Optimizing Hardware Function Evaluation

Oskar Mencer, Wayne Luk and Haohuan Fu
Department of Computing
Imperial College London
{o.mencer, wl, hfu}@doc.ic.ac.uk

March, 12, 2008

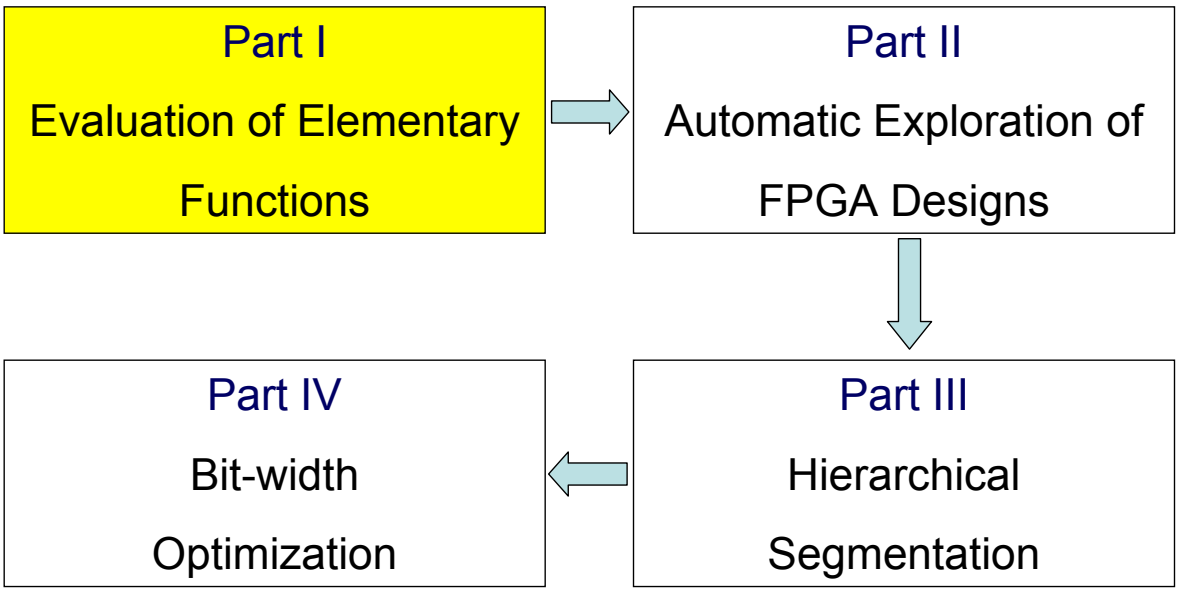
Acknowledgement

- The work of Dong-U Lee*, J.D. Villasenor*, Ray C.C. Cheung**, D.J. Pearce**, A. Abdul Gaffar**, G.A. Constantinides***, and P.Y.K. Cheung*** on function evaluation is gratefully acknowledged.
- We especially thank them for their help and support on planning and production of this tutorial.

* Electrical Engineering Department, UCLA
** Department of Computing, Imperial College London
*** Department of EEE, Imperial College London

Tutorial Outline

- Evaluation of Elementary Functions
- Automatic Exploration of FPGA Designs
- Hierarchical Segmentation
- Bit-width Optimization
- Further Reading



1. Evaluation of Elementary Functions

1.1 Elementary function: definition and usage

1.2 Elementary function evaluation

1.3 Function evaluation algorithms

1.4 Case study: evaluation of $\log_2(x)$

1.1 Elementary Functions

- Definition:
  - built from a finite number of exponentials, logarithms, constants, one variable, and roots of equations through composition and combinations using four elementary operations (+ - × ÷).
  - examples: trigonometric, exponential, and logarithmic functions …

$$y = \ln(-x^2)$$

$$y = \sin(\sqrt{1 + \ln^2 x})$$

Elementary Functions: Applications

- Scientific computations
- Signal processing
  - sine/cosine generator
- Engineering simulation
  - over 60% of total runtime of a jet engine simulation spent on elementary function calculations [1]

[1] E.O'Grady and C.Wang. "Performance Limitations in Parallel Processor Simulations". Trans. Soc. Computer Simulation. Vol.4, pp.311-330,1987.

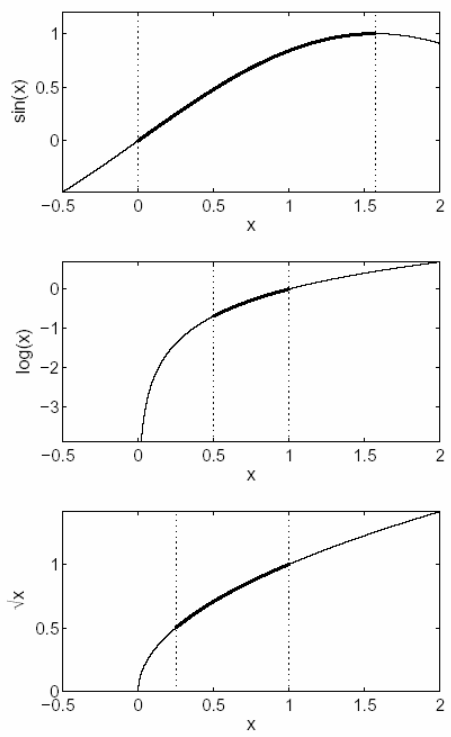
1.2 Elementary Function Evaluation

- In theory, not harder than computing quotient
  - similar to division in terms of Boolean circuit depth [2]
- In practice, hardware design requires optimizing
  - speed, area, power consumption
  - accuracy
  - performance

[2] H. Alt. "Comparison of arithmetic functions with respect to Boolean Circuits". In Proceedings of the 16th ACM STOC, pp. 466-470, 1984.

A General 3-step Approach

- f(x) where x=[a,b]
  - range reducing x to a more convenient interval y=[a',b']
  - function approximation on the reduced interval
  - range reconstruction: expanding the result back to the original range



1.3 Function Evaluation Algorithms

- Direct look-up table
- Polynomial/rational approximation
- Piecewise polynomial/rational approximation
- CORDIC

Polynomial/Rational Approximation

- Polynomial approximation

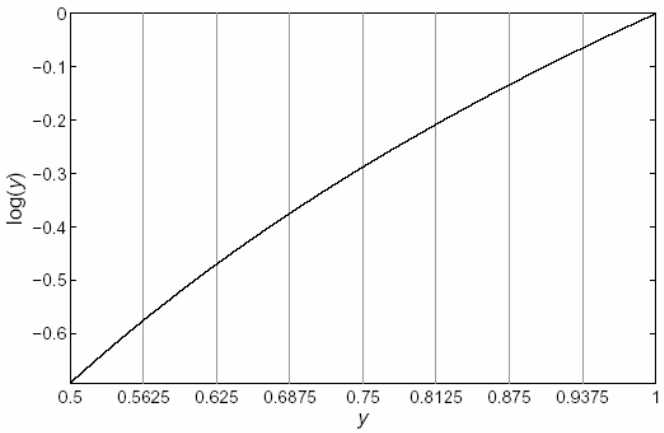
$$f(x) \approx p(x) = c_0 \cdot x^n + c_1 \cdot x^{n-1} + \cdots + c_k \cdot x^{n-k} + \cdots + c_n$$

- Rational approximation

$$f(x) \approx q(x) = \frac{c_0 \cdot x^n + c_1 \cdot x^{n-1} + \cdots + c_k \cdot x^{n-k} + \cdots + c_n}{d_0 \cdot x^m + d_1 \cdot x^{m-1} + \cdots + d_k \cdot x^{m-k} + \cdots + d_m}$$

Piecewise polynomial or rational approximation

- Divide the evaluation range into multiple segments
- Approximate each segment with a polynomial or rational expression



CORDIC

- COordinate Rotation DIgital Computer:
  - An efficient method to compute functions using shifts, additions and small lookup table
  - Key idea:

$$x' = x\cos(\phi) - y\sin(\phi) = \cos(\phi)(x - y\tan(\phi))$$

$$y' = y\cos(\phi) + x\sin(\phi) = \cos(\phi)(y - x\tan(\phi))$$

If we keep the rotation angle ϕ to be special values that $\tan(\phi) = \pm 2^{-i}$, the multiplication can be calculated with shifts.

1.4 Errors in Function Evaluation

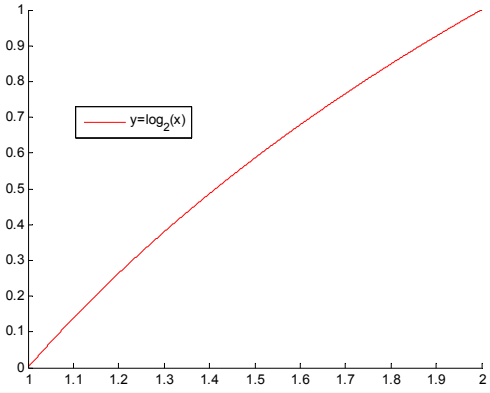
- 'total error' = 'approximation error' + 'quantization error'
- Approximation error:
  - inherent error of the approximation method
  - equals to the total error if we perform the approximation method with infinite precision
- Quantization error:
  - rounding / truncation error
  - due to limited precision of the variables in practical computation

Errors in Function Evaluation

- Define G = 'approximation error' / 'total error'.
- To keep total error below 1 ulp (unit at last place)
  - quantization error > 0.5 ulp,
    due to the rounding / truncation in the last step.
  - thus, we need to keep approximation error < 0.5 ulp,
    G < 0.5.
- Empirically, G = 0.3 is a good ratio and produces efficient designs with less resource.

1.5 Case Study: Evaluation of $\log_2(x)$

- Evaluation of $\log_2(x)$ over the range [1, 2)
  - input format: 24-bit (1:23) fixed-point number.
  - output format: 23-bit (0:23) fixed-point number.
  - error requirement: 1 ulp = 2^{-23}.
- Evaluation approach:
  - piecewise polynomial approximation



Approximation Error →Algorithm

- Reduce approximation error to G*1 ulp = $0.3*2^{-23} \approx 3.58e\text{-}8$
  - increase polynomial degree
  - divide the range into more segments
- Suppose we fix the polynomial degree to be 2, and divide the range into 2^K segments:

K	1	2	3	4	5	6	7
$error_{max}$	9.97e-4	1.67e-4	2.45e-5	3.35e-6	4.38e-7	5.60e-8	7.08e-9

- K=7 meets the approximation error requirement by dividing the range into 128 uniform segments.

Approximation Algorithm

- Evaluation of $\log_2(x)$ over range [1, 2), error<2^{-23}.
  - 128 uniform segments of length 2^{-7}.
  - each segment calculated with a degree-2 polynomial.

$$y = (c_0 \cdot x + c_1) \cdot x + c_2$$

Conversion of x and Coefficients

- e.g., for the first segment [1, $1+2^{-7}$], using minimax, we get the following polynomial for evaluation:

$$y = -0.71575 \cdot x^2 + 2.87418 \cdot x - 2.15843$$

where x = | 1. | 0000000 | other bits |

- Further convert x into x'= | 0. | other bits | using

$$x' = (x-1) \cdot 2^7$$

and calculate with the converted polynomial coefficients:

$$y = -0.000043686 \cdot x'^2 + 0.011271 \cdot x' - 0.0000000070827$$

Signals: Bit-width Determination

- Signals in $y = (c_0 \cdot x + c_1) \cdot x + c_2$
  - coefficients: c_0, c_1, c_2
  - intermediate variables:

$$d_0 = c_0 \cdot x \qquad d_1 = d_0 + c_1 \qquad d_2 = d_1 \cdot x$$

- Bit-width determination
  - integer bit-width: range.
  - fractional bit-width: precision.

Range Determination → Integer Bit-width

- e.g., for the first segment [1, $1+2^{-7}$].

signal	range	IB (integer bits)
c_0	-4.3686e-5	-15
c_1	0.011271	-7
c_2	7.08272e-9	-26
d_0	[-4.3686e-5, 0]	-15
d_1	[0.011227, 0.011271]	-7
d_2	[0, 0.011227]	-7
y	[7.08272e-9, 0.011227]	-7

- We summarize the results of all different segments to get the range and IB of the signals.

Precision: Quantization Error Model

- Suppose no error in input x ([0,1]), maximum quantization errors of the signals are:
  - coefficients:

$$e_{c_0} = 2^{-1-FB_{c_0}} \quad e_{c_1} = 2^{-1-FB_{c_1}} \quad e_{c_2} = 2^{-1-FB_{c_2}}$$ (round to nearest)

  - intermediate results:

$$d_0 = c_0 \cdot x \longrightarrow e_{d_0} = e_{c_0} \cdot x_{\max} + 2^{-FB_{d_0}} = e_{c_0} + 2^{-FB_{d_0}}$$ (truncation)

$$d_1 = d_0 + c_1 \longrightarrow e_{d_1} = e_{d_0} + e_{c_1} + 2^{-FB_{d_1}}$$ (truncation)

$$d_2 = d_1 \cdot x \longrightarrow e_{d_2} = e_{d_1} \cdot x_{\max} + 2^{-FB_{d_2}} = e_{d_1} + 2^{-FB_{d_2}}$$ (truncation)

$$y = d_2 + c_2 \longrightarrow e_q = e_y = e_{d_2} + e_{c_2} + 2^{-1-FB_y}$$ (round to nearest)

Quantization Error → Fractional Bit-width

- Suppose we use UFB for all signals

$$UFB = FB_{c_0} = FB_{c_1} = FB_{c_2} = FB_{d_0} = FB_{d_1} = FB_{d_2}$$

the total error is

$$e_{total} = e_a + e_q = 7.08e-9 + 4.5 \times 2^{-UFB} + 2^{-1-FB_y} < 2^{-23} \approx 1.192e-7$$

($FB_y = 23$)

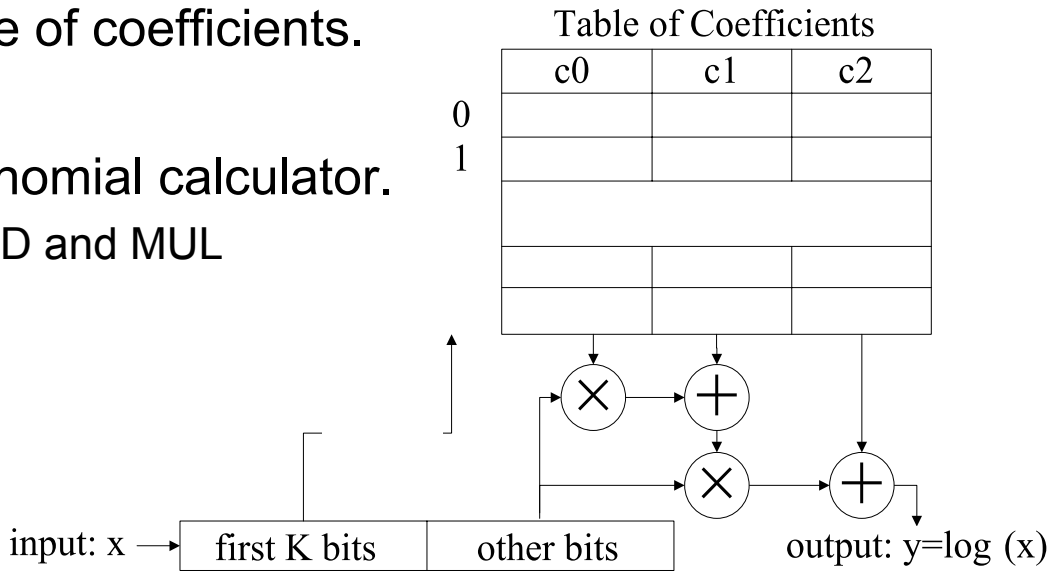
Note: e_{total} is total error, e_a is approximation error, and e_q is quantization error.

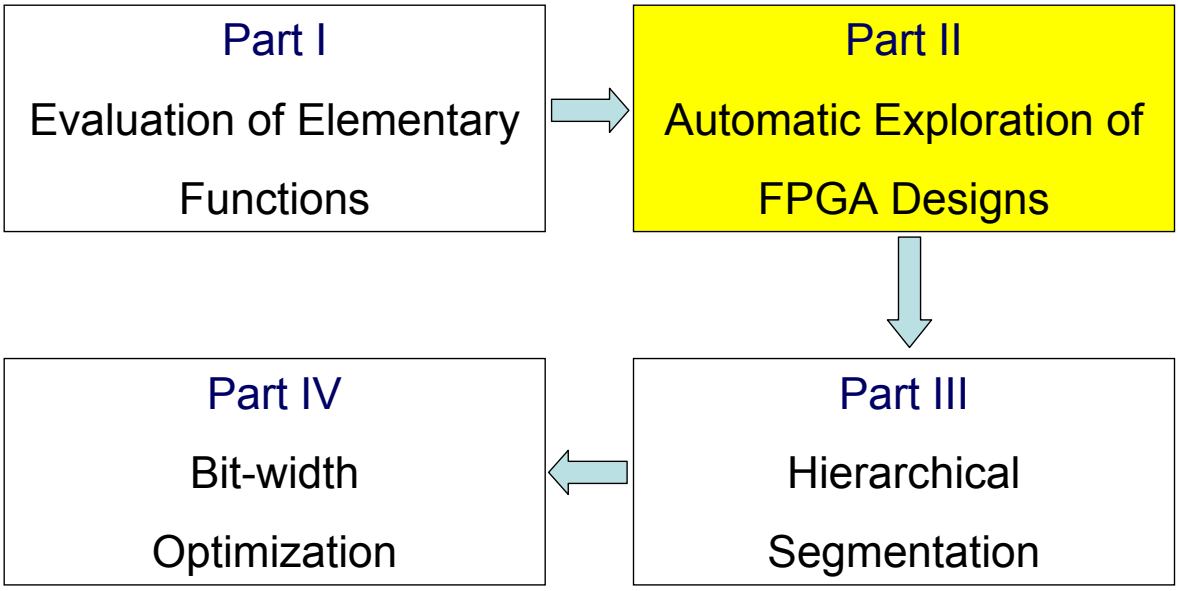
which gives UFB = 28.

- A more detailed discussion in Part IV.

Hardware Architecture

- Table of coefficients.
- Polynomial calculator.
  - ADD and MUL





2. Automatic Exploration of FPGA Designs

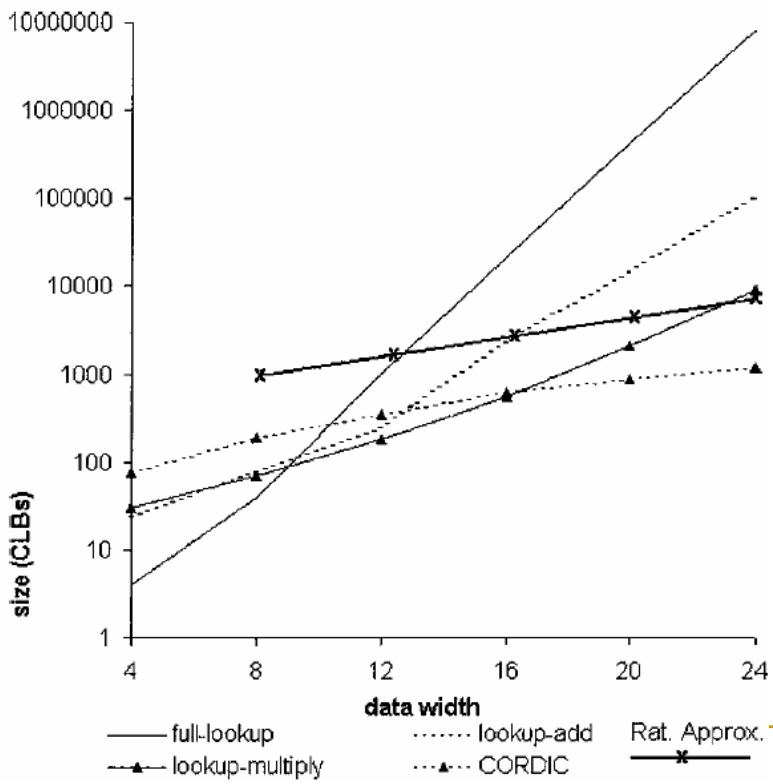
2(a) Optimizing evaluation method

1. Introduction
2. Methodology: Matlab and ASC
3. Case study: optimizing evaluation method for sin(x)
4. Summary

2(b) Adaptive range reduction

1. Introduction
2. Background
3. Case study: sin(x), log(x) and sqrt(x)
4. Summary

Function Evaluation on FPGA



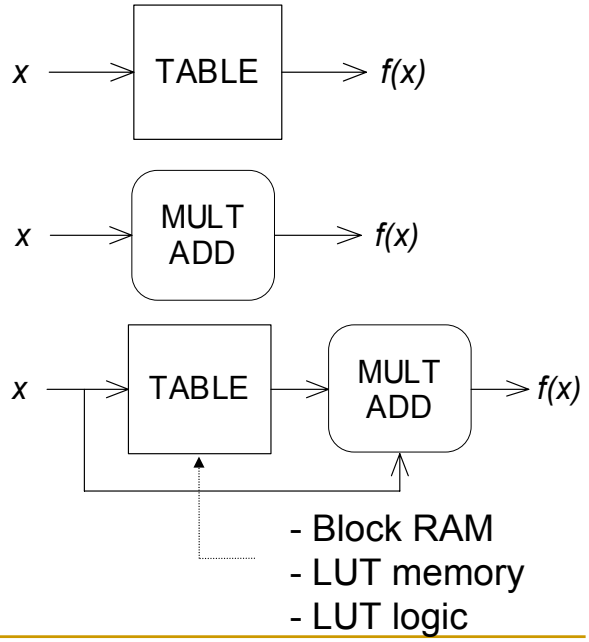
- Optimal method = f(precision,metric)
- FPGAs
  - min cost/area
  - max performance

Automatic Design Exploration (a):

Optimizing Evaluation Method

2(a).1 Introduction

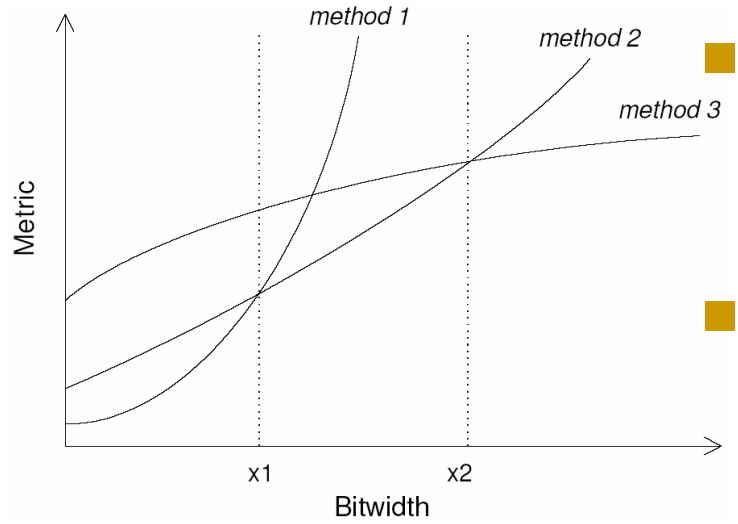
- Function evaluation methods:
  - table lookup
  - polynomial only
  - table with polynomial



Degrees of Freedom

- Evaluation method selection:
  TABLE, POLY or TABLE+POLY
- Optimize option:
  AREA, LATENCY or THROUGHPUT
- Bit-width: 8, 12, 16, 20 and 24
- Memory type:
  Block RAM, LUT Memory, LUT logic

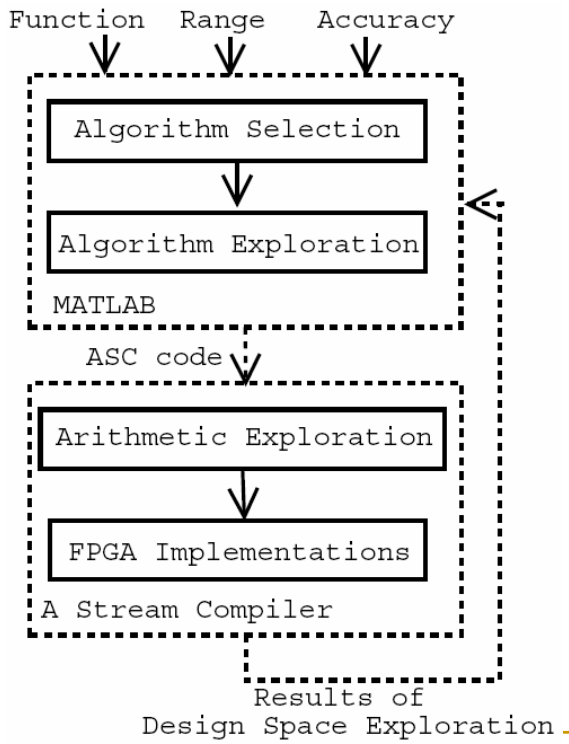
Automatic Design Exploration



- Different methods with various parameters
- Judge with uniform metrics

Very large design space => automate optimization

2(a).2 Methodology



- MATLAB automates algorithm level exploration of function approximation.
- A Stream Compiler, ASC automates design space exploration.

(1) MATLAB: Algorithmic Design Space Exploration

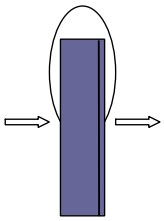
- Input parameters:
  - function, input range, operand bitwidth, accuracy, bitwidth of coefficients, polynomial degree.
- Three MATLAB programs generate ASC code:
  - TABLE (table lookup): generates a single table, holding results for all possible inputs.
  - POLY (polynomial only): finds polynomial degree and coefficients.
  - TABLE+POLY (table with polynomial): finds the number of uniform segments and coefficients.
- Binary search finds optimum uniform bit-widths

(2) ASC: Hardware Design Space Exploration

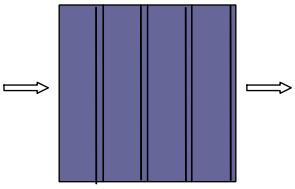
- Software-like programming of FPGAs, allows user to specify design space exploration: bitwidths of variables, optimization metric, etc...
- `RUN0 = -DBITWIDTH={8;16;24;32}`
  - in makefile: investigate the specified bitwidth on a variable, ASC generates and collects post-Xilinx-tools result data and graphs.
- MATLAB provides ASC with a set of ASC programs, which result in ~400 separate FPGA implementations.

Optimization in ASC

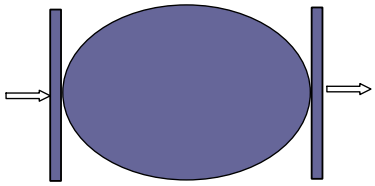
AREA Opt:
sequential
- loop x times



THROUGHPUT Opt:
pipelined
- loop unrolled



LATENCY Opt:
combinational
- loop unrolled
- no registers
- logic minimization



2(a).3 Case Study: Optimizing Evaluation Method for sin(x)

- Using proposed method, we automatically explore various sin(x) implementations
- At Matlab level
  - three different evaluation method:
    TABLE, POLY, and TABLE+POLY
  - five different input/output bit-width:
    8, 12, 16, 20 and 24
  - algorithmic space in MATLAB:
    3 methods, 5 bitwidths → 15 designs

Optimizing Evaluation Method for sin(x)

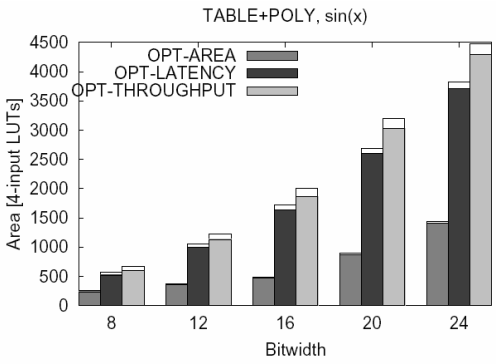
- At ASC level:
  - three different memory types:
    Block RAM, LUT Memory, LUT logic
  - three different optimization options:
    latency, throughput, area
  - design space in ASC:
    15 designs from Matlab, 3 memory types,
    3 optimization options → 135 circuit designs
- 135 circuit designs mapped into Xilinx Virtex-II XC2V6000 device
- One run with a single `make' file → 4 hours on a dual Athlon XP PC

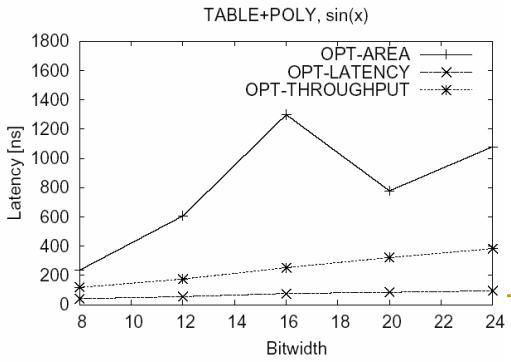
Comparison Results of Different Memory Types

ASC optimization	memory type	4-input LUTs	clock speed [MHz]	latency [ns]	throughput [Mbps]
latency	block RAM	**919 + 1BRAM**	17.89	**111.81**	250.41
	LUT memory	1086	15.74	63.51	220.43
	LUT logic	**813**	16.63	**60.11**	232.93
throughput	block RAM	**919 + 1BRAM**	39.49	177.28	**552.79**
	LUT memory	1086	36.29	192.88	508.09
	LUT logic	**967**	39.26	178.29	**549.67**

- Logic minimized LUTs better in latency and area, than LUT memory
- => convert tables to logic!

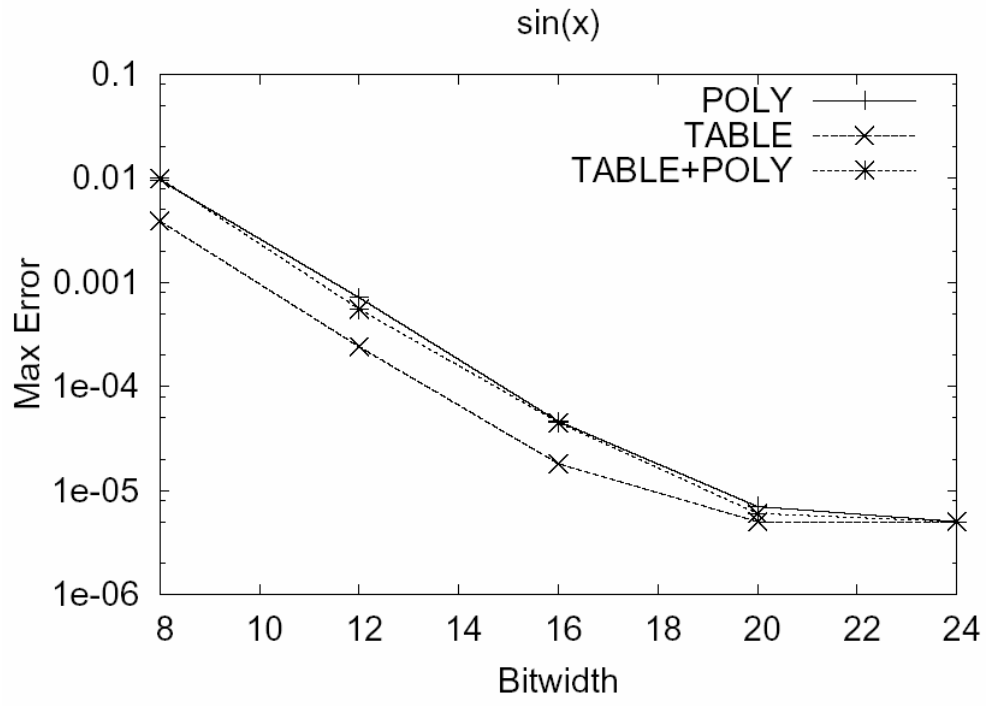
Comparison Results of Different Optimization Options



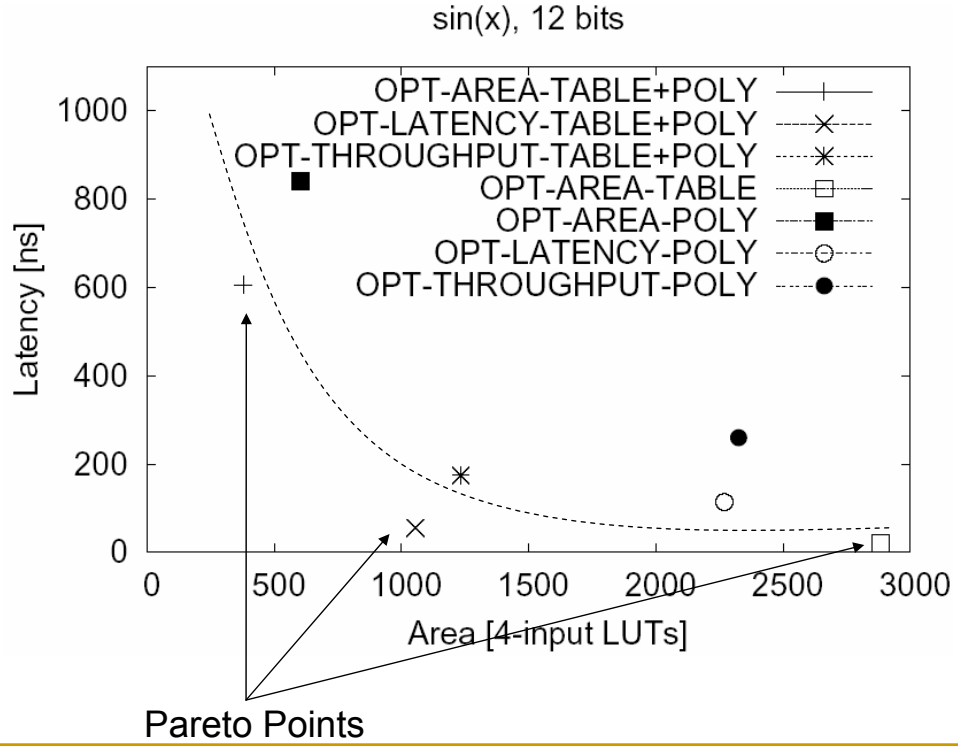


- Area
  - OPT-AREA: least area
  - increase with precision
  - routing overhead very small
- Latency
  - OPT-LATENCY: least latency
  - increase with precision

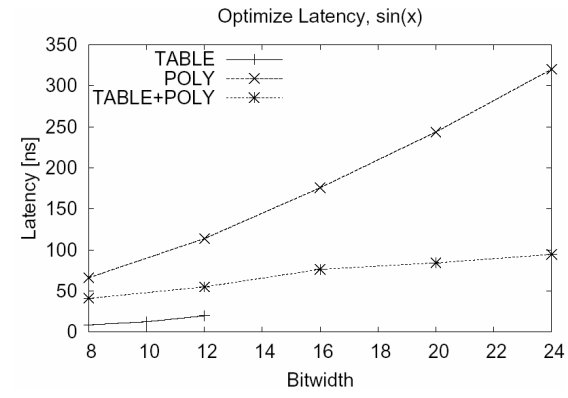
Accuracy Graphs of Different Methods

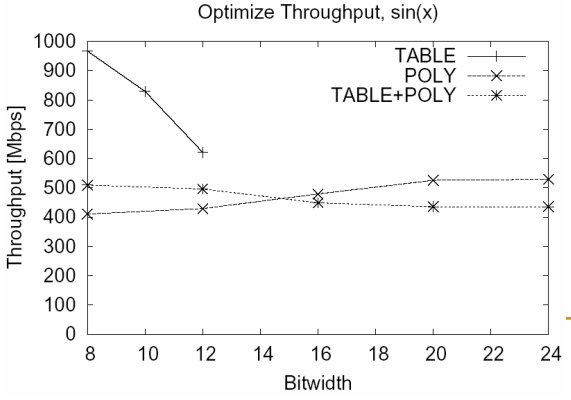


12-bit sin(x), Pareto Points



sin(x) - which method is best?





- Latency
  - lowest: TABLE
  - highest: POLY
- Throughput

bw <= 12: TABLE

12 < bw <= 14: TABLE+POLY

bw > 14: POLY

2(a).4 Summary

- Selecting the optimal evaluation method.
- Automatic productivity/design space exploration
  - MATAB: algorithmic
  - ASC: hardware
- Case study:
  - optimizing evaluation method of sin(x)
  - MATLAB + ASC → 135 designs in 4 hours

Automatic Design Exploration (b):

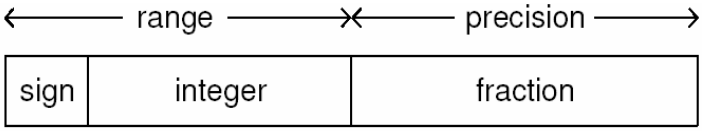
Adaptive Range Reduction

2(b).1 Introduction

- Adaptive range reduction based on parametric function evaluation library by polynomials/tables
- Automatic exploration of different parameters using MATLAB program and ASC system
- Case study for sin(x), log(x) and sqrt(x)

2(b).2 Background

- Function evaluation consists of:
  - range reduction
  - the actual approximation over a small interval
- Range reduction: widely studied for floating-point numbers on microprocessors
- Lack of attention on hardware implementation of function approximation with range reduction
- We consider sign-magnitude fixed-point and results accurate to 1 ulp (unit at last place)



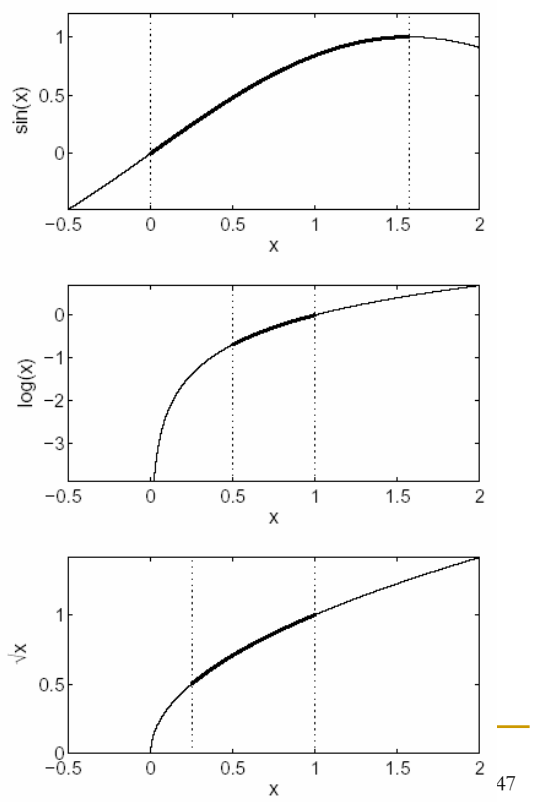
Range Reduction

- Additive reduction

$$x' = x - mC$$

- Multiplicative reduction

$$x' = x / C^m$$



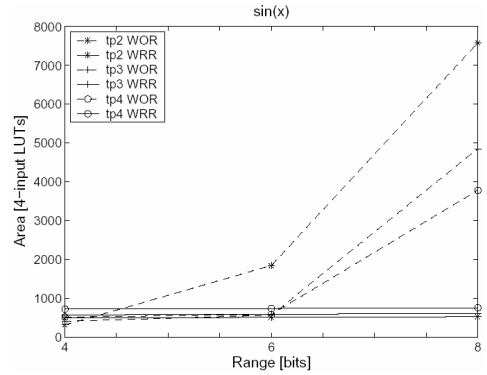
2(b).3 Case Study: sin(x), log(x) and sqrt(x)

- sin(x): $x' = x - 2\pi \cdot k$
  - reduce the range of x into $[0, \pi/2]$
- log(x): $x' = x / 2^k$
  - reduce the range of x into [0.5, 1]
- sqrt(x): $x' = x / 2^{2k}$
  - reduce the range of x into [0.25, 1].

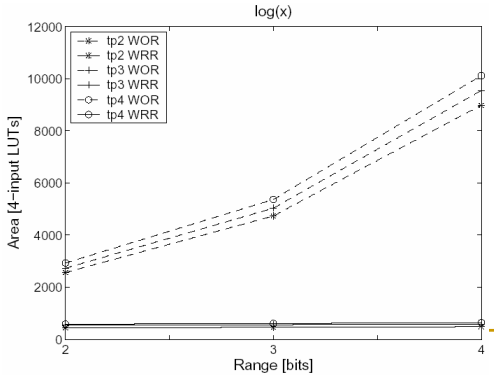
Automatic Design Exploration

- Applicability of range reduction (sin(x), log(x)):
  - WRR (with range reduction)
  - WOR (without range reduction)
- Evaluation method:
  - po, tp2, tp3, tp4
- Optimize option:
  - AREA, LATENCY or THROUGHPUT
- Bit-width:
  - range (integer bits): 4, 8, 12, 16
  - precision (fractional bits): 4, 8, 12, 16
- Matlab+ASC: over 1000 designs explored

WRR vs WOR (sin(x), log(x))

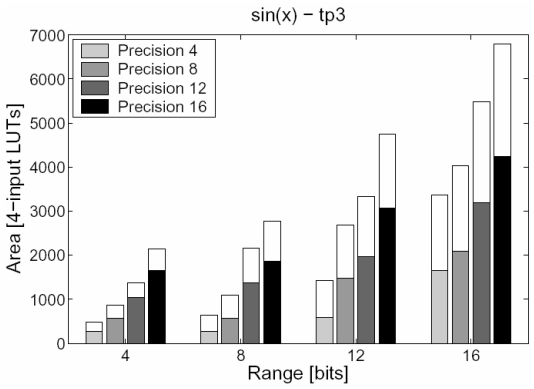


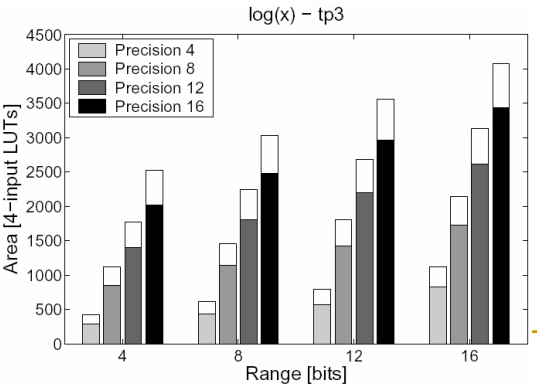
- sin(x): *WOR* smaller for ranges lower than 6 bits



- log(x): *WOR* always larger due to non-linearity of log(x)

Area Cost of Range Reduction





- Lower part: area for function approximation
- Upper part: area for range reduction
- Average range reduction percentage:
  - sin(x): 41% (divider)
  - log(x): 22% (barrel shifter)

Area and Latency Matrices

- Which method to use for minimal area/latency?

Precision [bits]

16	sin: tp2 log: tp2 sqrt: tp2	sin: tp2 log: tp2 sqrt: tp2	sin: tp2 log: tp2 **sqrt: tp3**	sin: tp2 log: tp2 **sqrt: tp3**
12	sin: tp2 log: tp2 sqrt: tp2	sin: tp2 log: tp2 sqrt: tp2	sin: tp2 log: tp2 sqrt: tp2	sin: tp2 log: tp2 sqrt: tp2
8	sin: tp2 log: tp2 sqrt: tp2	sin: tp2 log: tp2 sqrt: tp2	sin: tp2 log: tp2 sqrt: tp2	sin: tp2 log: tp2 sqrt: tp2
4	sin: tp2 log: tp2 **sqrt: po**	**sin: po** log: tp2 sqrt: tp2	sin: tp2 log: tp2 sqrt: tp2	**sin: po** **log: po** sqrt: tp2
	4	8	12	16

Range [bits]

Precision [bits]

16	sin: tp2 log: tp2 sqrt: tp2	sin: tp2 log: tp2 sqrt: tp2	sin: tp2 log: tp2 sqrt: tp2	sin: tp2 log: tp2 sqrt: tp2
12	sin: tp2 log: tp2 sqrt: tp2	sin: tp2 log: tp2 sqrt: tp2	sin: tp2 log: tp2 sqrt: tp2	sin: tp2 log: tp2 sqrt: tp2
8	sin: tp2 log: tp2 sqrt: tp2	sin: tp2 log: tp2 sqrt: tp2	sin: tp2 log: tp2 sqrt: tp2	sin: tp2 log: tp2 sqrt: tp2
4	**sin: po** **log: po** **sqrt: po**	sin: tp2 **log: po** sqrt: tp2	**sin: po** log: tp2 sqrt: tp2	sin: tp2 **log: po** sqrt: tp2
	4	8	12	16

Range [bits]

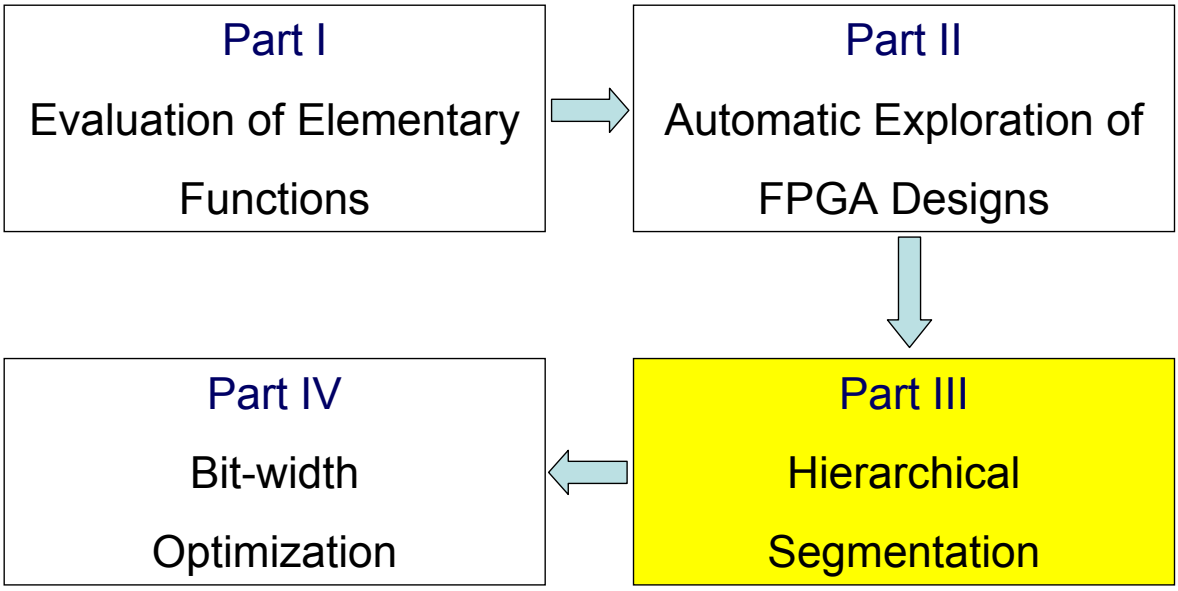
Latest Results: 2000 Designs

Minimal Latency (Optimized for Latency)

Range [bits] \ Precision [bits]	4	8	12	16	20	24
24	**sin**: *tp2*, 24, 78 **log**: *po2*, 8, 30 **sqr**: *tp2*, 18, 912	**sin**: *tp2*, 28, 348 **log**: *tp2*, 12, 78 **sqr**: *tp2*, 24, 2400	**sin**: *tp2*, 32, 792 **log**: *tp2*, 15, 384 **sqr**: *tp2*, 26,10368	**sin**: *tp2*, 32, 3168 **log**: *tp2*, 21, 1056 **sqr**: *tp2*, 30,23808	**sin**: *tp2*, 40, 7872 **log**: *tp2*, 24, 4800 **sqr**: *tp3*, 34,17920	**sin**: *tp3*, 42, 5504 **log**: *tp2*, 28,11136 **sqr**: *tp4*, 39,12800
20	**sin**: *po2*, 19, 61 **log**: *po2*, 7, 27 **sqr**: *tp2*, 18, 456	**sin**: *tp2*, 24, 300 **log**: *tp2*, 12, 78 **sqr**: *tp2*, 20, 2016	**sin**: *tp2*, 28, 696 **log**: *tp2*, 15, 384 **sqr**: *tp2*, 24, 4800	**sin**: *tp2*, 32, 3168 **log**: *tp2*, 21, 1056 **sqr**: *tp2*, 32,12288	**sin**: *tp2*, 32, 6336 **log**: *tp2*, 24, 4800 **sqr**: *tp3*, 33, 8704	**sin**: *tp3*, 38, 4992 **log**: *tp2*, 27,10752 **sqr**: *tp3*, 37,19456
16	**sin**: *tp2*, 16, 54 **log**: *po2*, 7, 27 **sqr**: *tp2*, 17, 324	**sin**: *tp2*, 19, 240 **log**: *tp2*, 12, 78 **sqr**: *tp2*, 18, 912	**sin**: *tp2*, 24, 600 **log**: *tp2*, 15, 384 **sqr**: *tp2*, 24, 2400	**sin**: *tp2*, 28, 2784 **log**: *tp2*, 21, 1056 **sqr**: *tp2*, 26,10368	**sin**: *tp2*, 32, 6336 **log**: *tp2*, 24, 4800 **sqr**: *tp2*, 30,23808	**sin**: *tp3*, 32, 4224 **log**: *tp2*, 27,10752 **sqr**: *tp3*, 34,17920
12	**sin**: *po2*, 9, 31 **log**: *po2*, 7, 27 **sqr**: *tp2*, 12, 156	**sin**: *tp2*, 16, 204 **log**: *tp2*, 12, 78 **sqr**: *tp2*, 18, 456	**sin**: *tp2*, 20, 504 **log**: *tp2*, 15, 384 **sqr**: *tp2*, 20, 2016	**sin**: *tp2*, 24, 2400 **log**: *tp2*, 21, 1056 **sqr**: *tp2*, 24, 4800	**sin**: *tp2*, 28, 5568 **log**: *tp2*, 24, 4800 **sqr**: *tp2*, 31,12288	**sin**: *tp2*, 32,12672 **log**: *tp2*, 27,10752 **sqr**: *tp3*, 33, 8704
8	**sin**: *po2*, 6, 22 **log**: *po2*, 7, 27 **sqr**: *tp2*, 7, 27	**sin**: *tp2*, 10, 132 **log**: *tp2*, 11, 72 **sqr**: *tp2*, 17, 324	**sin**: *tp2*, 16, 408 **log**: *tp2*, 15, 384 **sqr**: *tp2*, 18, 912	**sin**: *tp2*, 19, 1920 **log**: *tp2*, 21, 1056 **sqr**: *tp2*, 24, 2400	**sin**: *tp2*, 24, 4800 **log**: *tp2*, 24, 4800 **sqr**: *tp2*, 26,10368	**sin**: *tp2*, 28,11136 **log**: *tp2*, 27,10752 **sqr**: *tp2*, 30,23808
4	**sin**: *po2*, 6, 22 **log**: *po2*, 7, 27 **sqr**: *po2*, 7, 25	**sin**: *tp2*, 10, 132 **log**: *tp2*, 11, 72 **sqr**: *tp2*, 12, 156	**sin**: *tp2*, 15, 384 **log**: *tp2*, 15, 384 **sqr**: *tp2*, 18, 456	**sin**: *tp2*, 19, 1920 **log**: *tp2*, 17, 1056 **sqr**: *tp2*, 20, 2016	**sin**: *tp2*, 23, 4608 **log**: *tp2*, 24, 4800 **sqr**: *tp2*, 24, 4800	**sin**: *tp2*, 28,11136 **log**: *tp2*, 27,10752 **sqr**: *tp2*, 31,12288

2(b).5 Summary

- MATLAB+ASC →
  1000 designs automatically explored
- Adaptive range reduction based on parametric function evaluation library by polynomials/tables
- For given functions, use area/latency matrices to choose the best method.



3. Hierarchical Segmentation

3.1 Introduction

3.2 Approach
- segmentation hierarchies
- adapt to non-linearities of a function

3.3 Case studies: 6 functions

3.1 Introduction

- Function evaluation -
  central to numerous applications:
  - communications, computer graphics, digital signal processing, scientific computing
- Methods based on piecewise polynomials -
  popular for hardware implementations:
  - uniform segmentation: all segments of equal width, total number of segments constrained to power of 2
  - simple coefficient address computation, but can be problematic for non-linear functions

Previous Work

- Lewis [3]: 2-level uniform segmentation for approximating LNS functions
- Coleman et al. [4]: 2-level segmentation using
  - segments that vary by powers of two
  - uniform segments
- Both cover a subset of the schemes considered here; neither consider automation

[3] D. Lewis. “Interleaved Memory Function Interpolators with Application to an Accurate LNS arithmetic Unit”. *Trans. Computers*. Vol. 43, No. 8, pp.974 - 982, 1994.

[4] J.N. Coleman et al. “Arithmetic on the European Logarithmic Microprocessor”. *Trans. Computers*. Vol. 49, No. 7, pp. 702 - 715, 2000.

Previous Work

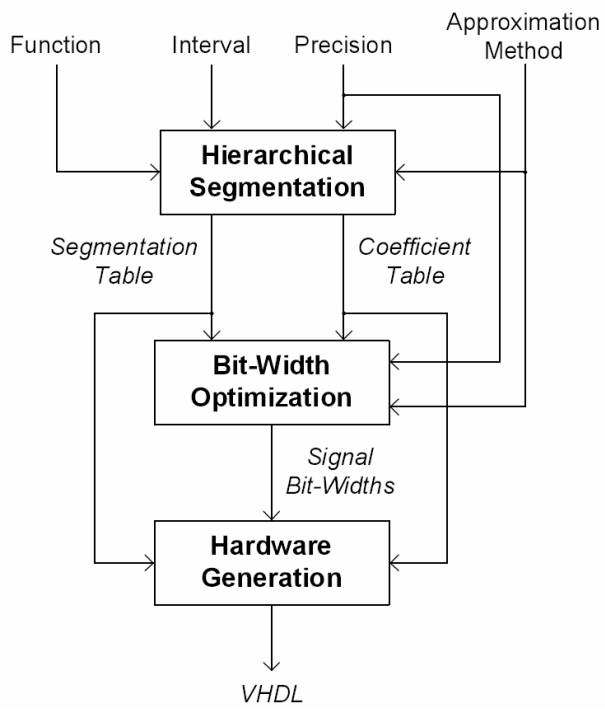
- T. Sasao et al. [5]: balanced error segmentation
  - a form of non-uniform segmentation
  - errors in all segments are equal
- Benefit
  - minimized number of segments
- Drawback
  - circuitry for computing the segment address can be complex

[5] T. Sasao et al. “Programmable Numerical Function Generators: Architectures and Synthesis Method”. *Proc. FPL*, 2005.

3.2 Approach

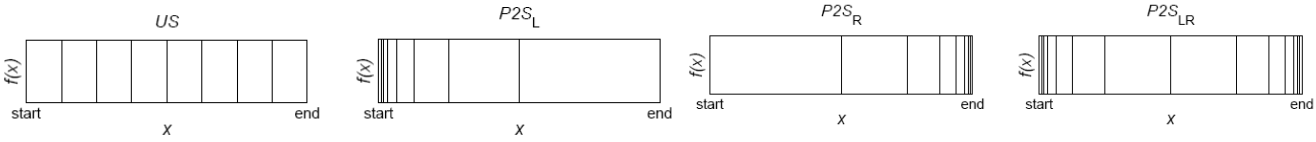
- Employ segmentation hierarchies
  - uniform segments
  - segments with size varying by powers of two
- Adapt to non-linearities of a function
  - significant reduction in number of segments compared to uniform segmentation

Design Flow



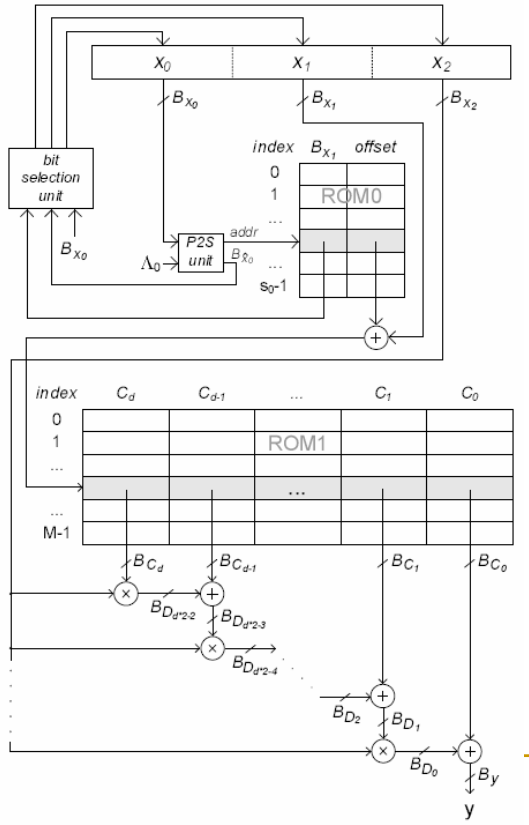
Hierarchical Segmentation Framework

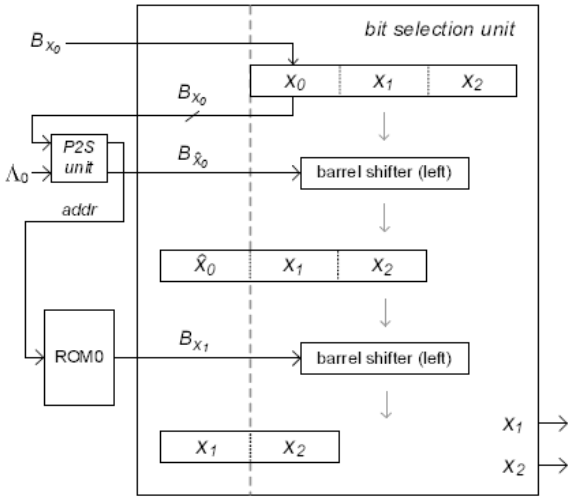
- 4 segmentation schemes: US, $P2S_L$, $P2S_R$, and $P2S_{LR}$



- Apply segmentation recursively
  - First pass, entire interval subdivided using one of the 4 schemes
  - Next pass, each segment further subdivided using one of the 4 schemes
- 2 passes where second pass is US: sufficient for many functions

Hardware Architecture





3.3 Case Studies

- Six different functions:

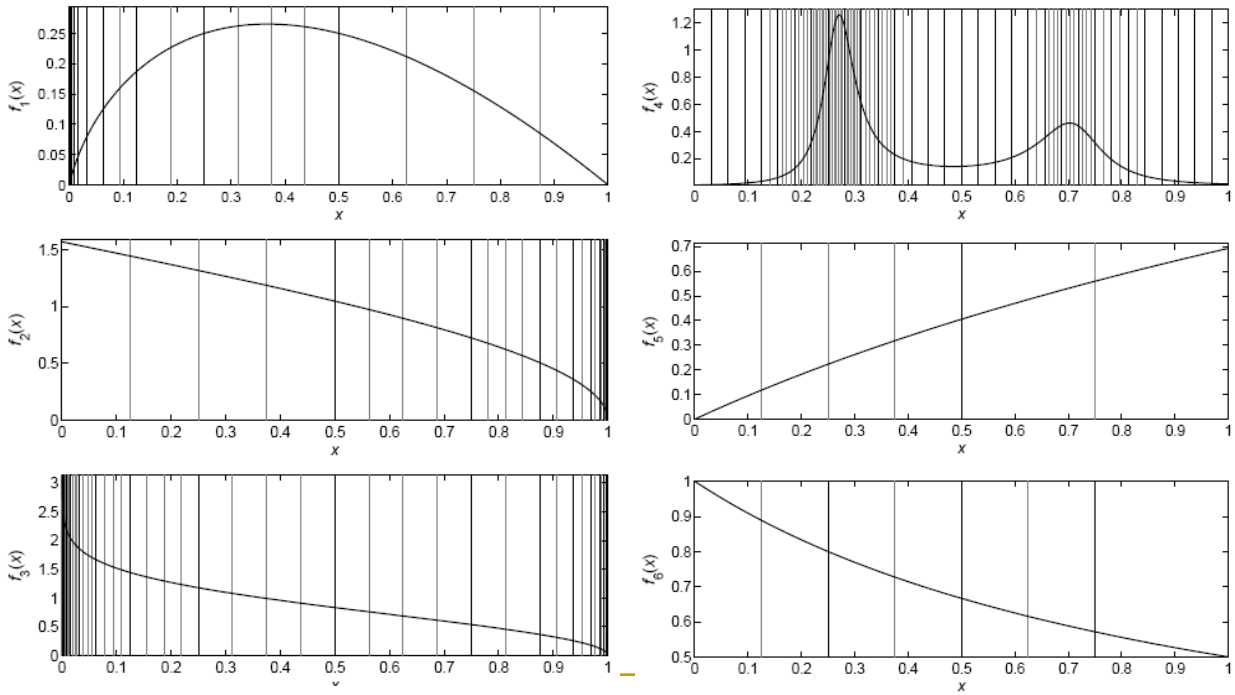
$$f_1 = -\frac{x}{2}\log_2 x \qquad f_2 = \cos^{-1}(x)$$

$$f_3 = \sqrt{-\ln(x)}$$

$$f_4 = \frac{0.0004x + 0.0002}{x^4 - 1.96x^3 + 1.348x^2 - 0.378x + 0.0373}$$

$$f_5 = \ln(1+x) \qquad f_6 = 1/(1+x)$$

Segmentation of the Six Functions



Degree-2 piecewise polynomials with an error requirement of 2^{-14}

Segment Count Comparisons

Function	ϵ_{req}	Uniform	Hierarchical	Balanced [5]
f_1	2^{-8}	8	5	3
	2^{-16}	4,096	31	19
	2^{-24}	1,048,576	191	120
f_2	2^{-8}	16	6	4
	2^{-16}	32,768	49	30
	2^{-24}	8,388,608	352	219
f_3	2^{-8}	128	11	7
	2^{-16}	262,144	126	78
	2^{-24}	67,108,864	1,076	547
f_4	2^{-8}	64	20	12
	2^{-16}	512	124	76
	2^{-24}	2,048	740	422
f_5	2^{-8}	2	2	2
	2^{-16}	8	8	7
	2^{-24}	128	55	45
f_6	2^{-8}	4	3	3
	2^{-16}	16	12	8
	2^{-24}	128	76	50

[5] T. Sasao et al. “Programmable Numerical Function Generators: Architectures and Synthesis Method”. *Proc. FPL*, 2005.

Memory Requirement Comparisons

- Comparisons between [5] and hierarchical segmentation for evaluating:

$$f_3 = \sqrt{-\ln(x)} \text{ OVER } x = [2^{-5}, 1) \text{ AND } f_5 = \ln(1+x) \text{ OVER } x = [0, 1)$$

Method	f_3				f_5			
	14-bit Precision		22-bit Precision		15-bit Precision		23-bit Precision	
	Degree-1	Degree-2	Degree-1	Degree-2	Degree-1	Degree-2	Degree-1	Degree-2
Balanced [5]	74,240	11,264	2,662,400	173,056	20,096	2,448	700,416	19,136
Hierarchical	16,548 (240)	5,836 (180)	524,571 (652)	76,387 (468)	3,238 (448)	522 (32)	63,392 (896)	4,921 (448)
Reduction Factor	4.5	1.9	5.1	2.3	6.2	4.7	11.0	3.9

- Balanced approach [5]
  - fewer segments
  - higher overall memory requirement: complex coefficient address indexing circuitry

[5] T. Sasao et al. “Programmable Numerical Function Generators: Architectures and Synthesis Method”. *Proc. FPL*, 2005.

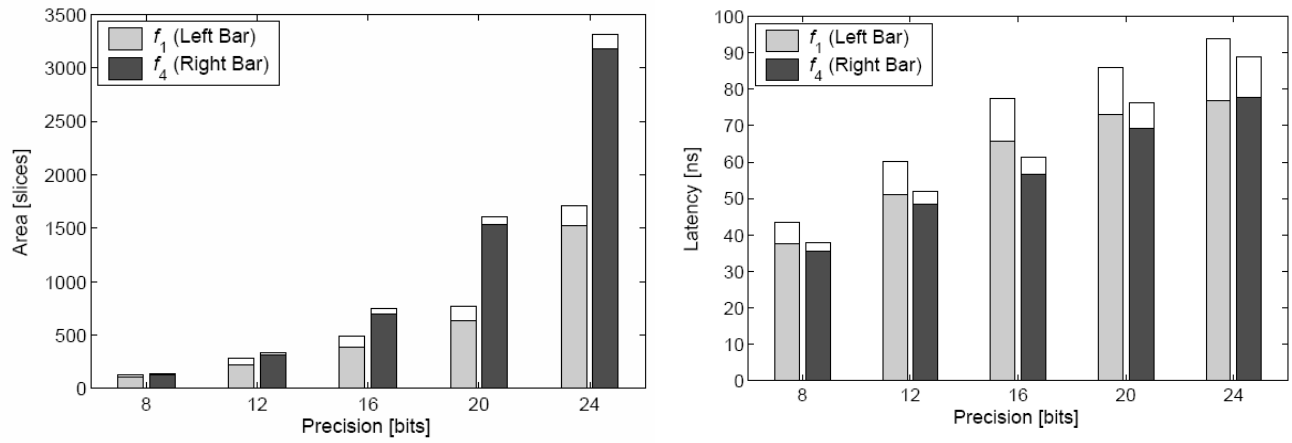
Optimization: Run Times

- Degree-2 approximations to f_4 on an Pentium-4 3.4 GHz

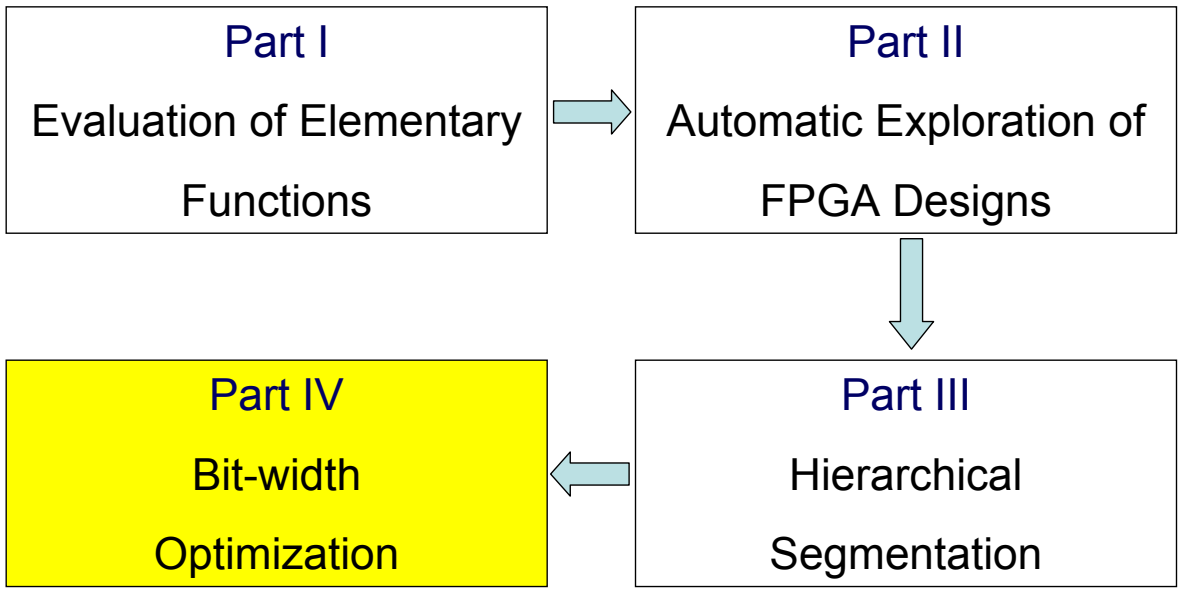
Precision [bits]	Segmentation [s]	Bit-Width Optimization [s]	VHDL Generation [s]	Total [s]
8	10	8	2	20
12	31	10	2	43
16	90	15	3	108
20	218	27	4	249
24	523	54	5	582

Area and Latency

- Degree-2 approximations to f_1 and f_4



Upper part: address decoding
Lower part: polynomial evaluation



4. Bit-Width Optimization

4(a) MiniBit: design tool based on affine arithmetic

1. Introduction
2. Overview and design flow
3. Range analysis and precision analysis
4. Case studies
5. Summary

4(b) Bitsize: design tool based on automatic differentiation

1. Introduction
2. Precision analysis: automatic differentiation
3. Case studies: ray tracing, function approx., FIR filtering
4. Summary

Bit-width Optimization (a): MiniBit

An Approach based on Affine Arithmetic

4(a).1 Introduction

- Challenge: determine correct number of bits for each signal
- Aim: find minimal bit-widths to all signals that overcome
  - excessive bit-width: waste hardware resources
  - insufficient bit-width: overflow + violate precision needs

Previous Work

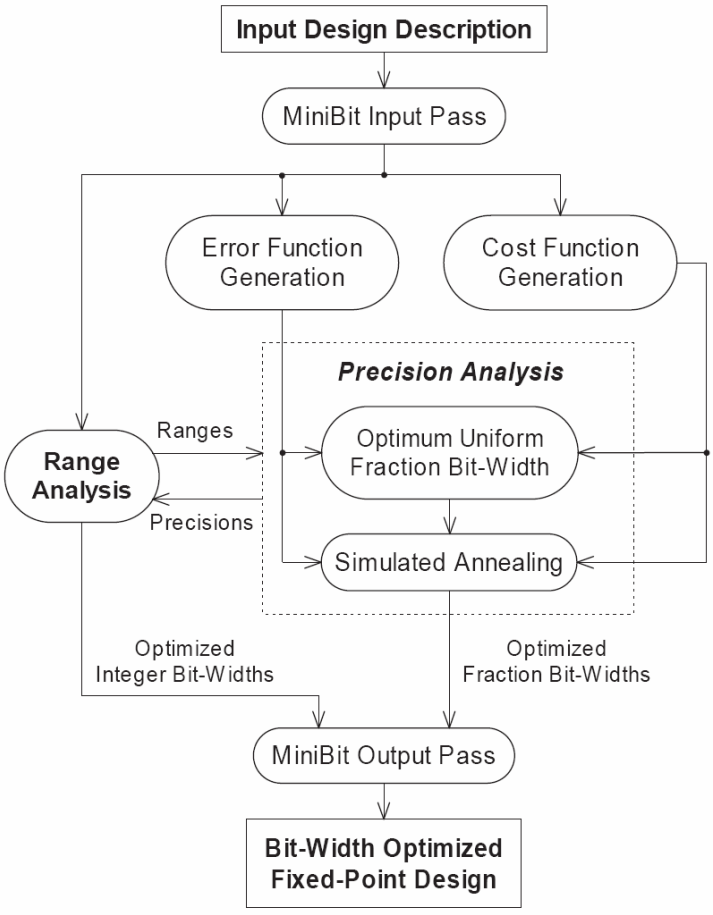
- Dynamic analysis
  - large set of stimuli inputs to evaluate bit-widths
  - long analysis times
- Static analysis
  - only characteristics of the input signals
  - more conservative estimates: worst case
  - faster analysis times
- Commonly used
  - dynamic method with a uniform bit-width
  - uniform bit-width gradually adjusted until all variables meet precision requirement

4(a).2 Overview

- Static bit-width optimization:
  for fixed-point designs via affine arithmetic (AA)
- Range and precision analysis:
  integer and fraction part determination
- Analytical approach:
  overflow protection and maximum error bounds
- Reduction in area and latency:
  up to 20% and 12% over optimum uniform fractional bit-width on Xilinx Virtex-4 FPGA

Design Flow

- Automated framework:
  generate bit-width optimized fixed-point hardware designs
- Separate range and precision analysis:
  use AA
- Static analysis:
  in contrast to dynamic methods, only need input signal characteristics



Affine Arithmetic (AA)

- Unlike interval arithmetic (IA), AA exploits correlations between signals
- Signal x over a range $[x_{min}, x_{max}]$ expressed as:

$x_0 = (x_{max} + x_{min}) / 2$

$x_1 = (x_{max} - x_{min}) / 2$

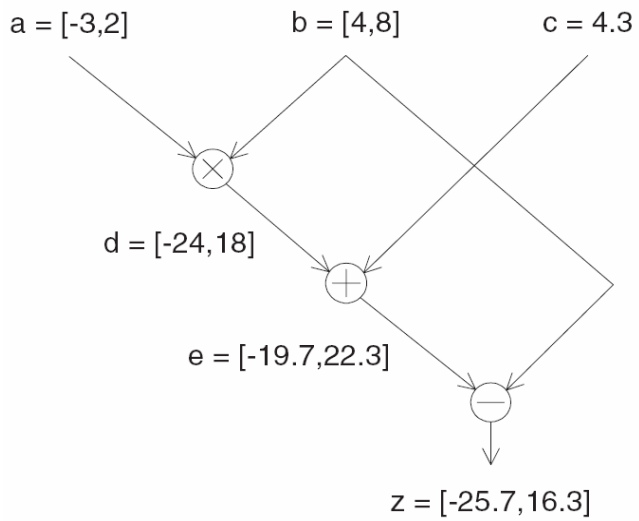
$$\hat{x} = x_0 + x_1\varepsilon_1 \qquad \text{where } \varepsilon = [-1, 1]$$

- e.g. addition / subtraction in AA form:

$$\hat{x} \pm \hat{y} \quad = \quad (x_0 \pm y_0) + \sum_{i=1}^{n} (x_i \pm y_i)\varepsilon_i$$

4(a).3 Range Analysis

Example Circuit



AA Range Models

$$\hat{a} = -0.5 + 2.5\varepsilon_1$$
$$\hat{c} = 4.3$$
$$\hat{e} = 1.3 + 15\varepsilon_1 - 1\varepsilon_2 + 5\varepsilon_3$$
$$\hat{b} = 6 + 2\varepsilon_2$$
$$\hat{d} = -3 + 15\varepsilon_1 - 1\varepsilon_2 + 5\varepsilon_3$$
$$\hat{z} = -4.7 + 15\varepsilon_1 - 7\varepsilon_2 + 5\varepsilon_3$$

$$IB_x = \begin{cases} \lceil \log_2(d) \rceil & \text{if } |d| > 1 \\ 1 & \text{if } |d| \leq 1 \end{cases}$$

where $d = \max(|x_{min}|, |x_{max}|)$

Precision Analysis

- two quantization methods
  - truncation: maximum error of 2^{-FB}
  - round-to-nearest: maximum error of 2^{-FB-1}
- round-to-nearest is used throughput for simplicity

$$\tilde{x} = x + 2^{-FB_{\tilde{x}}-1}\varepsilon \quad \text{where } \varepsilon = [-1, 1]$$

$$E_{\tilde{x}} = 2^{-FB_{\tilde{x}}-1}$$

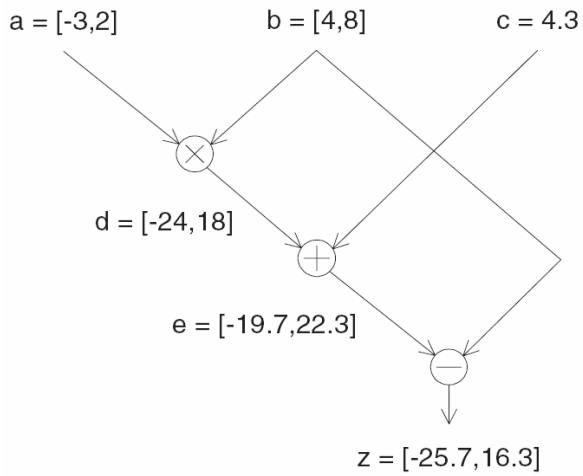
- e.g. addition / subtraction error model:

$$\begin{aligned} \tilde{z} &= \tilde{x} \pm \tilde{y} = x \pm y + E_{\tilde{x}} \pm E_{\tilde{y}} + 2^{-FB_{\tilde{z}}-1}\varepsilon_3 \\ \Rightarrow E_{\tilde{z}} &= E_{\tilde{x}} + E_{\tilde{y}} + 2^{-FB_{\tilde{z}}-1}\varepsilon_3 \end{aligned}$$

FB: Fraction Bit-width

Example: Precision Analysis

Example Circuit

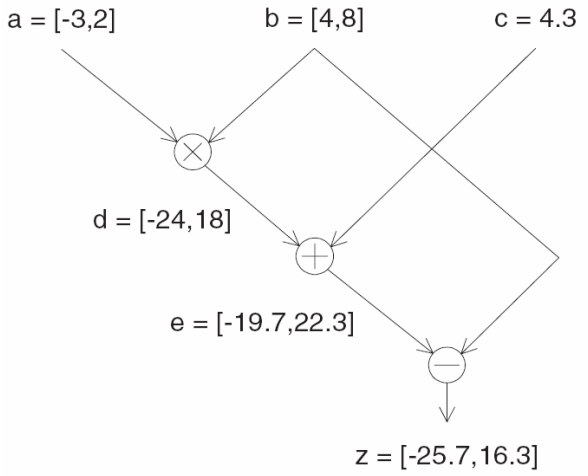


Error Expressions

$$\begin{aligned} E_{\tilde{a}} &= 2^{-FB_{\tilde{a}}-1}\varepsilon_1 \\ E_{\tilde{b}} &= 2^{-FB_{\tilde{b}}-1}\varepsilon_2 \\ E_{\tilde{c}} &= 2^{-FB_{\tilde{c}}-1}\varepsilon_3 \\ E_{\tilde{d}} &= aE_{\tilde{b}} + bE_{\tilde{a}} + E_{\tilde{a}}E_{\tilde{b}} + 2^{-FB_{\tilde{d}}-1}\varepsilon_4 \\ E_{\tilde{e}} &= E_{\tilde{d}} + E_{\tilde{c}} + 2^{-FB_{\tilde{e}}-1}\varepsilon_6 \\ E_{\tilde{z}} &= E_{\tilde{e}} - E_{\tilde{b}} + 2^{-FB_{\tilde{z}}-1}\varepsilon_7 \end{aligned}$$

FB: Fraction Bit-width

Worst Case Error



$$\max(E_{\tilde{z}}) = 2^{-FB_{\tilde{b}}} + 2^{-FB_{\tilde{a}}+2} + 2^{-FB_{\tilde{a}}-FB_{\tilde{b}}-2} + 2^{-FB_{\tilde{d}}-1} + 2^{-FB_{\tilde{c}}-1} + 2^{-FB_{\tilde{e}}-1} + 2^{-FB_{\tilde{z}}-1}$$

Optimal Uniform Fraction Bit-Width

- Initial state for Adaptive Simulated Annealing: substitute FBs with a single variable UFB

$$2^{-FB_{\tilde{z}}-1} \geq 2^{-\underline{UFB}} + 2^{-\underline{UFB}+2} + 2^{-2\underline{UFB}-2} + 3(2^{-\underline{UFB}-1})$$

- if FB_z = 16 bits, i.e. $\max(FB_z) \leq 2^{-16}$
  solve equation for minimum value of UFB
  gives UFB = 20 bits:
  analytical solution for uniform bit-width selection

FB: Fraction Bit-width, UFB: Uniform Fraction Bit-width

4(a).4 Case Studies

- Compiled using A Stream Complier (ASC), an FPGA complier based on C++
- Synthesized with ASC, placed-and-routed with Xilinx ISE 6.3 on a Virtex-4 XC4VLX100-11 FPGA
- Implementation: combinational using slices only
- Compare accuracy: double-precision floating point
- Area models: *(IB: Integer Bit-width, FB: Fraction Bit-width)*
  - x + y: $\max(IB_x+FB_x, IB_y+FB_y)$
  - x × y: $(IB_x+FB_x)\times(IB_y+FB_y)$

Case Studies on Virtex-4 FPGAs

Case Studies		UFB/MFB Comparisons							
Application	Prec	Area [slices]				Latency [ns]			
	[bits]	UFB	MFB	Diff	Impr [%]	UFB	MFB	Diff	Impr [%]
Degree 4	8	803	723	80	9.96	114.00	105.39	8.61	7.55
Polynomial	16	1921	1797	124	6.45	168.55	151.81	16.74	9.93
RGB to YCbCr	8	1165	1132	33	2.83	37.47	36.95	0.52	1.39
	16	1641	1602	39	2.38	50.26	48.83	1.43	2.85
2 × 2 Matrix	8	1896	1799	97	5.12	44.20	42.73	1.47	3.33
Multiplication	16	4240	4072	168	3.96	59.22	56.14	3.08	5.20
B-Splines	8	1189	952	237	19.93	88.39	78.58	9.81	11.10
	16	2652	2165	487	18.36	130.11	114.03	16.08	12.36
8 × 8 DCT	8	5368	5217	151	2.81	54.83	50.73	4.10	7.48
	16	7320	7167	153	2.09	66.39	59.42	6.97	10.50

UFB: Uniform Fraction Bit-width

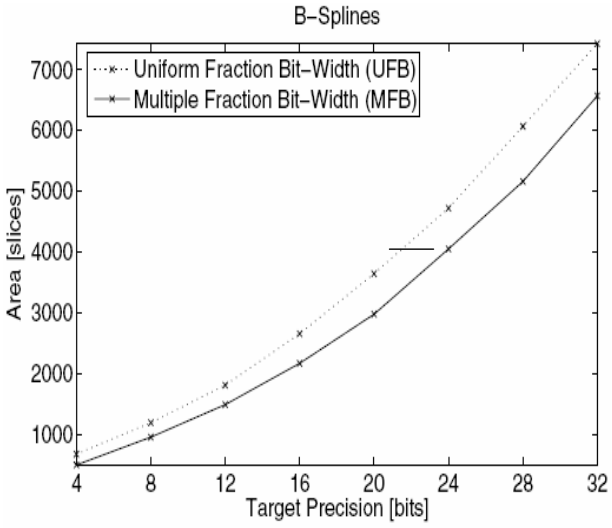
MFB: Multiple Fraction Bit-width (enabled by MiniBit approach)

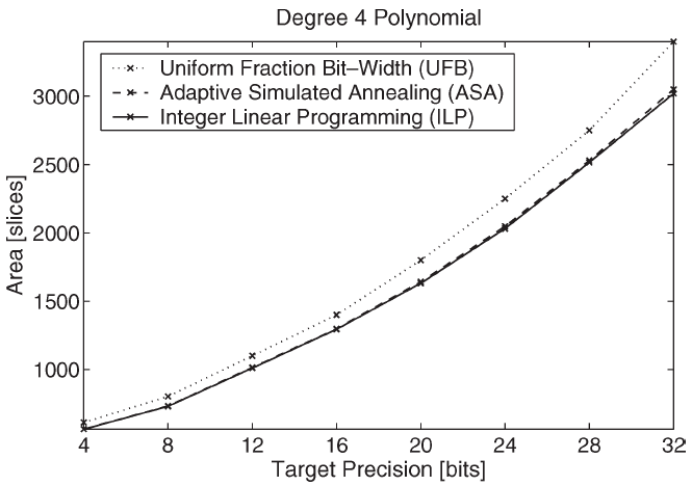
Comparing Range Analysis Methods

Case Study	Area [slices]			Latency [ns]		
	Simulation	Affine Arithmetic	Interval Arithmetic	Simulation	Affine Arithmetic	Interval Arithmetic
Degree 4 Poly	701	723	732	103.21	105.39	107.37
RGB to YCbCr	1109	1132	1143	35.17	36.95	38.82
2 × 2 Matrix Mult	1747	1799	1838	40.71	42.73	43.68
B-Splines	937	952	978	76.01	78.58	89.86
8 × 8 DCT	5103	5217	5312	47.61	50.73	51.30

- Target precision: 8 bits
- Simulation
  - Input: random samples
  - Smallest area and latency, but exponential run time
- Affine Arithmetic better than Interval Arithmetic
  - AA can exploit correlations between signals

Technology Trends





- Increase target precision
  - larger gain of MFB over UFB
  - ASA results still within 1% of ILP results (global optimum)

4(a).5 Summary

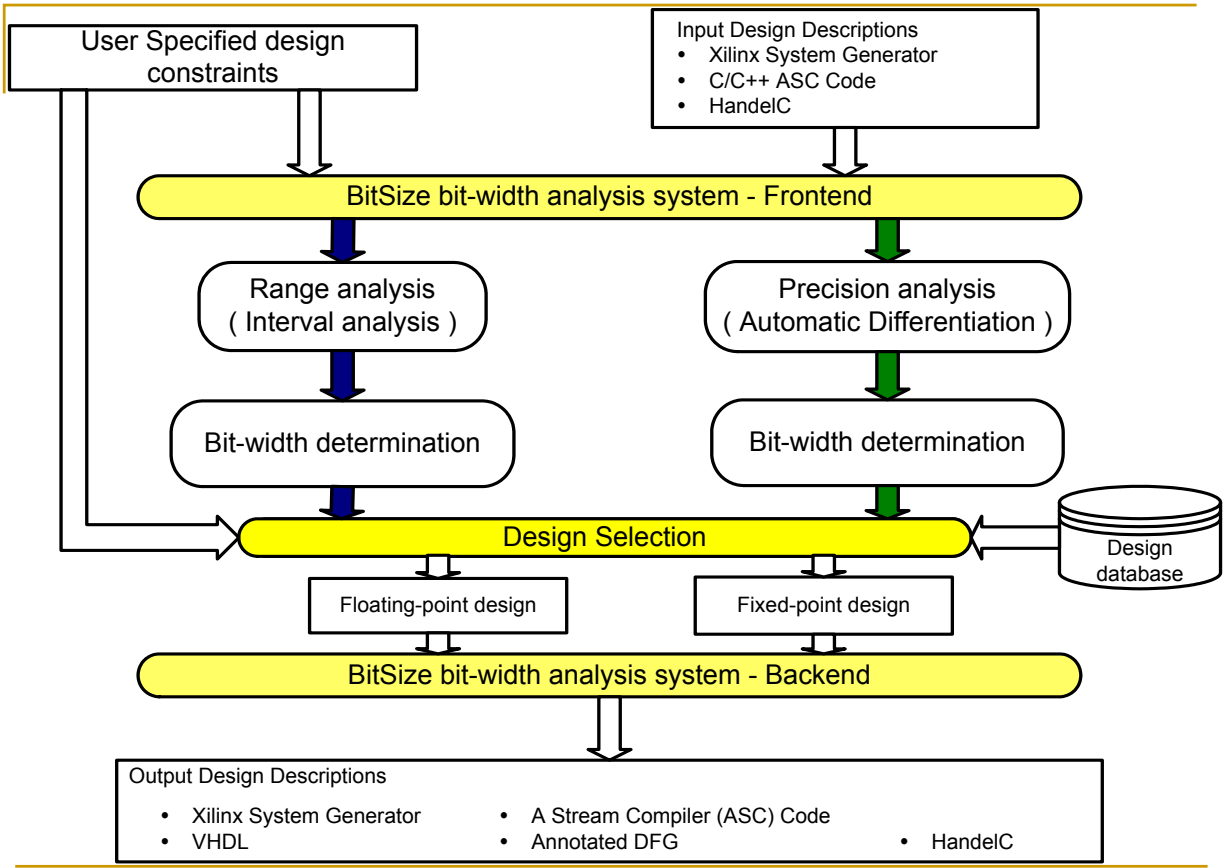
- Analytical models based on affine arithmetic
  - guarantee accuracy
  - better than interval arithmetic, close to optimum
- Current and future work
  - extend scope of approach, e.g. floating-point designs
  - replace ASA by other optimizations
    e.g. genetic algorithms

Bit-width Optimization (b): BitSize

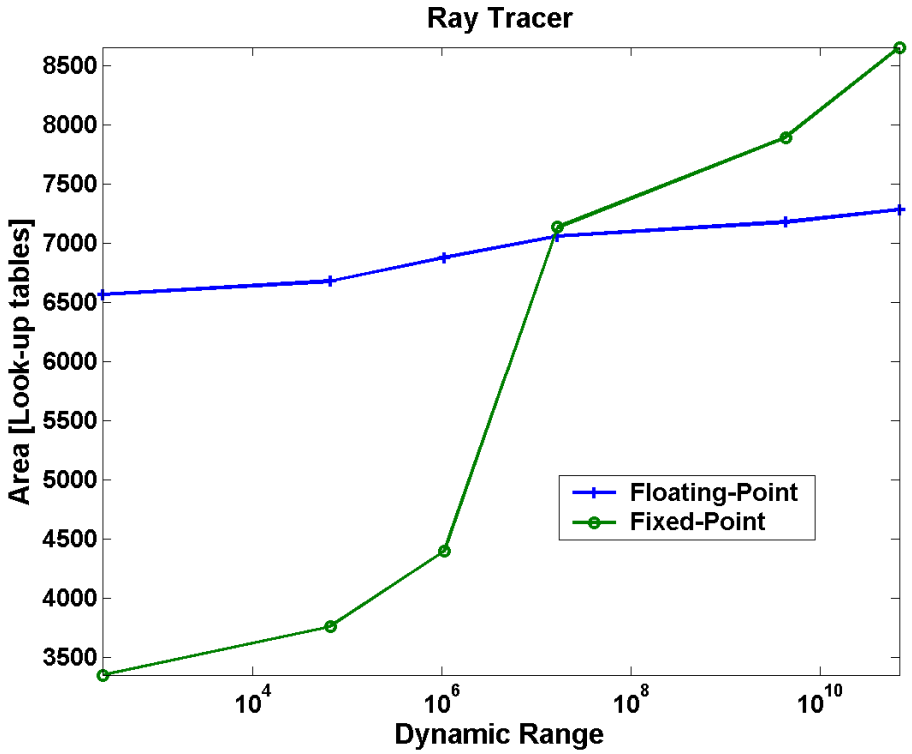
An Approach Based on Automatic Differentiation

4(b).1 Introduction

- Automate selection of number representation
  - fixed-point versus floating-point
- Design space exploration
  - area, latency, throughput, power, accuracy tradeoffs
  - optimised bit-width: range, precision
- Custom automatic differentiation technique, minimising data structure traversal
- Framework supports multiple source and target formats
  - C/C++(ASC), Xilinx System Generator, HandelC



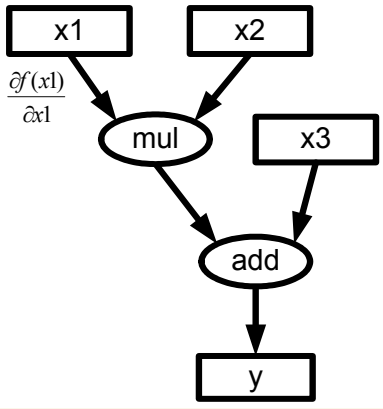
Range Analysis: Interval Arithmetic

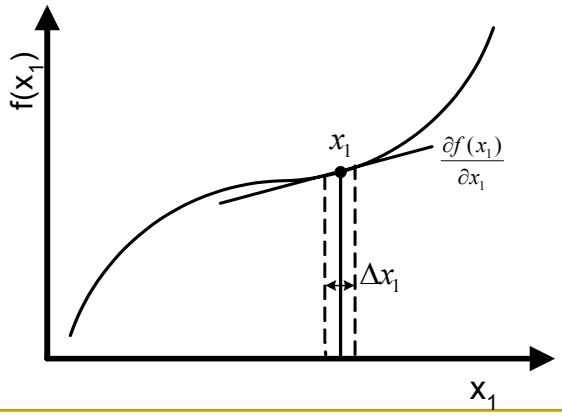


Precision Analysis: Sensitivity Approach

- **Sensitivity** s, of output y is the change in y to a change in x_1:

$$\Delta y = s(x_1)\Delta x_1 \approx f'(x_1)\partial x_1$$





4(b).2 Precision Analysis: Automatic Differentiation

- *Automatic* method to differentiate a program given: a program computes the function $y = f(\vec{x})$
- Automatic differentiation (AD) provides a gradient vector

  for the program variables: $\vec{x} = [x_1, x_2, \ldots, x_n]$
  and output: y

  gradient vector: $s(\vec{x}) \approx \left[\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, \cdots, \frac{\partial y}{\partial x_n}\right]$

From Gradient Vector to Bit-width

- For program with *n* variables (data-flow graph with *n* nodes):
  - extend definition of sensitivity:

$$\Delta y \geq s(\vec{x})\Delta\vec{x} \approx \Delta x_1 \frac{\partial Y}{\partial x_1} + \cdots + \Delta x_n \frac{\partial Y}{\partial x_n}$$

- Given Δy, AD calculates $s(\vec{x})$ and our tool determines $\Delta\vec{x} = [\Delta x_1, \Delta x_2, \ldots, \Delta x_n]$
- Only a single evaluation of the function is required, unlike *simulation-only* methods

Determine Bit-width: Fixed-point

- Variable x_i in fixed-point :

$p_{k-1} \cdots p_2\ p_1\ p_0$	$q_0\ q_1\ q_2 \cdots q_{l-1}$

- Precision is the *fraction* bit-width:

$$l \geq \lceil \log_2(|\Delta x_i|) \rceil + 1$$

- Range is the *integer* bit-width:

$$k \geq \lceil \log_2(|\max(x_i)/\min(x_i)|) \rceil$$

Determine Bit-width: Floating-point

- Variable x_i in floating-point :

S	$a_0\ a_1\ a_2 \cdots a_{m-1}$	$b_{e-1} \cdots b_2\ b_1\ b_0$

- Precision is the *mantissa* bit-width:

$$m \geq E_i - \lceil \log_2(|\Delta x_i|) \rceil + 1$$

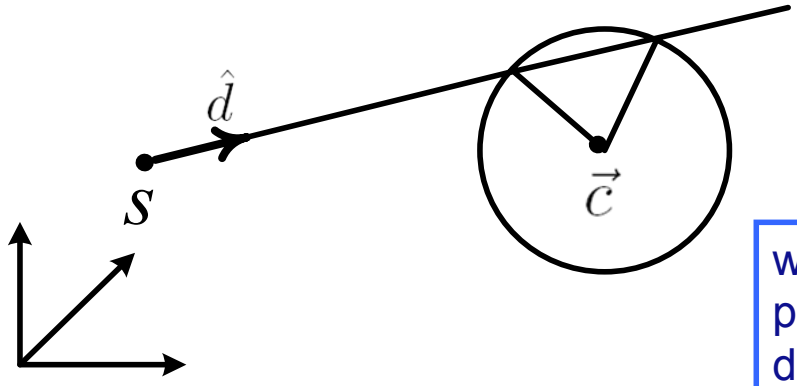
  - where E_i is the exponent

- Range is the *exponent* bit-width:

$$e \geq \lceil log_2(|\max(E_i) - \min(E_i)|) \rceil$$

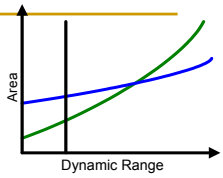
4(b).3 Case Studies

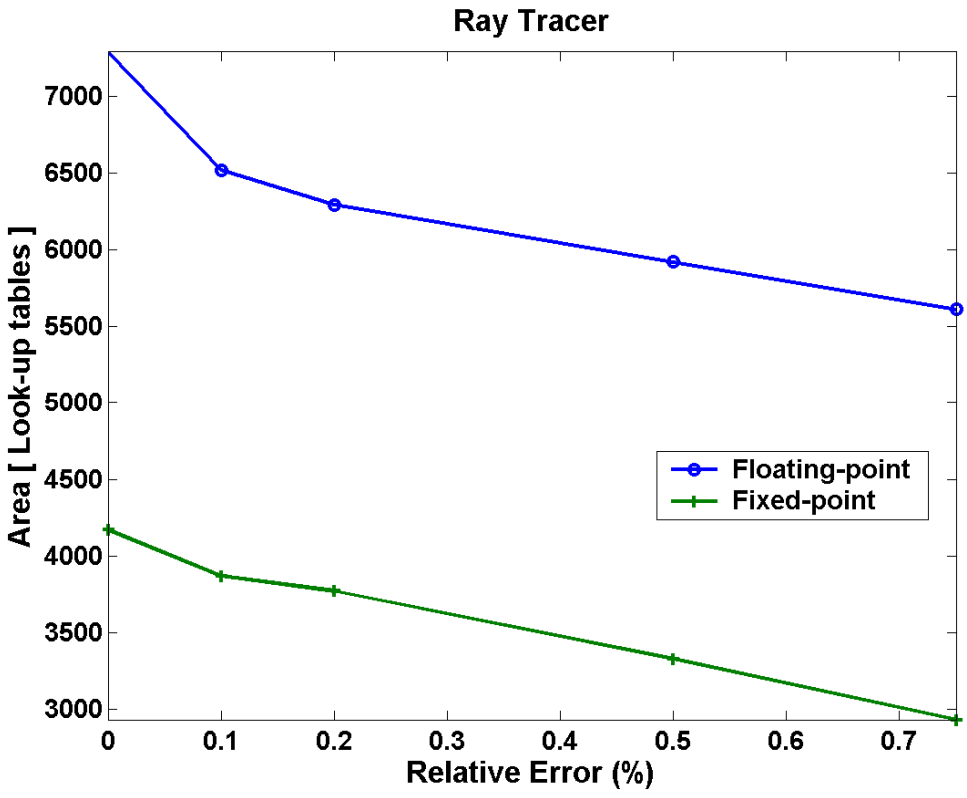
- Ray tracing: solving the determinant equation for ray intersection



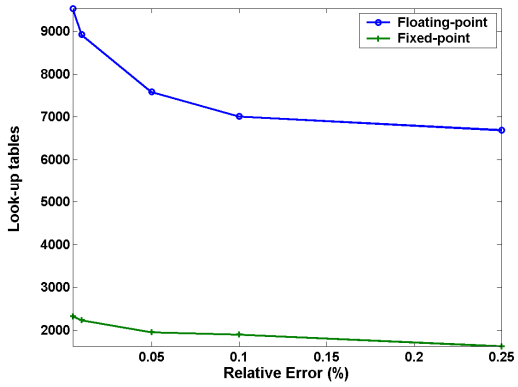
when a ray from point *S*, with direction vector *d* intersects the sphere with centre *C* and radius *R*

Ray Tracing – Area vs. Precision

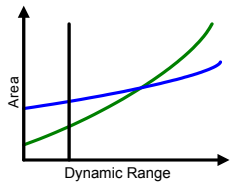




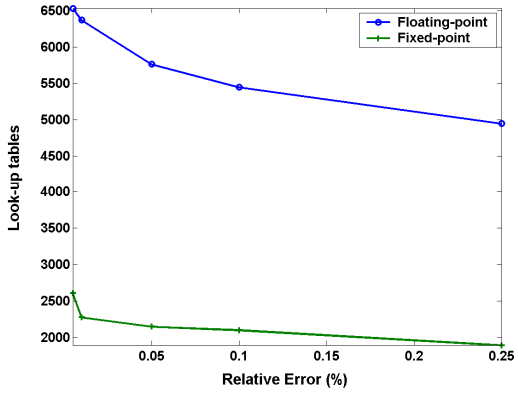
Function Approximation - Area vs. Precision



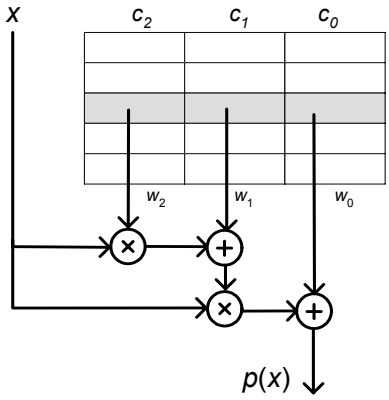
$ln(x)$



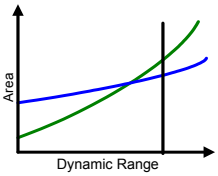
$$p(x) = (c_2 x + c_1)x + c_0$$

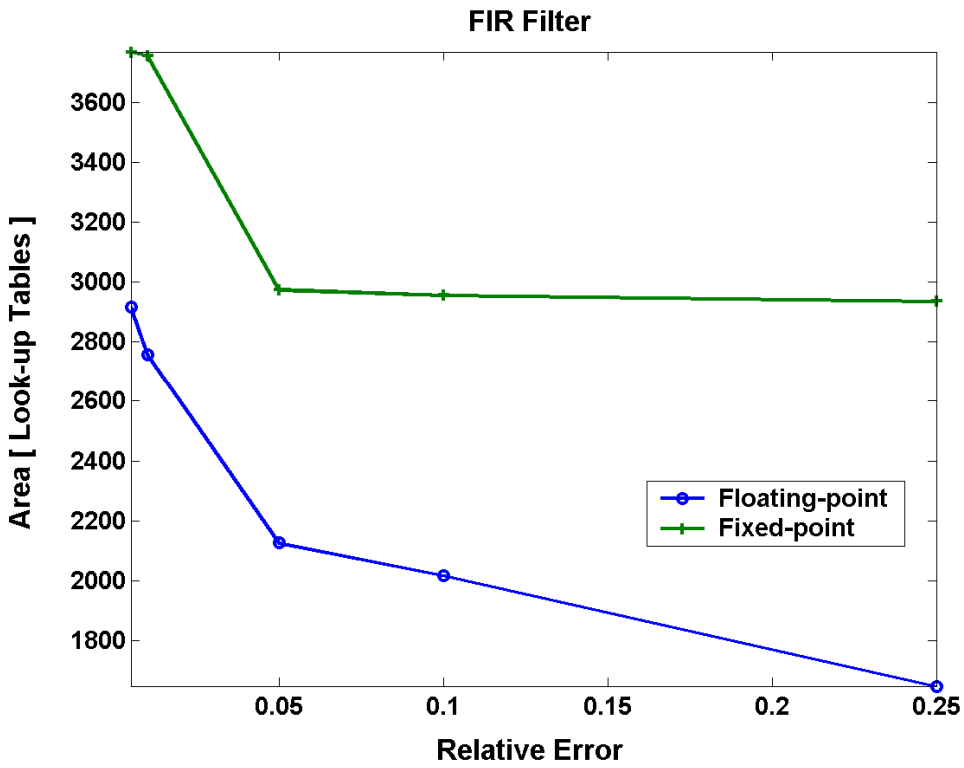


$xln(x)$



FIR Filtering - Area vs. Precision





4(b).4 Summary

- Unified dynamic analysis framework for
  - fixed point designs
  - floating point designs
- Small reduction in accuracy
  - often results in large improvement in area
- Current and future work
  - support other number representations
  - support designs with multiple number representations
  - combine static and dynamic analysis methods
  - support hardware/software codesign
  - support designs involving dynamic reconfiguration
  - study trade-offs in accuracy, speed, area, power

Further Reading

- Evaluation of Elementary Functions
  - O. Mencer and W. Luk, *Parameterized high throughput function evaluation for FPGAs*, Journal of VLSI Signal Processing, vol. 36, no. 1, pp. 17-25, January 2004.
- Automatic Exploration of FPGA Designs
  - D. Lee, A. Abdul Gaffar, O. Mencer and W. Luk, *Optimizing hardware function evaluation*, IEEE Transactions on Computers, vol. 54, no. 12, pp. 1520-1531, December 2005.
- Hierarchical Segmentation
  - D. Lee, W. Luk, J. Villasenor and P.Y.K. Cheung, *Hierarchical segmentation schemes for function evaluation*, Proc. IEEE Int. Conf. on Field Programmable Technology, pp.92-99, 2003.
- Bit-width Optimization
  - D. Lee, A. Abdul Gaffar, R.C.C. Cheung, O. Mencer, W. Luk and G.A. Constantinides, *Accuracy-guaranteed bit-width optimization*, IEEE Transactions on Computer-Aided Design, vol. 25, no. 10, pp. 1990-2000, October 2006.
  - A. Abdul Gaffar, O. Mencer, W. Luk and P.Y.K. Cheung, *Unifying bit-width optimisation for fixed-point and floating-point designs*, Proc. IEEE Symp. on Field Programmable Custom Computing Machines, pp. 79-88, 2004.