SIZE)
    OBM_BANK_C (b, double, MAX_OBM_SIZE)
    OBM_BANK_D (c, double, MAX_OBM_SIZE)
    OBM_BANK_E (x, double, MAX_OBM_SIZE)
    OBM_BANK_F (y, double, MAX_OBM_SIZE)
    int i,j, nbytes, indx;
    int64_t tm0,tm1;
    double varx,indxtmp;

    /******************************************************************
    * Read in the cooeff and segment endpoints
    ******************************************************************/
    nbytes = 4*ncoef * 8; /* 4 data values (seg,a,b,c), 64bits each */
    DMA_CPU (CM2OBM, ysmap, MAP_OBM_stripe(1,"A,B,C,D"), ac, 1, nbytes, 0);
    wait_DMA (0);

    /******************************************************************
    * Read in the Number of points
    ******************************************************************/
    nbytes = npts * 8;
    DMA_CPU (CM2OBM, x, MAP_OBM_stripe(1,"E"), xc, 1, nbytes, 0);
    wait_DMA (0);

    /******************************************************************
    * Useful in Debug Mode to determine when in Map
    ******************************************************************/
    printf ("\n\n************ NOW IN MAP **********\n");
    printf ("MAP subr ncoef %i npts %i\n",ncoef,npts);

    /******************************************************************
    * Read timer and use a constant for UNIFORM Segmentation
    ******************************************************************/
    read_timer (&tm0);
printf("incre: %15.10lf  offset: %15.10lf\n", incre, offsetV);
for (i=0; i<npts; i++)
{
    varx    = x[i];
    indxtmp = incre * varx;
    indx    = (int)(indxtmp - offsetV);  // For interval [a,b]; when
    if (a!=0)
        y[i] = a[indx] * varx * varx + varx * b[indx] + c[indx];
    // For Debug only
    printf("indxtmp: %15.10lf indx: %i x: %15.10lf a: %15.10lf ",
            indxtmp,          indx,    varx,       a[indx]);
    printf("b: %15.10lf c: %15.10lf fx: %15.10lf\n",
            b[indx],    c[indx],   y[i]);
}

read_timer (&tml);
*time0 = tml - tm0;

read_timer (&tm0);
if (ncoef == 4017){
    for (i=0; i<npts; i++)
        ysmap[i] = sqrt(-1*logf(x[i]));         // func 10
        // ysmap[i] = cosf(x[i]*3.14159265358979);    // func 8
    read_timer (&tml);
    *time1 = tml - tm0;
}

/********************************************
* Send back the results
*********************************************/
bytes = npts * 8;
DMA_CPU (OBM2CM, y, MAP_OBM_stripe(1,"F"), yc, 1, nbytes, 0);
wait_DMA (0);

bytes = npts * 8;
DMA_CPU (OBM2CM, ysmap, MAP_OBM_stripe(1,"A"), ys, 1, nbytes, 0);
wait_DMA (0);
2. Fixed Point
   
   a. Main.c

```c
#include<stdio.h>
#include<stdlib.h>
#include<strings.h>
#include<libmap.h>
#include<math.h>

// Subroutine initialization in Main
void subr_map (int64_t acoef[], int ncoef, int64_t incre, int64_t offsetV, int64_t x[], int64_t y[], int xpts, int64_t *time0, int mapnum);

// MAIN
main () {

   // Initialize Variables
   FILE *fp1;
   int i,ir,nc,xpts,inNum;
   int mapnum,nmap,ncoef;
   int arr_indx;
   int64_t *arraym,*xm,*ym,incre,offsetV;
   int64_t tm0,tml,hexval;
```
char hexstr[80], *token, *stpstr, strDelimit[]=" 
";

// Starting NFG
printf ("\n***START UP THE NFG ***\n");
mapnum = 0;
nmap = 1;

// User interface
printf("---------------------------------------------\n");
printf("Function   1.  2^x                           :  1\n");
printf("Function   2.  1/x                           :  2\n");
printf("Function   3.  sqrt(x)                       :  3\n");
printf("Function   4.  1/sqrt(x)                     :  4\n");
printf("Function   5.  log2(x)                       :  5\n");
printf("Function   6.  ln(x)                         :  6\n");
printf("Function   7.  sin(pi*x)                     :  7\n");
printf("Function   8.  cos(pi*x)                     :  8\n");
printf("Function   9.  tan(pi*x)                     :  9\n");
printf("Function  10.  sqrt(-ln(x))                  : 10\n");
printf("Function  11.  tan(pi*x)^2 + 1               : 11\n");
printf("Function  12.  -(x*log2(x) + (1-x)*log2(1-x)): 12\n");
printf("Function  13.  1/(1+e^(-x))                  : 13\n");
printf("Function  14.  (1/sqrt(2*pi))*exp(-x^2/2)    : 14\n");
printf("Function  15.  sin(exp(x))                   : 15\n");
printf("---------------------------------------------\n");

//inNum = 1;           // dummy default value
printf("\nSelect which function to implement: ");
scanf("%i", &inNum);
printf("What value did I enter: %i \n ",inNum);

// Open the Hex data file to read
switch (inNum)
{
    case 1: fp1 = fopen("Data/memH1.mem","r");
        break;
    case 2: fp1 = fopen("Data/memH2.mem","r");
        break;
    case 3: fp1 = fopen("Data/memH3.mem","r");
        break;
    case 4: fp1 = fopen("Data/memH4.mem","r");
        break;
    case 5: fp1 = fopen("Data/memH5.mem","r");
        break;
    case 6: fp1 = fopen("Data/memH6.mem","r");
        break;
    case 7: fp1 = fopen("Data/memH7.mem","r");
        break;
    case 8: fp1 = fopen("Data/memH8.mem","r");
        break;
    case 9: fp1 = fopen("Data/memH9.mem","r");
        break;
}
case 10: fp1 = fopen("Data/memH10.mem","r");
    break;
case 11: fp1 = fopen("Data/memH11.mem","r");
    break;
case 12: fp1 = fopen("Data/memH12.mem","r");
    break;
case 13: fp1 = fopen("Data/memH13.mem","r");
    break;
case 14: fp1 = fopen("Data/memH14.mem","r");
    break;
default: fp1 = fopen("Data/memH15.mem","r");
    break;
}
printf("fp1 %i\n",fp1);

// ! allocate map to this problem
map_allocate (nmap);

// Read in the number of segments (decimal #)
fscanf (fp1, "%i", &ncoef);
fscanf (fp1, "%llx", &incre);
fscanf (fp1, "%llx", &offsetV);
printf("ncoef: %3i incre: %8llx\n",ncoef,incre);

// Accommodate lots of results
nc = 30000;

// array is enough room to hold 4 64 bit data pieces
// Perform cache alignment
arraym = (int64_t *)Cache_Aligned_Allocate (4*ncoef*8);
xm      = (int64_t *)Cache_Aligned_Allocate (nc*8  );
ym      = (int64_t *)Cache_Aligned_Allocate (nc*8  );

// Get rid of first npc
fgets (hexstr, sizeof hexstr, fp1);

// Read all endpoints and coefficients into OBM banks
for (i=0;i<ncoef;i++)  {
    fgets (hexstr, sizeof hexstr, fp1);

    token         = strtok(hexstr,strDelimit);
    sscanf (token, "%llx", &hexval);
    arraym[i*4]   = hexval;

    token = strtok(NULL,strDelimit);
    sscanf (token, "%llx", &hexval);
    arraym[i*4+1] = hexval;

    token = strtok(NULL,strDelimit);
    sscanf (token, "%llx", &hexval);
    arraym[i*4+2] = hexval;

    token = strtok(NULL,strDelimit);
    sscanf (token, "%llx", &hexval);
    arraym[i*4+3] = hexval;
{149}

// close the file
fclose(fp1);

// create some values to test with
xpts = 100;
for (ir=0;ir<xpts;ir++) {
    arr_indx = ir % ncoef;
    xm[ir] = arraym[arr_indx*4];       // Optional -0x2061d;
    printf("arr_indx = %3i  xm[%2i]= %10llx
",
            arr_indx,   ir, xm[ir]);
}

printf ("Right Before MAP *** \nmain ncoef %i xpts %i\n",
            ncoef,   xpts);
subr_map (arraym,ncoef,incre,offsetV,xm,ym,xpts,&tm0,mapnum);

printf ("\n************Back from the MAP!!!************\n");
printf ("\n************    SHIFT8   ***************\n");
for(i=0;i<xpts;i++){
    printf("i: %3i x: %8llx fx: %10llx\n",i,xm[i],ym[i] );
}
printf("\n************End of the MAP!!!************\n");

	// ! release the map resources
map_free (nmap);

b. subr.mc

#include <libmap.h>

void subr_map (int64_t ac[],
        int        ncoef,
        int64_t    incre,
        int64_t    offsetV,
        int64_t    xc[],
        int64_t    yc[],
        int        xpts,
        int64_t    *time0,
        int        mapno) {

    /**************************************************************
    * Declarations
    ***************************************************************/
    OBM_BANK_A (segend, int64_t, MAX_OBM_SIZE)
    OBM_BANK_B (a,        int64_t, MAX_OBM_SIZE)
    OBM_BANK_C (b,        int64_t, MAX_OBM_SIZE)
    OBM_BANK_D (c,        int64_t, MAX_OBM_SIZE)
    OBM_BANK_E (x,        int64_t, MAX_OBM_SIZE)
    OBM_BANK_F (y,        int64_t, MAX_OBM_SIZE)
    int  i,j, nbytes;
int64_t tm0, tm1, varx, varsq, vara, varb, varc, ax2, bx1, fx;
int64_t varxtmp, indx;

/*********************************************
* Read into OBM. Coeff & segment endpoints *
*********************************************/

// 4 data values (seg,a,b,c), 64bit Hex values
nbytes = 4*ncoef * 8;
DMA_CPU(CM2OBM, segend, MAP_OBM_stripe(1,"A,B,C,D"), ac, 1, nbytes, 0);
wait_DMA (0);

// Read in the Number of points
nbytes = xpts * 8;
DMA_CPU (CM2OBM, x, MAP_OBM_stripe(1,"E"), xc, 1, nbytes, 0);
wait_DMA (0);

// DEBUG: determine when in Map
printf ("************ NOW IN MAP **********");
printf ("MAP subr ncoef %i xpts %i \n", ncoef, xpts);

/**********************************************************
* Read timer and use selector to determine the segment   *
**********************************************************/
read_timer (&tm0);

incre >>= 16;         // asr to open integer bits
offsetV >>= 16;         // asr to match in subtraction

for (i=0;i<xpts;i++)
{
    varx    = x[i];             // Take from OBM put in BRAM
    indx    = varx * incre;     // Segment index Number * x input
    indx   >>= 32;               // Return to 16 fraction points
    indx   = indx - offsetV;   // Adjust index to interval start
    indx   >>= 16;               // remove fracion
    vara    = a[(int)indx];     // Move from OBM into BRAM
    varb    = b[(int)indx];
    varc    = c[(int)indx];

    // ------ Square X and shift ----/
    varx   >>= 8;                    // Remove lower 8 bits, 40.24
    varsq   = varx*varx;            // Now we have 80.48 -> 16.48
    varsq   >>= 24;                   // SRL eliminate 40.24
    if (varx < 0x8000000000000000)  // if varx is positive
        varsq = varsq & 0x000000FFFFFFFFFFFFF; // bitwise AND; 24bits

    // --- X^2 * first Coefficient ---/
    ax2    = varsq*vara;            // a[indx];
    ax2    >>= 16;                   // Want 32.32, so srl 16
    if (vara < 0x8000000000000000)  // if both +ve
        ax2 = ax2 & 0x000000FFFFFFFFFFFFF; // bitwise AND; 16bits
// --- X * second Coefficient --//
varb >>= 8;                  // Remove lower 8 bits, 40.24
bx1 = varx*varb;            // both are already shifted
bx1 >>= 16;                  // Return to 32.32 (int.fract)
if (varb < 0x8000000000000000) // if both +ve
    bx1 = bx1 & 0x0000FFFFFFFFFFFF;     // bitwise AND; 16bits

// -- 3 input add to complete --//
y[i]  = ax2+bx1+varc;        // no need to shift varc

// DEBUG
// printf("indx: %4llx -> %4li varx: %6llx incre: %6llx\n",
//         indx,      (int)indx,varx,       incre);
}

// Time it took to compute
read_timer (&tm1);
*time0 = tm1-tm0;

/************************************************************
* Send back the results
************************************************************/
  nbytes = xpts * 8;
  DMA_CPU (OMB2CM, y, MAP_OBM_stripe(1,"F"), yc, 1, nbytes, 0);
  wait_DMA (0);
}
C.2 NON-UNIFORM SEGMENTATION

1. Floating Point

   a. Main.c

```c
#include<stdio.h>
#include<stdlib.h>
#include<strings.h>
#include<libmap.h>

// Subroutine initialization in Main
void subr_map( double acoef[],
    int ncoef,
    double x[],
    double y[],
    double ys[],
    int npts,
    int64_t *time0,
    int64_t *time1,
    int = mapnum);

// MAIN
main () {

    // Initialize Variables
    FILE *fp1;
    double *array, *x, *y, *ys;
    double val;
    int i,ir,nc,npts, mapnum,nmap, ncoef,arr_indx;
    int64_t tm0, tml;

    printf ("\n***START UP THE NFG ***\n");

    // select map number
    mapnum = 0;
    nmap   = 1;

    // ! allocate map to this problem
    map_allocate (nmap);

    // Depending on number segments
    //nc = 50;      // For 16 bit accuracy
    nc = 200;   // For 23 bits
    //nc = 1500;    // For 32 bits
    //nc = 5000;    // For 42 bits
    array     = (double*)Cache_Aligned_Allocate (4*nc*8);
    x         = (double*)Cache_Aligned_Allocate (nc*8  );
    y         = (double*)Cache_Aligned_Allocate (nc*8  );
    ys        = (double*)Cache_Aligned_Allocate (nc*8  );

    fp1 = fopen ("Data/memDEC.mem","r");
    fscanf (fp1, "%i", &ncoef);
    // check if the right thing was read
    printf ("    ncoef \%i\n",ncoef);
```
// read_file
for (i=0; i<ncoef; i++) {
    fscanf (fp1, "%lf", &val);
    array[i*4] = val;
    fscanf (fp1, "%lf", &val);
    array[i*4+1] = val;
    fscanf (fp1, "%lf", &val);
    array[i*4+2] = val;
    fscanf (fp1, "%lf", &val);
    array[i*4+3] = val;
} // end read_file

/* // print_array
for (i=0; i<ncoef; i++) {
    printf("endpt %10.6f a %10.6f b %10.6f c %10.6f\n",
            array[4*i+0],
            array[4*i+1],
            array[4*i+2],
            array[4*i+3]);
} // end print_array
*/

npts = 100;
// create_samples
for (ir=0; ir<npts; ir++) {
    arr_indx = ir % ncoef;
    x[ir] = array[arr_indx*4];
    printf("ir %3i x_values are: %lf\n",ir,x[ir]);
} // end create_samples

printf("main ncoef %i npts %i\n",ncoef,npts);

subr_map (array, ncoef, x, y, ys, npts, &tm0, &tmo, mapnum);

printf("\n************ BACK FROM MAP ************\n");
printf("%lld clocks\n",tm0);
printf("%lld clocks\n",tm1);

for (i=0; i<npts; i++) {
    printf("x: %5.18lf ysubr: %5.18lf SRCMacro2^x: %5.18f\n",
            x[i], y[i], ys[i]);
    // printf("x: %5.18f ysubr: %5.18f\n",x[i],y[i]);
}

// ! release the map resources
map_free (nmap);


b. \( \text{subr.mc} \left( \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \right) \)

```c
#include <libmap.h>

void subr_map ( double ac[],
    int ncoef, double xc[], double yc[], double ys[],
    int npts,
    int64_t *time0, int64_t *time1,
    int mapno) {

    /**************************************************************************
    * Declarations
    **************************************************************************/
    OBM_BANK_A (ysmap, double, MAX_OBM_SIZE)
    OBM_BANK_B (a, double, MAX_OBM_SIZE)
    OBM_BANK_C (b, double, MAX_OBM_SIZE)
    OBM_BANK_D (c, double, MAX_OBM_SIZE)
    OBM_BANK_E (x, double, MAX_OBM_SIZE)
    OBM_BANK_F (y, double, MAX_OBM_SIZE)
    int i,j, nbytes, indx, sel;
    int64_t tm0,tm1;
    double varx;

    /**************************************************************************
    * Read in the cooeff and segment endpoints  
    **************************************************************************/
    nbytes = 4*ncoef * 8; /* 4 data values (seg,a,b,c), 64bits each */
    DMA_CPU (CM2OBM, ysmap, MAP_OBM_stripe(1,"A,B,C,D"), ac, 1, nbytes,
        0);
    wait_DMA (0);

    /**************************************************************************
    * Read in the Number of points
    **************************************************************************/
    nbytes = npts * 8;
    DMA_CPU (CM2OBM, x, MAP_OBM_stripe(1,"E"), xc, 1, nbytes, 0);
    wait_DMA (0);

    /**************************************************************************
    * Useful in Debug Mode to determine when in Map
    **************************************************************************/
    printf ("
        ************ NOW IN MAP **********
    ");
    printf ("MAP subr ncoef %i npts %i\n",ncoef,npts);

    /**************************************************************************
    * Read timer and use a constant for UNIFORM Segmentation
    **************************************************************************/
    read_timer (&tm0);
}
for (i=0;i<npts;i++)
{
    varx = x[i];

    if ( varx <= 1.010456600772177400)
        sel = 1;
    else if ( varx <= 1.254138569173091300)
        sel = 2;
    else if ( varx <= 1.393018725189699000)
        sel = 3;
    else if ( varx <= 1.414213562373095100)
        sel = 4;
    switch (sel)
    {
    case 1:
        select_pri_64bit_32val( varx <= 0.065896761049097793, 0,
                                varx <= 0.113411555832503830, 1,
                                varx <= 0.155068672182882060, 2,
                                varx <= 0.193392483833442240, 3,
                                varx <= 0.229462794562507500, 4,
                                varx <= 0.263882719864044100, 5,
                                varx <= 0.297032283926999020, 6,
                                varx <= 0.329159950015850300, 7,
                                varx <= 0.360453699017791120, 8,
                                varx <= 0.391055896896180420, 9,
                                varx <= 0.421076852419120200, 10,
                                varx <= 0.450608489560304140, 11,
                                varx <= 0.479725761713275970, 12,
                                varx <= 0.508490894337077280, 13,
                                varx <= 0.536959041815795230, 14,
                                varx <= 0.565178287586335250, 15,
                                varx <= 0.593191057714890110, 16,
                                varx <= 0.621035536389326210, 17,
                                varx <= 0.648747078855175910, 18,
                                varx <= 0.676359626273058010, 19,
                                varx <= 0.703904291372063890, 20,
                                varx <= 0.731409358451725390, 21,
                                varx <= 0.758903111811573990, 22,
                                varx <= 0.786413835711417700, 23,
                                varx <= 0.813969814569603100, 24,
                                varx <= 0.841596504137608230, 25,
                                varx <= 0.869322188753617440, 26,
                                varx <= 0.897175152175194600, 27,
                                varx <= 0.925183680328846240, 28,
                                varx <= 0.953788843170826200, 29,
                                varx <= 0.981791877411713590, 30,
                                varx <= 0.981791877411713590, 31, &indx);
        break;
    case 2:
        select_pri_64bit_8val( varx <= 1.039409823987864900, 32,
                                varx <= 1.068692559293097600, 33,
\begin{verbatim}
\texttt{varx <= 1.098347233137172600, 34,}
\texttt{varx <= 1.128421928829294700, 35,}
\texttt{varx <= 1.158970386538573600, 36,}
\texttt{varx <= 1.190056245938955900, 37,}
\texttt{varx <= 1.221750217779271400, 38,}
\texttt{39, &indx); break;
\texttt{case 3:
 select_pri_64bit_4val( varx <= 1.287320295168777200, 40,
 varx <= 1.321418432469292400, 41,
 varx <= 1.356585716292107800, 42,
 43, &indx); break;
\texttt{case 4:
 select_pri_64bit_4val( varx <= 1.414213562373095100, 44,
 varx <= 1.414213562373095100, 44,
 varx <= 1.414213562373095100, 44,
 44, &indx); break;)

\texttt{y[i] = a[indx]*varx*varx + varx*b[indx] + c[indx];
 // printf ("i %3i a %f b %f c %f x %20.18f y %20.18f\n",
 //  indx,a[indx],b[indx],c[indx],varx,y[i]);
}
\end{verbatim}

read_timer (&tm1);
*time0 = tm1-tm0;

read_timer (&tm0);
\texttt{// Function 1}
\texttt{for (i=0; i<npts; i++)
  ysmap[i] = (1/sqrtf(2*3.14159265258979))*powf(2.71828182845905,-
    0.5*powf(x[i],2)); \texttt{// func 14}

//ysmap[i] = powf(2,x[i]); \texttt{// func 1}
//ysmap[i] = 1/x[i]; \texttt{// func 2}
//ysmap[i] = sqrtf(x[i]); \texttt{// func 3}
//ysmap[i] = 1/sqrtf(x[i]); \texttt{// func 4}
//ysmap[i] = logf(x[i])/0.693147180559945; \texttt{// func 5}
//ysmap[i] = logf(x[i]); \texttt{// func 6}
//ysmap[i] = sinf(x[i]*3.14159265258979); \texttt{// func 7}
//ysmap[i] = cosf(x[i]*3.14159265258979); \texttt{// func 8}
//ysmap[i] = tanf(x[i]*3.14159265258979); \texttt{// func 9}
//ysmap[i] = sqrt((-1*\texttt{logf(x[i]))}); \texttt{// func 10}
//ysmap[i] = powf(tanf(x[i]*3.14159265258979),2); \texttt{// func 11}
//ysmap[i] = -(x[i]*\texttt{\bf{logf(x[i])}}/0.693147180559944 + (1-x[i])*\texttt{\bf{logf(1-x[i])/0.693147180559944}}); \texttt{// func 12}
//ysmap[i] = 1/(1-powf(0.693147180559945,(-1*x[i]))); \texttt{// func 13}
//ysmap[i] = (1/sqrtf(2*3.14159265258979))*powf(2.71828182845905,-
    0.5*powf(x[i],2)); \texttt{// func 14

156
\end{verbatim}
//ysmap[i] = sinf(powf(2.71828182845905,x[i])); // func 15
read_timer (&tm1);

*time1 = tm1 - tm0;

/***************************
 * Send back the results
 ***************************/
nbytes = npts * 8;
DMA_CPU (OBM2CM, y, MAP_OBM_stripe(1,"F"), yc, 1, nbytes, 0);
wait_DMA (0);

nbytes = npts * 8;
DMA_CPU (OBM2CM, ysmap, MAP_OBM_stripe(1,"A"), ys, 1, nbytes, 0);
wait_DMA (0);
}

2. Fixed Point
   a. Main.c

#include<stdio.h>
#include<stdlib.h>
#include<strings.h>
#include<libmap.h>
#include<math.h>

// Subroutine initialization in Main
void   subr_map (int64_t acoef[],
  int     ncoef,
  int64_t x[],
  int64_t y[],
  int     xpts,
  int64_t *time0,
  int     mapnum);

// MAIN
main () {

  // Initialize Variables
  FILE    *fp1;
  int     i,ir,nc,xpts;
  int     mapnum,nmap,ncoef;
  int     arr_indx;
  int64_t *arraym,*xm,*ym;
  int64_t tm0,tml,hexval;
  char    hexstr[80], *token, *stpstr, strDelimit[]=" \n";

  // Starting NFG
  printf ("\n***START UP THE NFG ***\n");
  mapnum = 0;
nmap   = 1;
/allocate map to this problem
map_allocate (nmap);
nc = 300;

// array is enough room to hold 4 64 bit data pieces
// Perform cache allignment
arraym = (int64_t *)Cache_Aligned_Allocate (4*nc*8);
xm   = (int64_t *)Cache_Aligned_Allocate (nc*8  );
ym   = (int64_t *)Cache_Aligned_Allocate (nc*8  );

// Open the Hex data file to read
fp1   = fopen ("Data/memHEX0x.mem","r");
printf ("fp1 %i
",fp1);

// Read in the number of segments (decimal #)
fscafn (fp1, "%i", &ncoef);
printf ("  ncoef   %i
",ncoef);

// Get rid of first npc
fgets (hexstr, sizeof hexstr, fp1);

// Read all endpoints and coefficients into OBM banks
for (i=0;i<ncoef;i++)  {
    fgets (hexstr, sizeof hexstr, fp1);
    token   = strtok(hexstr,strDelimit);
    scanf (token, "%llx", &hexval);
    arraym[i*4]   = hexval;

    token   = strtok(NULL,strDelimit);
    scanf (token, "%llx", &hexval);
    arraym[i*4+1] = hexval;

    token   = strtok(NULL,strDelimit);
    scanf (token, "%llx", &hexval);
    arraym[i*4+2] = hexval;

    token   = strtok(NULL,strDelimit);
    scanf (token, "%llx", &hexval);
    arraym[i*4+3] = hexval;
}
fclose(fp1);

/*
// print out the contents of the array first 30 elements only
for (i=0;i<30;i++)  {
    printf ("endpoint: %llx a: %llx  b: %llx  c: %llx \n",
    arraym[i*4],arraym[i*4+1],arraym[i*4+2],arraym[i*4+3]);
}
*/

// create some values to test with
xpts = 30;
for (ir=0;ir<xpts;ir++)  {
    //arr_indx = (int) fabs(remainder(ir,20));
    arr_indx = ir % ncoef;
    xm[ir]   = arraym[arr_indx*4]/+0xa0000000;
printf ("arr_indx = %d  xm[%d]= %llx\n",arr_indx,ir,xm[ir]);
}
printf ("Right Before MAP *** \nmain ncoef %i xpts %i\n",ncoef,xpts);
subr_map (arraym, ncoef, xm, ym, xpts, &tm0, mapnum);
printf ("\n************Back from the MAP!!!********** \n");
printf ("\nI: %3d x values: %16llx   y values: %16llx \n",i,xm[i],ym[i]);

//  ! release the map resources
map_free (nmap);
}

b. subr.mc

#include <libmap.h>

void subr_map (int64_t ac[], int ncoef, int64_t xc[], int64_t yc[], int xpts, int64_t *time0, int mapno) {

OBM_BANK_A (segend, int64_t, MAX_OBM_SIZE)
OBM_BANK_B (a, int64_t, MAX_OBM_SIZE)
OBM_BANK_C (b, int64_t, MAX_OBM_SIZE)
OBM_BANK_D (c, int64_t, MAX_OBM_SIZE)
OBM_BANK_E (x, int64_t, MAX_OBM_SIZE)
OBM_BANK_F (y, int64_t, MAX_OBM_SIZE)
int i,j, nbytes, sel;
int64_t tm0,tml,indx,varx,varsq,vara,varb, varc,ax2,bx1,fx;

DMA_CPU (CM2OBM, segend, MAP_OBM_stripe(1,"A,B,C,D"), ac, 1, nbytes, 0);
wait_DMA (0);

// 4 data values (seg,a,b,c), 64bit Hex values
nbytes = 4*ncoef * 8;
DMA_CPU (CM2OBM, segend, MAP_OBM_stripe(1,"E"), xc, 1, nbytes, 0);
wait_DMA(0);

// DEBUG: determine when in Map
printf("\n\n************ NOW IN MAP **********\n");
printf("MAP subr ncoef %i xpts %i \n",ncoef,xpts);

/*******************************************************************************/
* Read timer and use selector to determine the segment *
/*******************************************************************************/
read_timer(&tm0);
for (i=0;i<xpts;i++)
{
  varx = x[i];

  if ( varx <= 0x000000001816a7a6)  
    sel =  1;
  else if ( varx <= 0x000000003b3b34a8)  
    sel =  2;
  else if ( varx <= 0x0000000040000000)  
    sel =  3;

  switch (sel)
  {
    case  1:
    select_pri_64bit_128val( varx <= 0x0000000000841cdf,   0,
                             varx <= 0x0000000000885b08,   1,
                             varx <= 0x0000000000cbea6,   2,
                             varx <= 0x00000000091438e,   3,
                             varx <= 0x00000000095ede6,   4,
                             varx <= 0x0000000009abdb9,   5,
                             varx <= 0x0000000009fb301,   6,
                             varx <= 0x000000000a4d1e3,   7,
                             varx <= 0x000000000a1a646,   8,
                             varx <= 0x000000000af8c81,   9,
                             varx <= 0x000000000b5283d,  10,
                             varx <= 0x000000000bafebf,  11,
                             varx <= 0x000000000c0e908,  12,
                             varx <= 0x000000000c71241,  13,
                             varx <= 0x000000000cd666a,  14,
                             varx <= 0x000000000d3fa84,  15,
                             varx <= 0x000000000dfabdb8, 16,
                             varx <= 0x000000000e1705,  17,
                             varx <= 0x000000000e8e66b,  18,
                             varx <= 0x000000000f05015,  19,
                             varx <= 0x000000000ff7f01, 20,
                             varx <= 0x000000000ffdf65a, 21,
                             varx <= 0x00000000107f71f,  22,
                             varx <= 0x000000001105651,  23,
                             varx <= 0x00000000118f818,  24,
                             varx <= 0x00000000120e09e,  25,
                             varx <= 0x0000000012b0fe3,  26,
                             varx <= 0x0000000013485e7,  27,
                             varx <= 0x0000000013e46d4,  28,
                             varx <= 0x0000000014856d2,  29,
varx <= 0x0000000000152b5e2,  30,
varx <= 0x000000000015d6404,  31,
varx <= 0x0000000000168698a,  32,
varx <= 0x0000000000173c675,  33,
varx <= 0x000000000017f7ac4,  34,
varx <= 0x000000000018b8aa1,  35,
varx <= 0x0000000000197fa35,  36,
varx <= 0x00000000001a4cda9,  37,
varx <= 0x00000000001b204fe,  38,
varx <= 0x00000000001bfa45d,  39,
varx <= 0x00000000001cdabc6,  40,
varx <= 0x00000000001dc238b,  41,
varx <= 0x00000000001eb0bad,  42,
varx <= 0x00000000001fa6855,  43,
varx <= 0x000000000020a3dac,  44,
varx <= 0x000000000021a8fdc,  45,
varx <= 0x000000000022b5ee4,  46,
varx <= 0x000000000023cb318,  47,
varx <= 0x000000000024e8c77,  48,
varx <= 0x0000000000260ef2a,  49,
varx <= 0x0000000000273e386,  50,
varx <= 0x00000000002876988,  51,
varx <= 0x000000000029b8985,  52,
varx <= 0x00000000002b0437b,  53,
varx <= 0x00000000002c59fbf,  54,
varx <= 0x00000000002db9e4f,  55,
varx <= 0x00000000002f2477f,  56,
varx <= 0x00000000003099f78,  57,
varx <= 0x0000000000321ae8c,  58,
varx <= 0x000000000033a74bc,  59,
varx <= 0x0000000000353fa5a,  60,
varx <= 0x000000000036e4390,  61,
varx <= 0x000000000038958b0,  62,
varx <= 0x00000000003a539bb,  63,
varx <= 0x00000000003c1f32c,  64,
varx <= 0x00000000003df892d,  65,
varx <= 0x00000000003fdffe7,  66,
varx <= 0x000000000041d5fac,  67,
varx <= 0x000000000043db0d1,  68,
varx <= 0x000000000045efba6,  69,
varx <= 0x00000000004814457,  70,
varx <= 0x00000000004a48f0b,  71,
varx <= 0x00000000004c8e83f,  72,
varx <= 0x00000000004ee541c,  73,
varx <= 0x0000000000514df1f,  74,
varx <= 0x0000000000538d71,  75,
varx <= 0x00000000005656765,  76,
varx <= 0x000000000058f7975,  77,
varx <= 0x00000000005bac7cd,  78,
varx <= 0x00000000005e75ee8,  79,
varx <= 0x00000000006154718,  80,
varx <= 0x000000000064488b1,  81,
varx <= 0x0000000000675302f,  82,
varx <= 0x00000000006a745e3,  83,
varx <= 0x00000000006dad222,  84,
varx <= 0x000000000070fe58f,  85,
varx <= 0x00000000007468456,  86,
varx <= 0x000000000077ebf1a,  87,
varx <= 0x00000000007b89e30,  88,
varx <= 0x00000000007f42e12,  89,
varx <= 0x0000000000817b3d,  90,
varx <= 0x0000000000870b2c,  91,
varx <= 0x00000000008b17f5e,  92,
varx <= 0x00000000008f45375,  93,
varx <= 0x00000000009396c6,  94,
varx <= 0x000000000097fd9f5,  95,
varx <= 0x00000000009c8a97f,  96,
varx <= 0x0000000000a139609,  97,
varx <= 0x0000000000a60b038,  98,
varx <= 0x0000000000b04089,  99,
varx <= 0x0000000000b0e4a1,  100,
varx <= 0x0000000000b559e25,  101,
varx <= 0x0000000000b904bb,  102,
varx <= 0x0000000000c0e808,  103,
varx <= 0x0000000000c605cda,  104,
varx <= 0x0000000000cbe6fb,  105,
varx <= 0x0000000000df3980,  106,
varx <= 0x0000000000d82c6c5,  107,
varx <= 0x0000000000e93079,  108,
varx <= 0x0000000000e52874,  109,
varx <= 0x0000000000ebdfeb,  110,
varx <= 0x0000000000f2e5370,  111,
varx <= 0x0000000000fa0f275,  112,
varx <= 0x00000000010165f3,  113,
varx <= 0x0000000001090160,  114,
varx <= 0x00000000010cc00a,  115,
varx <= 0x000000000118d071,  116,
varx <= 0x000000000120e6db,  117,
varx <= 0x000000000129826c,  118,
varx <= 0x000000000132f41d,  119,
varx <= 0x00000000013b3590f,  120,
varx <= 0x0000000001446c61,  121,
varx <= 0x00000000014de56e,  122,
varx <= 0x000000000157a397,  123,
varx <= 0x000000000161a759,  124,
varx <= 0x00000000016bf32a,  125,
varx <= 0x00000000017688d8,  126,
    127,   &indx);

break;

case  2:

  select_pri_64bit_32val(     varx <= 0x000000000189a234,  128,
                      varx <= 0x00000000019819a5b,  129,
                      varx <= 0x0000000001aeaeb8d,  130,
                      varx <= 0x0000000001b01193f,  131,
                      varx <= 0x0000000001bc8e294,  132,
                      varx <= 0x0000000001c963b27,  133,
                      varx <= 0x0000000001d69444,  134,
                      varx <= 0x0000000001e422789,  135,
                      varx <= 0x0000000001f20a6a,  136,
varx <= 0x0000000020061683, 137,
varx <= 0x0000000020f17574, 138,
varx <= 0x0000000021e350d9, 139,
varx <= 0x0000000022dbd679, 140,
varx <= 0x0000000023db2ff2, 141,
varx <= 0x0000000024e18b0b, 142,
varx <= 0x0000000025ef158a, 143,
varx <= 0x000000002703fd37, 144,
varx <= 0x0000000028207402, 145,
varx <= 0x000000002944a7b1, 146,
varx <= 0x000000002a70ce5e, 147,
varx <= 0x000000002ba519fa, 148,
varx <= 0x000000002ce1c09e, 149,
varx <= 0x000000002e26f43b, 150,
varx <= 0x000000002f74ef13, 151,
varx <= 0x0000000030cbe73f, 152,
varx <= 0x00000000322c12da, 153,
varx <= 0x000000003395ac27, 154,
varx <= 0x000000003508f191, 155,
varx <= 0x0000000036861933, 156,
varx <= 0x00000000380d5d4f, 157,
varx <= 0x00000000399efc52, 158,
159, &indx);
break;
case 3:

select_pri_64bit_4val( varx <= 0x000000003ce244bd, 160,
varx <= 0x000000003e9466d5, 161,
varx <= 0x0000000040000000, 162,
162, &indx);
break;
}

// ------ Shift by 8 bits - ------//
vara = a[indx];
varb = b[indx];
varx >>= 8; // Shift right 8 for mult 40.24
vara >>= 8; // Shift right 8
varb >>= 8; // Shift right 8

// ------ Square X and shift ----//
varsq = varx*varx; // Now we have 80.48 -> 16.48
varsq >>= 24; // SRL eliminate 40.24
varsq = varsq & 0x000000FFFFFFFFFF; // bitwise AND; 24bits

// -- X^2 * first Coefficient --//
ax2 = varsq*vara; // a[indx];
ax2 >>= 16; // Want 32.32, so srl 16
if (vara < 0x8000000000000000) // if both +ve
ax2 = ax2 & 0x0000FFFFFFFFFFF; // bitwise AND; 16bits

// --- X * second Coefficient --//
bx1 = varx*varb; // both are already shifted
bx1 >>= 16; // Return to 32.32 (int.fract)
if (varb < 0x8000000000000000) // if both +ve
bx1 = bx1 & 0x0000FFFFFFFFFFFF; // bitwise AND; 16bits

// -- 3 input add to complete --/
y[i] = ax2+bx1+c[indx]; // Add all, no need to shift varc

// DEBUG: printf for debug information on variable status
printf ("indx: %3i, varx: %8llx vasq: %10llx a: %10llx ax2:
%16llx b: %16llx bx1: %16llx c: %16llx f: %16llx \n",
    (int)indx, varx, varsq, vara, ax2,
    varb, bx1, c[indx], y[i]);

} // Time it took to compute
read_timer (&tm1);
*time0 = tm1-tm0;

/****************************************************************
*  Send back the results
****************************************************************/
nbytes = xpts * 8;
DMA_CPU (OBM2CM, y, MAP_OBM_stripe(1,"F"), yc, 1, nbytes, 0);
wait_DMA (0);
}

3. **Fixed Point with Macro**

This implementation did not produce the correct values. The multiplier macro used in this case was the VHDL macro shown in Appendix B.

The user can add macros to the *Makefile* that are coded in VHDL, Verilog or in both description languages. Here we show two VHDL files added to the *Makefile* and the *blk.v* and *info* files.

### a. Makefile

```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
```

164
# 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 = subr.mc

bin = nfg

# -----------------------------------
# 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)
# -----------------------------------
#macros = my_macro/mult_vrlg_64.v
#my_blkbox = my_macro/blk.v
#my Ngo Dir = my_macro
#my_info = my_macro/info
#macros = my_macro/mult_32to32.vhd \ 
# my_macro/add_32.vhd
#my_blkbox = my_macro/blk.v
#my Ngo Dir = my_macro
#my_info = my_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

MCCFLAGS = -log -explain_dep -g -keep -use_par
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 = icc   # for C codes
#LD = ifort # for Fortran or C/Fortran mixed

CFLAGS =
FFLAGS =
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)

b.  subr.mc

#include <libmap.h>

void subr_map (int64_t ac[],
               int    ncoef,
               int64_t xc[],
               int64_t yc[],
               int    xpts,
               int64_t *time0,
               int    mapno) {

    /****************************************************************
    * Declarations
    ****************************************************************/

    OBM_BANK_A (segend, int64_t, MAX_OBM_SIZE)
    OBM_BANK_B (a,        int64_t, MAX_OBM_SIZE)
    OBM_BANK_C (b,        int64_t, MAX_OBM_SIZE)
    OBM_BANK_D (c,        int64_t, MAX_OBM_SIZE)
    OBM_BANK_E (x,        int64_t, MAX_OBM_SIZE)
OBM_BANK_F (y, int64_t, MAX_OBM_SIZE)
int i, j, nbytes;
int64_t tm0, tm1, indx;
int varx, vara, varb, varc, prod3, prod2, prod1, fx;
int xg, ag, bg, cg;

/********************************************
* Read into OBM. Coeff & segment endpoints *
***************************************

// 4 data values (seg,a,b,c), 64bit Hex values
nbytes = 4*ncoef * 8;
DMA_CPU (CM2OBM, segend, MAP_OBM_stripe(1,"A,B,C,D"), ac, 1,
nbytes, 0);
wait_DMA (0);

// Read in the Number of points
nbytes = xpts * 8;
DMA_CPU (CM2OBM, x, MAP_OBM_stripe(1,"E"), xc, 1, nbytes, 0);
wait_DMA (0);

// DEBUG: Tell me I'm in the MAP
printf ("\n\n************ NOW IN MAP **********\n");
printf ("MAP subr ncoef %i xpts %i \
",ncoef, xpts);

/**********************************************************
* Read timer and use selector to determine the segment   *
**********************************************************/
read_timer (&tm0);
for (i=0; i<xpts; i++)
{
    split_64to32(x[i],&xg,&varx);

    // SEGMENT INDEX ENCODER
    // Based on x input, determine which index to select
    // the coefficients for approximation

    select_pri_32bit_16val( varx<= 0x12de, 0,
                              varx<= 0x2087, 1,
                              varx<= 0x2c8c, 2,
                              varx<= 0x37a9, 3,
                              varx<= 0x422b, 4,
                              varx<= 0x4c45, 5,
                              varx<= 0x5613, 6,
                              varx<= 0x5faa, 7,
                              varx<= 0x6916, 8,
                              varx<= 0x7268, 9,
                              varx<= 0x7bac, 10,
                              varx<= 0x7fff, 11,
                              varx<= 0x7fff, 11,
                              varx<= 0x7fff, 11,
                              varx<= 0x7fff, 11,
                              varx<= 0x7fff, 11,
                              varx<= 0x7fff, 11,
                              &indx);
indx = i%12;
split_64to32(a[indx],&ag,&vara);
split_64to32(b[indx],&bg,&varb);
split_64to32(c[indx],&cg,&varc);

// use macro multiplier
my_mult(varx,varx,&prod1);  // prod1 = x^2  term

// Perform together
my_mult(prod1,vara,&prod2);  // prod2 = ax^2 term
my_mult(varx, varb,&prod3);  // prod3 = bx  term

// Perform final add stage
//my_add(prod2,prod3,varc,&fx);  // 3 input macro adder
fx = prod2+prod3+varc;

// Perform final add stage
// Put result in OBM
y[i] = fx & 0x00000000FFFFFFFF;

// DEBUG: printf for debug information on variable status
//printf ("indx: %3i a[\%3]: %llx varb: %x c: %x x: %x fx: %lx, y[\%3]: %llx\n",
        indx,a[indx],varb,varc,varx,fx,y[i]);
// printf ("indx: %3i a: %x b: %x c: %x x: %x fx: %lx, y[\%3]: %llx\n",
        indx,vara,varb, varc, varx, fx, y[i]);
// printf ("prod1: %x prod2: %x prod3: %x \n",
// prodl,prod2,prod3);

}  // End for(i=0;i<xpts;i++)

read_timer (&tm1);  
*time0 = tm1-tm0;

/***************************************************************/
/* Send back the results
***************************************************************/
nbytes = xpts * 8;
DMA_CPU (OBM2CM, y, MAP_OBM_stripe(1,"F"), yc, 1, nbytes, 0);  
wait_DMA (0);
}

---

c. blk.v

module mult_32to32(a, b, clk, prod) /* synthesis syn_black_box */ ;
   input  [31:0] a;
   input  [31:0] b;
   output  [31:0] prod;
   input  clk;
   endmodule

module add_32(a, b, c, sum) /* synthesis adderparthere */ ;
   input  [31:0] a;

168
input [31:0] b;
input [31:0] c;
output [31:0] sum;
endmodule

d. info
BEGIN_DEF "my_mult"
MACRO = "mult_32to32";
STATEFUL = NO;
EXTERNAL = NO;
PIPELINED = YES;
LATENCY = 7;
INPUTS = 2:
   I0 = INT 32 BITS (a) // explicit input
   I1 = INT 32 BITS (b) // explicit input
;
OUTPUTS = 1:
   O0 = INT 32 BITS (prod) // explicit output
;
IN_SIGNAL : 1 BITS "clk" = "CLOCK";

DEBUG_HEADER = #
   void my_mult__dbg (int a, int b, int *prod);
#;

DEBUG_FUNC = #
   void my_mult__dbg (int a, int b, int *prod){
      *prod = a*b;
      *prod >>= 32;
   }
#
END_DEF

BEGIN_DEF "my_add"
MACRO = "add_32";
STATEFUL = NO;
EXTERNAL = NO;
PIPELINED = NO;
LATENCY = 1;
INPUTS = 3:
   I0 = INT 32 BITS (a) // explicit input
   I1 = INT 32 BITS (b) // explicit input
   I2 = INT 32 BITS (c) // explicit input
;
OUTPUTS = 1:
   O0 = INT 32 BITS (sum) // explicit output
;
DEBUG_HEADER = #
   void my_add__dbg (int a, int b, int c, int *sum);
#;
DEBUG_FUNC = #
   void my_add_dbg (int a, int b, int c, int *sum){
      *sum = a+b+c;
   }
#
END_DEF
APPENDIX D. COPY OF PROFILE REPORT

The profile report shows the execution time for non-uniform segmentation with the following parameters: $\sqrt{-\ln(x)}$, $e = 2^{-33}$ and $N = 1,000,000$. Profile reports are used to debug functions, optimize files and understand the dynamics and choke points in the program. Parent functions and child functions can be analyzed to find the slow points in the program.

The longest times in the report, 62.906s and 50.703s belong to xlabel and ylabel, respectively. They were used to display graphs for debugging purposes. Any function used to drive graphics is slow compared to computation. In a final version, the display is not required and these times do not exist and therefore have no impact.

The next longest functions are 29.063 seconds and 26.359 seconds which correspond to multipleQuadApprox and varQuadApproxHybThirdNew respectively. However notice that these are total times. multipleQuadApprox is a parent function to varQuadApproxHybThirdNew. Notice too that the column Self Time indicates the amount of time that the function actually spends in itself, i.e. the remaining time is spent in the child functions. The child function to varQuadApproxHybThirdNew is chebyRemez. This makes chebyRemez the longest part of the code. The child functions in chebyRemez take up a lot of time, but chebyRemz is the most suitable metric for comparing the speed of the different functions.

Profile Summary
Generated 28-Jul-2007 08:59:56

<table>
<thead>
<tr>
<th>Function name</th>
<th>Calls</th>
<th>Total Time</th>
<th>Self Time*</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserInput</td>
<td>1</td>
<td>0.094 s</td>
<td>0.094 s</td>
</tr>
<tr>
<td>ancestor</td>
<td>5252</td>
<td>1.141 s</td>
<td>function is recursive</td>
</tr>
<tr>
<td>ancestor&gt;isatype</td>
<td>10504</td>
<td>0.469 s</td>
<td>function is recursive</td>
</tr>
<tr>
<td>axes (Opaque-function)</td>
<td>5252</td>
<td>0.141 s</td>
<td>0.141 s</td>
</tr>
<tr>
<td>Function</td>
<td>Calls</td>
<td>Time (start)</td>
<td>Time (end)</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------</td>
<td>--------------</td>
<td>------------</td>
</tr>
<tr>
<td>axescheck</td>
<td>18356</td>
<td>0.859 s</td>
<td>0.859 s</td>
</tr>
<tr>
<td>cell.intersect</td>
<td>3</td>
<td>0.016 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>cell.setdiff</td>
<td>3</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>cell.sort</td>
<td>15</td>
<td>0.016 s</td>
<td>0.016 s</td>
</tr>
<tr>
<td>cell.strmatch</td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>cell.unique</td>
<td>9</td>
<td>0.016 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>cellfun (MEX-function)</td>
<td>49</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>cellstr</td>
<td>6</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>chebyRemz</td>
<td>4182</td>
<td>20.469 s</td>
<td>11.328 s</td>
</tr>
<tr>
<td>colstyle</td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>deal</td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>double.superiorfloat</td>
<td>34078</td>
<td>0.141 s</td>
<td>0.141 s</td>
</tr>
<tr>
<td>fcnchk</td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>findall</td>
<td>2</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>fliplr</td>
<td>35592</td>
<td>0.391 s</td>
<td>0.391 s</td>
</tr>
<tr>
<td>gca</td>
<td>13113</td>
<td>0.922 s</td>
<td>0.578 s</td>
</tr>
<tr>
<td>gcf</td>
<td>13113</td>
<td>0.391 s</td>
<td>0.391 s</td>
</tr>
<tr>
<td>getF</td>
<td>1</td>
<td>0.094 s</td>
<td>0.078 s</td>
</tr>
<tr>
<td>getappdata</td>
<td>6</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>graph2d.series.schema&gt;LdoDirtyAction</td>
<td>3</td>
<td>0.016 s</td>
<td>function is recursive</td>
</tr>
<tr>
<td>...h2d.series.schema&gt;LdoModeSwitchAction</td>
<td>2</td>
<td>0 s</td>
<td>function is recursive</td>
</tr>
<tr>
<td>....series.schema&gt;LdoSetManualModeAction</td>
<td>1</td>
<td>0 s</td>
<td>function is recursive</td>
</tr>
<tr>
<td>graph2d.series.schema&gt;LdoYDataAction</td>
<td>1</td>
<td>0 s</td>
<td>function is recursive</td>
</tr>
<tr>
<td>graph2d.series.schema&gt;LsetXDataSilently</td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>graphics/private\clo</td>
<td>2</td>
<td>0.031 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>graphics/private\clo&gt;find_kids</td>
<td>2</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>Function</td>
<td>Count</td>
<td>Time 1</td>
<td>Time 2</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td><code>handle.listener</code> (Opaque-function)</td>
<td>6</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td><code>hasbehavior</code></td>
<td>2</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td><code>hold</code></td>
<td>2623</td>
<td>1.422 s</td>
<td>0.438 s</td>
</tr>
<tr>
<td><code>inline.feval</code></td>
<td>164</td>
<td>0.031 s</td>
<td>0.016 s</td>
</tr>
<tr>
<td><code>inline.inline</code></td>
<td>3</td>
<td>0.094 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td><code>inline.inline&gt;strtrim</code></td>
<td>3</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td><code>inline_subsref</code></td>
<td>46123</td>
<td>12.172 s</td>
<td>1.563 s</td>
</tr>
<tr>
<td><code>inlineeval</code></td>
<td>46287</td>
<td>10.625 s</td>
<td>10.625 s</td>
</tr>
<tr>
<td><code>int2str</code></td>
<td>2622</td>
<td>0.219 s</td>
<td>0.219 s</td>
</tr>
<tr>
<td><code>intersect</code></td>
<td>3</td>
<td>0.016 s</td>
<td>0.016 s</td>
</tr>
<tr>
<td><code>isappdata</code></td>
<td>7871</td>
<td>0.563 s</td>
<td>0.344 s</td>
</tr>
<tr>
<td><code>iscell</code></td>
<td>9</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td><code>iscellstr</code></td>
<td>41</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td><code>isfield</code></td>
<td>7877</td>
<td>0.219 s</td>
<td>0.219 s</td>
</tr>
<tr>
<td><code>ishghandle</code></td>
<td>5252</td>
<td>0.250 s</td>
<td>0.250 s</td>
</tr>
<tr>
<td><code>ishold</code></td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td><code>iskeyword</code></td>
<td>2636</td>
<td>0.078 s</td>
<td>0.078 s</td>
</tr>
<tr>
<td><code>ismembc</code> (MEX-function)</td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td><code>ismembc2</code> (MEX-function)</td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td><code>ismember</code></td>
<td>8</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td><code>isObject</code></td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td><code>ispc</code></td>
<td>6</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td><code>isstruct</code></td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td><code>isvarname</code></td>
<td>2638</td>
<td>0.188 s</td>
<td>0.109 s</td>
</tr>
<tr>
<td><code>legend</code></td>
<td>1</td>
<td>0.016 s</td>
<td>function is recursive</td>
</tr>
<tr>
<td><code>legend&gt;find_legend</code></td>
<td>1</td>
<td>0 s</td>
<td>function is recursive</td>
</tr>
<tr>
<td>Function</td>
<td>Count</td>
<td>Time 1</td>
<td>Time 2</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>legend&gt;islegend</td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>legendinfo</td>
<td>1</td>
<td>0.016 s</td>
<td>function is recursive</td>
</tr>
<tr>
<td>legendinfo&gt;check_xydata</td>
<td>3</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>legendinfo&gt;parsestruct</td>
<td>2</td>
<td>0 s</td>
<td>function is recursive</td>
</tr>
<tr>
<td>line (Opaque-function)</td>
<td>3</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>linspace</td>
<td>4183</td>
<td>0.391 s</td>
<td>0.391 s</td>
</tr>
<tr>
<td>log10</td>
<td>2622</td>
<td>0.016 s</td>
<td>0.016 s</td>
</tr>
<tr>
<td>maple</td>
<td>36</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>maplemex (MEX-function)</td>
<td>36</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>meshgrid</td>
<td>4</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>multipleQuadApprox</td>
<td>1</td>
<td>29.063 s</td>
<td>0.188 s</td>
</tr>
<tr>
<td>newplot</td>
<td>2624</td>
<td>1.453 s</td>
<td>0.500 s</td>
</tr>
<tr>
<td>newplot&gt;ObserveAxesNextPlot</td>
<td>2624</td>
<td>0.641 s</td>
<td>0.047 s</td>
</tr>
<tr>
<td>newplot&gt;ObserveFigureNextPlot</td>
<td>2624</td>
<td>0.094 s</td>
<td>0.094 s</td>
</tr>
<tr>
<td>num2str</td>
<td>5244</td>
<td>0.828 s</td>
<td>0.594 s</td>
</tr>
<tr>
<td>opaque.double</td>
<td>11</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>parseparams</td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>plotdoneevent</td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>polyval</td>
<td>34078</td>
<td>1.922 s</td>
<td>1.781 s</td>
</tr>
<tr>
<td>quadl</td>
<td>1</td>
<td>0.078 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>quadl&gt;quadlstep</td>
<td>163</td>
<td>0.078 s</td>
<td>function is recursive</td>
</tr>
<tr>
<td>scribe.legendinfo (Opaque-function)</td>
<td>4</td>
<td>0 s</td>
<td>function is recursive</td>
</tr>
<tr>
<td>scribe.legendinfo.legendinfo</td>
<td>1</td>
<td>0 s</td>
<td>function is recursive</td>
</tr>
<tr>
<td>scribe.legendinfochild</td>
<td>12</td>
<td>0 s</td>
<td>function is recursive</td>
</tr>
<tr>
<td>scribe.legendinfochild.legendinfochild</td>
<td>3</td>
<td>0 s</td>
<td>function is recursive</td>
</tr>
<tr>
<td>setdiff</td>
<td>7</td>
<td>0.078 s</td>
<td>0.063 s</td>
</tr>
<tr>
<td>Function Name</td>
<td>Calling Count</td>
<td>Elapsed Time</td>
<td>CPU Time</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>---------------</td>
<td>--------------</td>
<td>----------</td>
</tr>
<tr>
<td>sortcellchar (MEX-function)</td>
<td>15</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>specgraph.baseline (Opaque-function)</td>
<td>7</td>
<td>0.047 s</td>
<td></td>
</tr>
<tr>
<td>specgraph.baseline.baseline</td>
<td>1</td>
<td>0.047 s</td>
<td></td>
</tr>
<tr>
<td>specgraph.stemseries (Opaque-function)</td>
<td>25</td>
<td>0.078 s</td>
<td></td>
</tr>
<tr>
<td>specgraph.stemseries.refresh</td>
<td>2</td>
<td>0.016 s</td>
<td></td>
</tr>
<tr>
<td>...stemseries.schema&gt;LdoEdgeColorAction</td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>...stemseries.schema&gt;LdoFaceColorAction</td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>...ies.schema&gt;LdoSetManualCodeModeAction</td>
<td>3</td>
<td>0 s</td>
<td></td>
</tr>
<tr>
<td>...aph.stemseries.schema&gt;LdoUpdateAction</td>
<td>2</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>...series.schema&gt;LdoUpdateBaselineAction</td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>...ies.schema&gt;LdoUpdateChildMarkerAction</td>
<td>2</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>...series.schema&gt;LdoUpdateChildrenAction</td>
<td>3</td>
<td>0.016 s</td>
<td></td>
</tr>
<tr>
<td>...temseries.schema&gt;LdoUpdateXDataAction</td>
<td>2</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>specgraph.stemseries.setLegendInfo</td>
<td>1</td>
<td>0.016 s</td>
<td></td>
</tr>
<tr>
<td>specgraph.stemseries.stemseries</td>
<td>1</td>
<td>0.063 s</td>
<td></td>
</tr>
<tr>
<td>specgraph\private\checkpvpairs</td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>specgraph\private\nextstyle</td>
<td>1</td>
<td>0.016 s</td>
<td>0.016 s</td>
</tr>
<tr>
<td>stem</td>
<td>1</td>
<td>0.109 s</td>
<td>0.016 s</td>
</tr>
<tr>
<td>stem&gt;parseargs</td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>str2num</td>
<td>10</td>
<td>0.016 s</td>
<td>0.016 s</td>
</tr>
<tr>
<td>str2num&gt;protected_conversion</td>
<td>10</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>strmatch</td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>sym.abs</td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>sym.char</td>
<td>16</td>
<td>0.016 s</td>
<td>0.016 s</td>
</tr>
<tr>
<td>sym.diff</td>
<td>3</td>
<td>0.016 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>sym.eq</td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td>Function</td>
<td>Count</td>
<td>Self Time</td>
<td>Elapsed Time</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------</td>
<td>-----------</td>
<td>--------------</td>
</tr>
<tr>
<td><code>sym.findsym</code></td>
<td>3</td>
<td>0.016 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td><code>sym.findsym&gt;pickvar</code></td>
<td>3</td>
<td>0.016 s</td>
<td>0.016 s</td>
</tr>
<tr>
<td><code>sym.log</code></td>
<td>2</td>
<td>0.047 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td><code>sym.maple</code></td>
<td>10</td>
<td>0.047 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td><code>sym.sqrt</code></td>
<td>2</td>
<td>0.016 s</td>
<td>0.016 s</td>
</tr>
<tr>
<td><code>sym.sym</code></td>
<td>1329</td>
<td>0.266 s</td>
<td>0.063 s</td>
</tr>
<tr>
<td><code>sym.sym&gt;char2sym</code></td>
<td>1324</td>
<td>0.203 s</td>
<td>0.094 s</td>
</tr>
<tr>
<td><code>sym.sym&gt;trim</code></td>
<td>1324</td>
<td>0.031 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td><code>sym.uminus</code></td>
<td>2</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td><code>syms</code></td>
<td>1314</td>
<td>0.578 s</td>
<td>0.219 s</td>
</tr>
<tr>
<td><code>symvar</code></td>
<td>3</td>
<td>0.094 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td><code>symvar&gt;findrun</code></td>
<td>12</td>
<td>0.016 s</td>
<td>0.016 s</td>
</tr>
<tr>
<td><code>symvar&gt;isquoted</code></td>
<td>3</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td><code>title</code></td>
<td>5244</td>
<td>1.234 s</td>
<td>function is recursive</td>
</tr>
<tr>
<td><code>unique</code></td>
<td>4</td>
<td>0.016 s</td>
<td>0.016 s</td>
</tr>
<tr>
<td><code>usev6plotapi</code></td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td><code>varQuadApproxHybThirdNew</code></td>
<td>1311</td>
<td>26.359 s</td>
<td>3.297 s</td>
</tr>
<tr>
<td><code>vectorize</code></td>
<td>3</td>
<td>0.031 s</td>
<td>0.031 s</td>
</tr>
<tr>
<td><code>xlabel</code></td>
<td>5244</td>
<td>62.906 s</td>
<td>function is recursive</td>
</tr>
<tr>
<td><code>xychk</code></td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
</tr>
<tr>
<td><code>ylabel</code></td>
<td>5244</td>
<td>50.703 s</td>
<td>function is recursive</td>
</tr>
</tbody>
</table>

**Self time** is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling.
APPENDIX E. LESSONS LEARNED

This section provides information and a record of problems that were encountered while using the SRC-6, and other software applications in this thesis. The intent is to provide a reference to specific issues previously encountered and to reduce the amount of time to resolve or understand them in the future.

E.1 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

E.2 USING THE `CONST` CONSTRUCT IN C

**Problem:** A `martello64` error is obtained when using

```c
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

```c
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

### E.3 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

```c
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.

```c
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

### E.4 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.5 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

### E.6 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
E.7 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

E.8 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])
    );
The ROM initialization values are in the .sdc file below. The INITs 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.

```text
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 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.

```text
module MY_ROM (  
din_0,  
dout_0,  
din_1,  
dout_1,  
adr_0,  
adr_1,  
w_en_0,  
w_en_1,  
clk  
);
```
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.

```binary
0000011001100100010000000000000000000000
000001100011011010000000000000000000000
00000101111111111100000000000000000000
00000101101101000000000000000000000000
00000101010110000000000000000000000000
00000100111101010000000000000000000000
00000100011100000000000000000000000000
00000111110111111000000000000000000000
00000111111111111100000000000000000000
```

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.
E.9 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

E.10 CANNOT USE PRIORITY SELECTOR GREATER THAN 128

Problem: When implementing a priority selector with 256 elements, 64 bits wide, I could not compile the .mc file. This is because the architecture already had 3 64 bit wide multipliers and other hardware that consumed some of the resources. However, if you don’t need all 256 priority selectors, it would be nice to have a selector that is greater than 128, and smaller than 256.

Background: When implementing the priority selectors with 150 elements, the only option for a single selector is to use the 256 selector, but that is 106 more elements than required.

Solution: Use multiple selectors of smaller sizes.

Author: N. Macaria
Date: 26JUL07
E.11 IF-THEN-ELSE STATEMENT WITH SRC PRIORITY SELECTORS

Problem: When implementing multiple priority selectors in the .mc file, SRC would not accept an if-then-else statement to contain priority selectors in the body.

Background: When running the program, it would not compile if a priority selector was used inside an if-then-else statement.

Solution: Put the if-then-else statement prior to the priority selector, use a variable to store the selector you want to pick, then use a case statement to reach that selector.

Author: N. Macaria
Date: 26JUL07

E.12 FIND THE SLOW CODE IN MATLAB PROGRAMS

Problem: When running MATLAB programs, sometimes the code takes very long to execute and you may not be sure where the problem exists.

Background: When running the chebyRemz, program, there were portions of code that would take very long to run.

Solution: Put the if-then-else statement prior to the priority selector, use a variable to store the selector you want to pick, then use a case statement to reach that selector.

Author: N. Macaria
Date: 26JUL07
APPENDIX F. SEGMENT ESTIMATION EQUATION

The segment estimation equation is derived from analyzing the Chebyshev approximation error equation (0.6) is the general case:

\[
e = \frac{2(b-a)^{d+1}}{4^{d+1}(d+1)!} |f_{\text{max}}^{d+1}(x)|
\]  

(0.6)

The variable \(d\) is the order of the approximation to be used. For the case of quadratic approximation, \(d=2\) and \((b-a)\) is the estimated width of the segment.

\[
e = \frac{2(b-a)^3}{4^3(3)!} |f_{\text{max}}^{iii}(x)|
\]

\[
\frac{(4^3 \times 3 \times 2)e}{2 \times |f_{\text{max}}^{iii}(x)|} = (b-a)^3
\]

\[
(b-a)^3 = \frac{(4^3 \times 3 \times 2)e}{2 \times |f_{\text{max}}^{iii}(x)|} = (4^3) \times \frac{3e}{|f_{\text{max}}^{iii}(x)|}
\]

\[
(b-a) = \sqrt[3]{(4^3) \times \frac{3e}{|f_{\text{max}}^{iii}(x)|}}
\]

\[
\text{EstLenSeg} = (b-a) = 4 \times \left[ \frac{3e}{|f_{\text{max}}^{iii}(x)|} \right]^{\frac{1}{3}}
\]

(0.3)

185
LIST OF REFERENCES


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

7. Prof. Douglas Fouts
   Department of Applied Mathematics
   Naval Postgraduate School
   Monterey, California

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

9. Mr. Alan Hunsberger
   National Security Agency
   Ft. Meade, Maryland
10. Dr. John Harkins
   National Security Agency
   Ft. Meade, Maryland

11. Dr. Pedro Claudio
    Staff Systems Engineer
    Technical Operations Applied Research
    Lockheed Martin Missiles and Fire Control
    Orlando, Florida

12. Mr. David Caliga
    SRC Computers
    Colorado Springs, Colorado

13. Mr. John Huppenthal
    SRC Computers
    Colorado Springs Colorado

14. LT Tim Knudstrup
    Department of Electrical and Computer Engineering
    Naval Postgraduate School
    Monterey, California