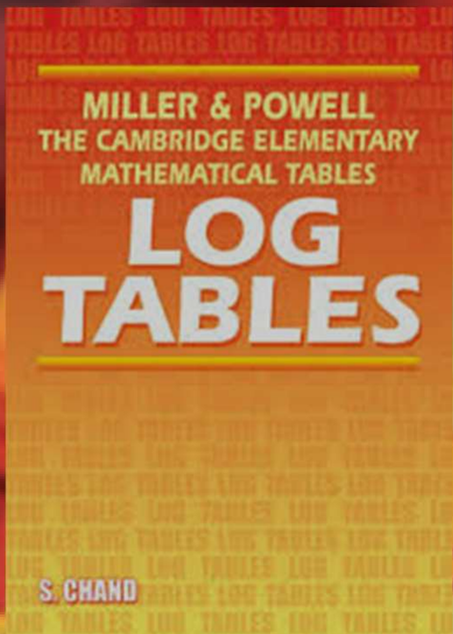


A Case for Table-Based Approximate Computing

Behrooz Parhami

University of California,
Santa Barbara

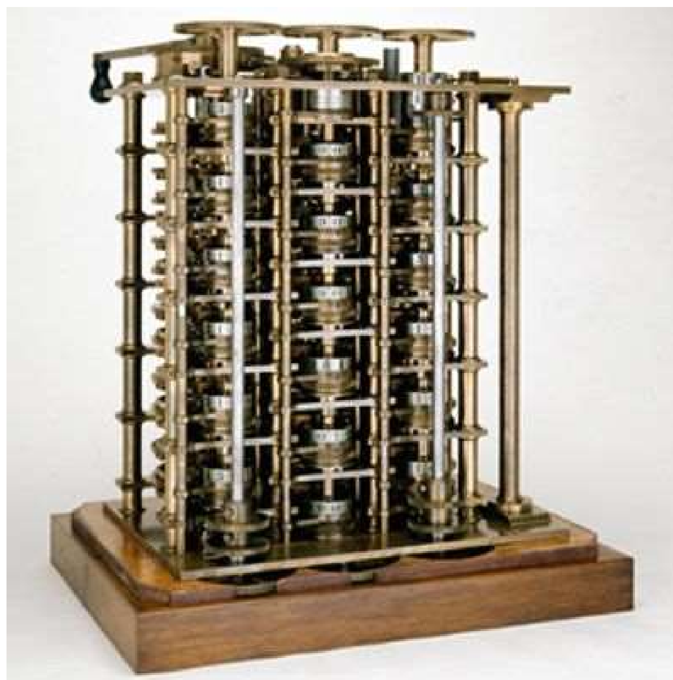
About This Presentation

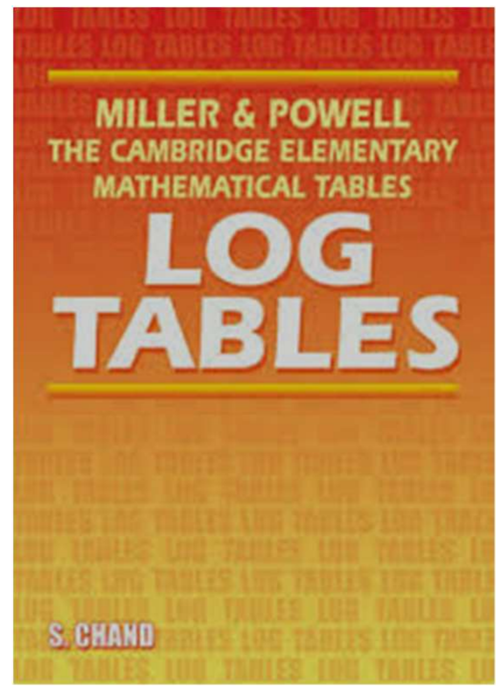
This slide show was first developed in fall of 2018 for a November 2018 talk at IEEE IEMCON (Information Technology, Electronics & Mobile Communication Conf.), University of British Columbia, Vancouver, BC, Canada.

Edition	Released	Revised	Revised	Revised
First	Fall 2018			

Tabular Computing: A History

Ancient tables: Manually computed
Charles Babbage: Polynomial approximation of functions
Math handbooks: My generation used them
Modern look-up tables: Speed-up; Caching; Seed value





Example of Table-Based Computation

Compute $\log_{10} 35.419$

Table: log of values in [1.00, 9.99], in increments of 0.01]

Pre-scaling: $\log_{10} 35.419 = 1 + \log_{10} 3.5419$

Table access: $\log_{10} 3.54 = 0.549\ 003$

$\log_{10} 3.55 = 0.550\ 228$

Interpolation: $\log_{10} 3.5419 = \log_{10} 3.54 + \varepsilon$

$= 0.549\ 003 + (0.19)(0.550\ 228 - 0.549\ 003)$

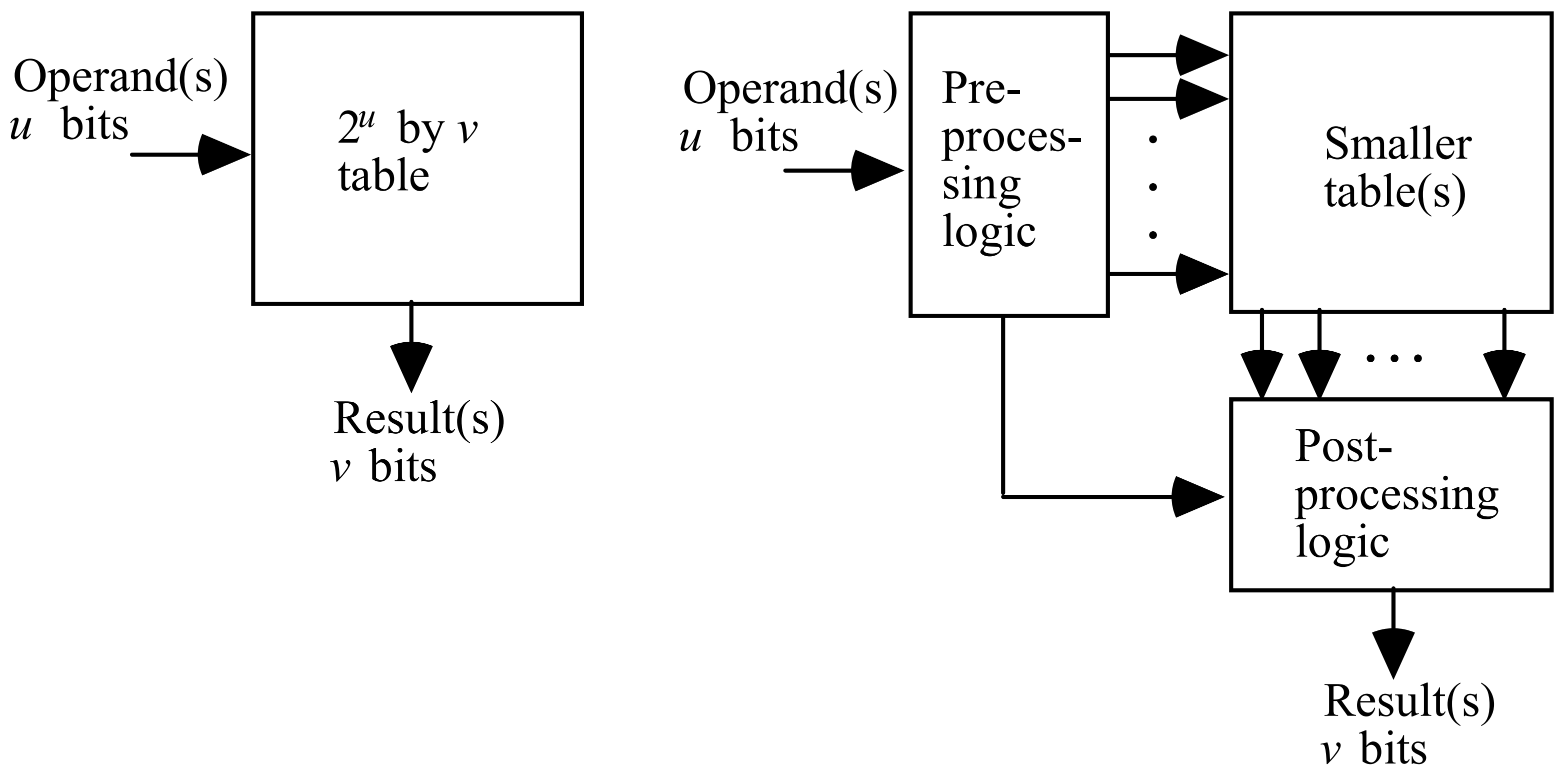
$= 0.549\ 236$

Final result: $\log_{10} 35.419 = 1 + 0.549\ 236 = 1.549\ 236$

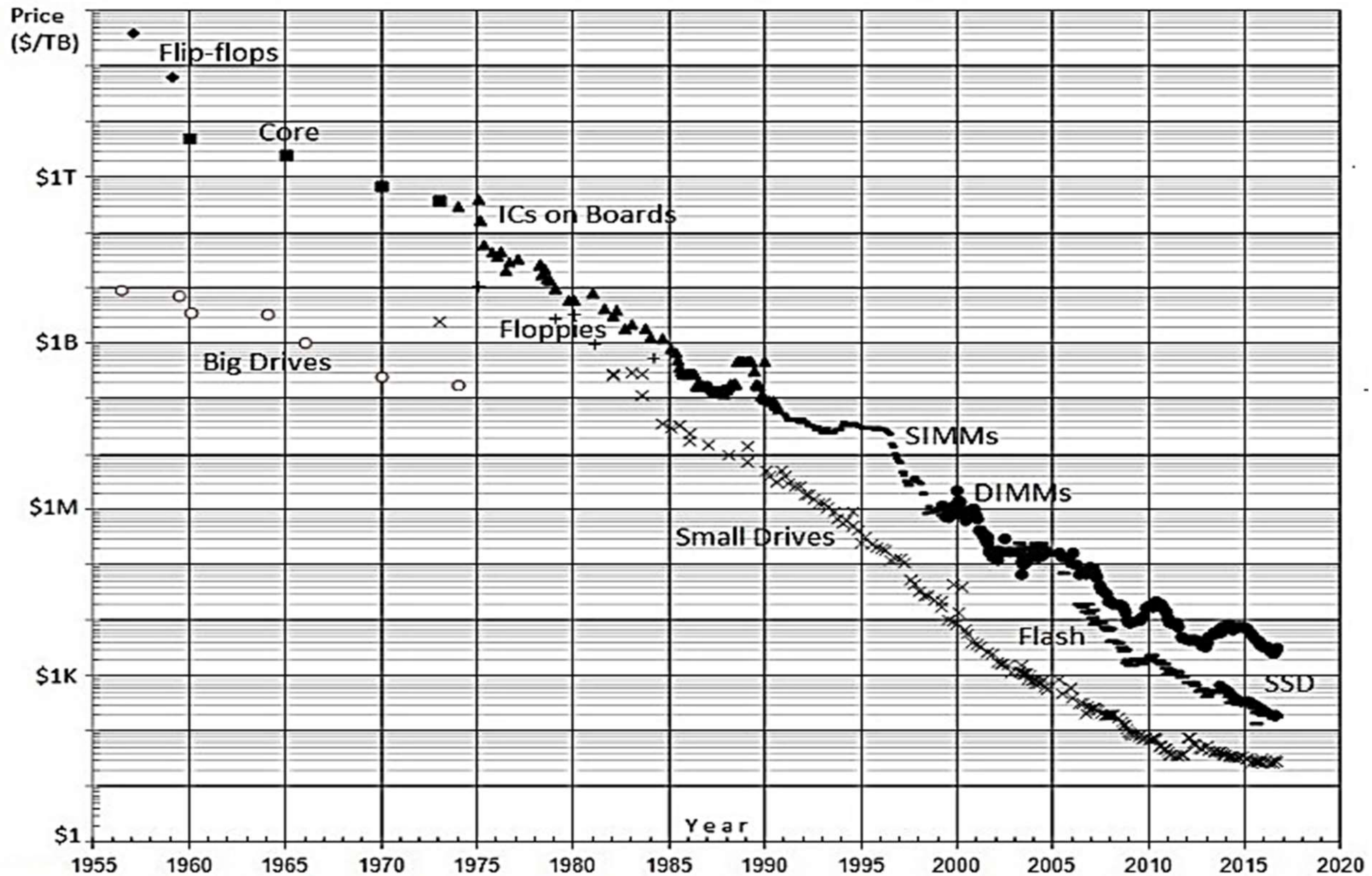
Direct and Indirect Table-Lookup

Direct lookup: Operands serve as address bits into table

Indirect lookup: Inputs pre-processed; output post-processed



Memory Cost Reduction Trends



Example for Table Size Reduction

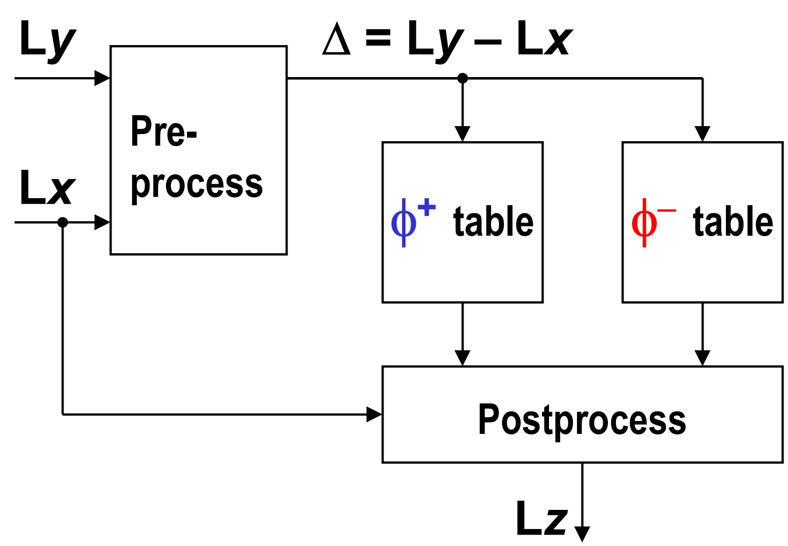
Strategy: Reduce the table size by using an auxiliary unary function to evaluate a desired binary function

Addition/subtraction in a logarithmic number system; i.e., finding Lz = log($x \pm y$), given Lx and Ly

Solution: Let Δ = Ly – Lx

$$
\begin{aligned}
\mathrm{L}z &= \log(x \pm y) \\
&= \log(x(1 \pm y/x)) \\
&= \log x + \log(1 \pm y/x) \\
&= \mathrm{L}x + \log(1 \pm \log^{-1}\Delta)
\end{aligned}
$$

$\mathrm{L}x + \phi^{+}(\Delta)$ $\qquad$ $\mathrm{L}x + \phi^{-}(\Delta)$

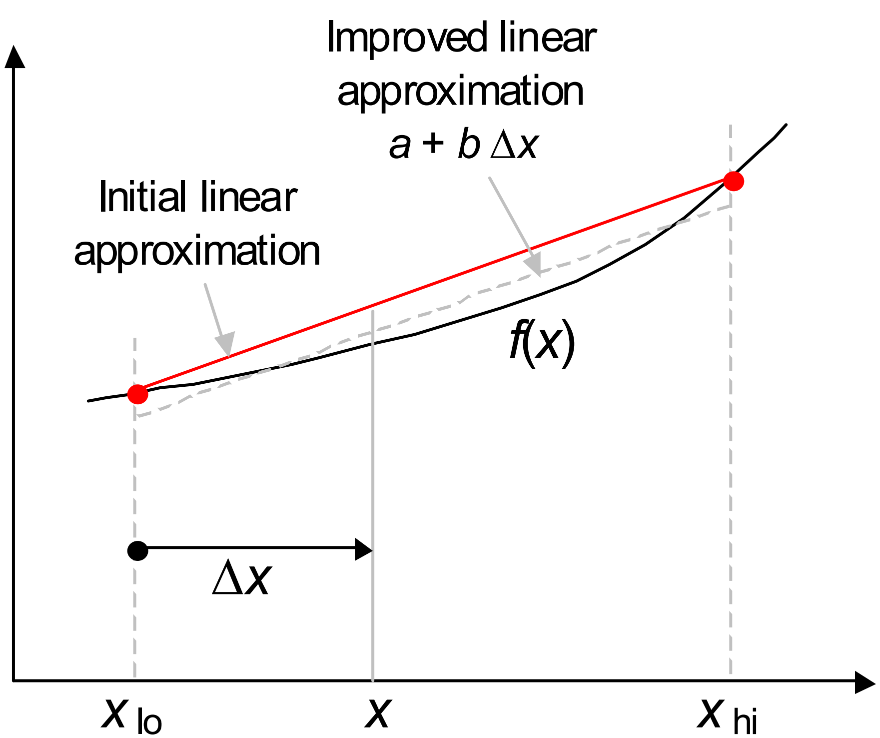


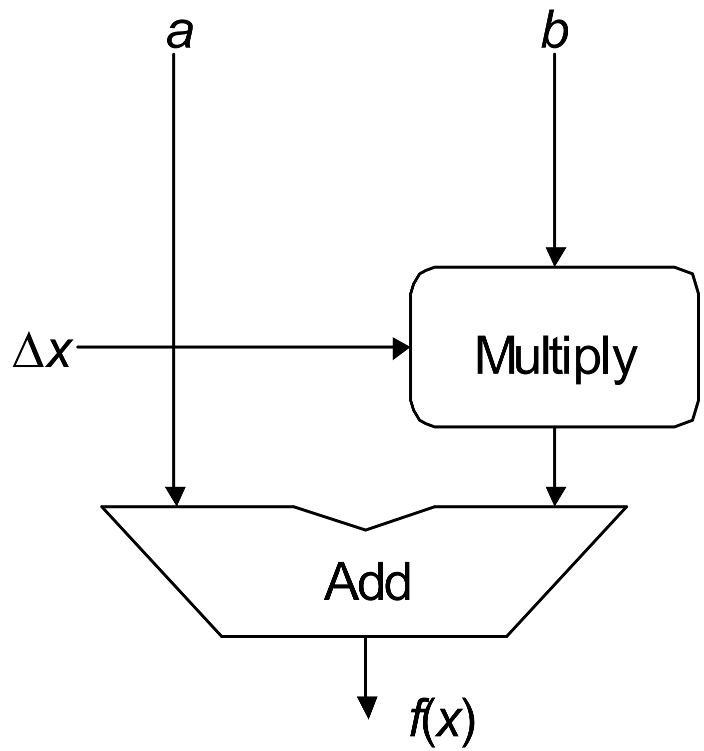
Interpolating Memory Unit

Linear interpolation: Computing $f(x)$, $x \in [x_{lo}, x_{hi}]$, from $f(x_{lo})$ and $f(x_{hi})$

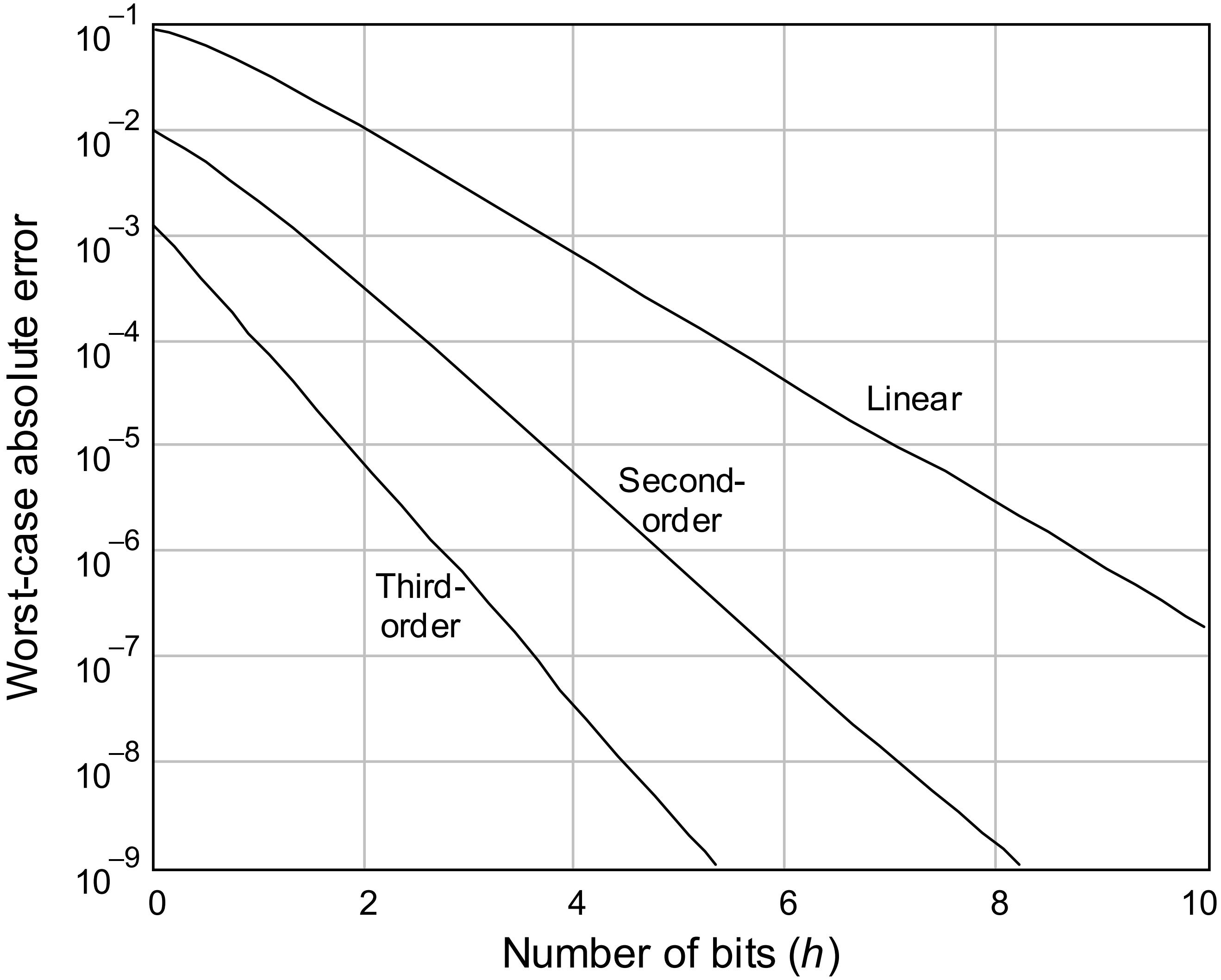
$$f(x) = f(x_{lo}) + \frac{x - x_{lo}}{x_{hi} - x_{lo}}[f(x_{hi}) - f(x_{lo})]$$

4 adds, 1 divide, 1 multiply
(2 adds) (1 shift)





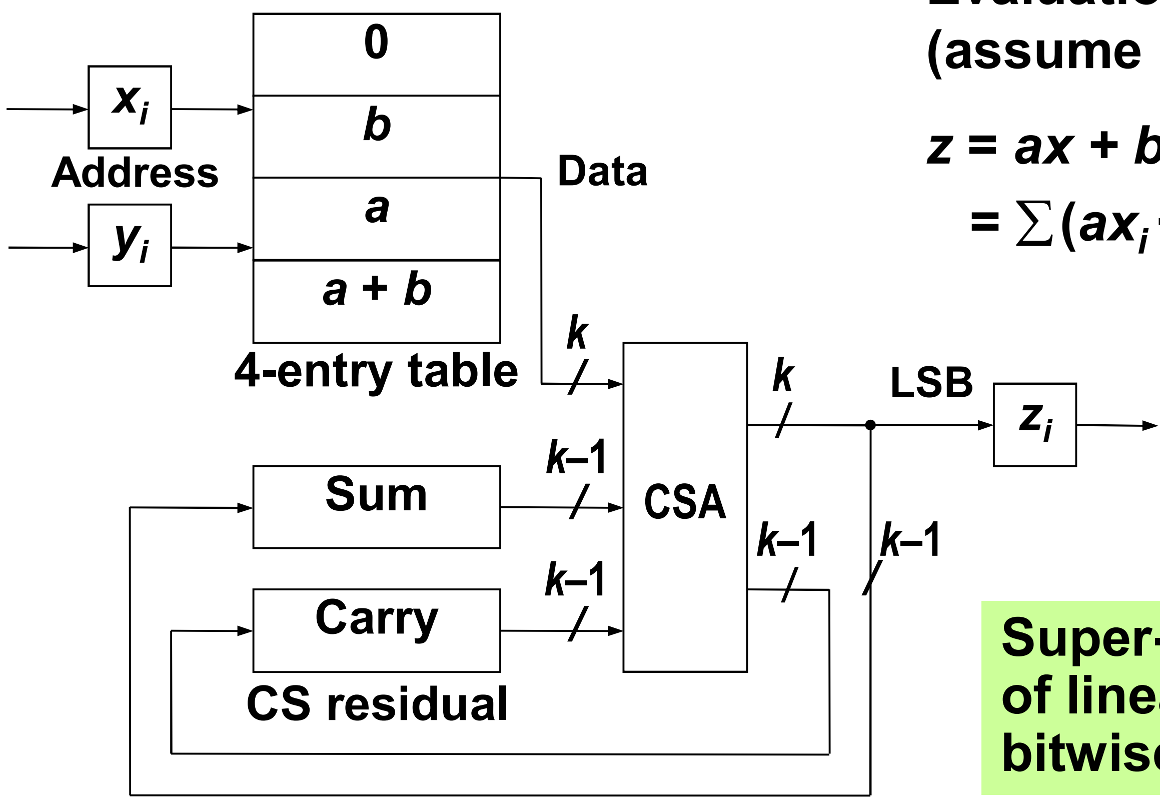
Trade-offs in Cost, Speed, and Accuracy



For the same target error, higher-order interpolation leads to smaller tables (2^h entries) but greater hardware complexity on the periphery

Tables in Bit-Serial Arithmetic

Distributed arithmetic for the evaluation of weighted sums and other linear expressions

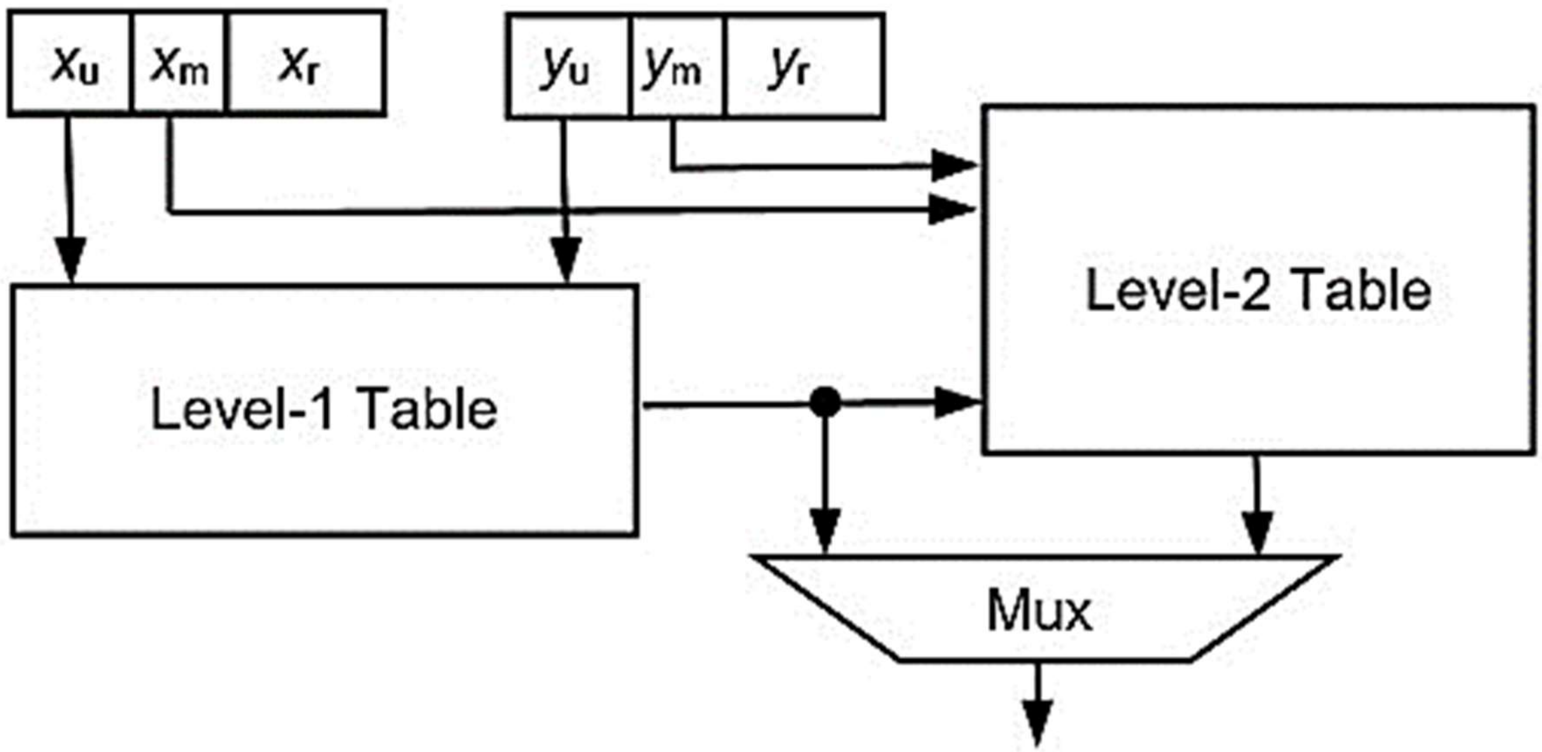


Evaluation of linear expressions (assume unsigned values)

$$z = ax + by = a \sum x_i 2^i + b \sum y_i 2^i$$
$$= \sum (ax_i + by_i)\, 2^i$$

Super-efficient computation of linear forms using only bitwise addition hardware

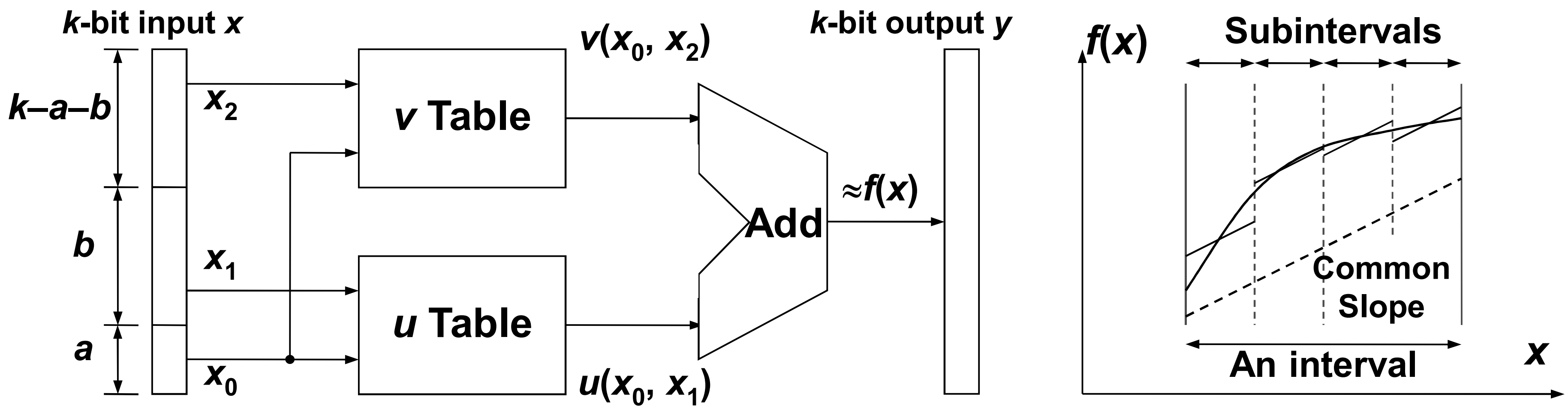
Two-Level Table for Approximate Sum



Level-1 table provides a rough approximation for the sum

Level-2 table refines the sum for a better approximation

Bipartite and Multipartite Lookup Tables



(a) Hardware realization

(b) Linear approximation

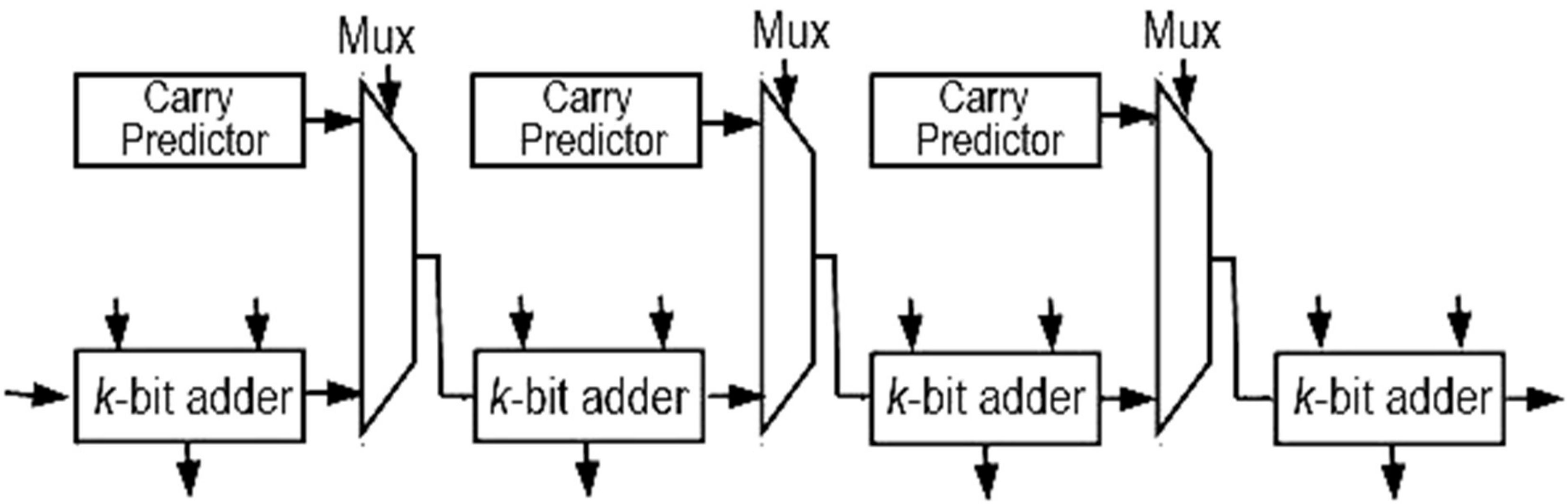
Divide the domain of interest into 2^a intervals, each of which is further divided into 2^b smaller subintervals

Bipartite tables: Main idea

The trick: Use linear interpolation with an initial value determined for each subinterval and a common slope for each larger interval

Total table size is $2^{a+b} + 2^{k-b}$, in lieu of 2^k; width of table entries has been ignored in this comparison

Approximate Computing Example

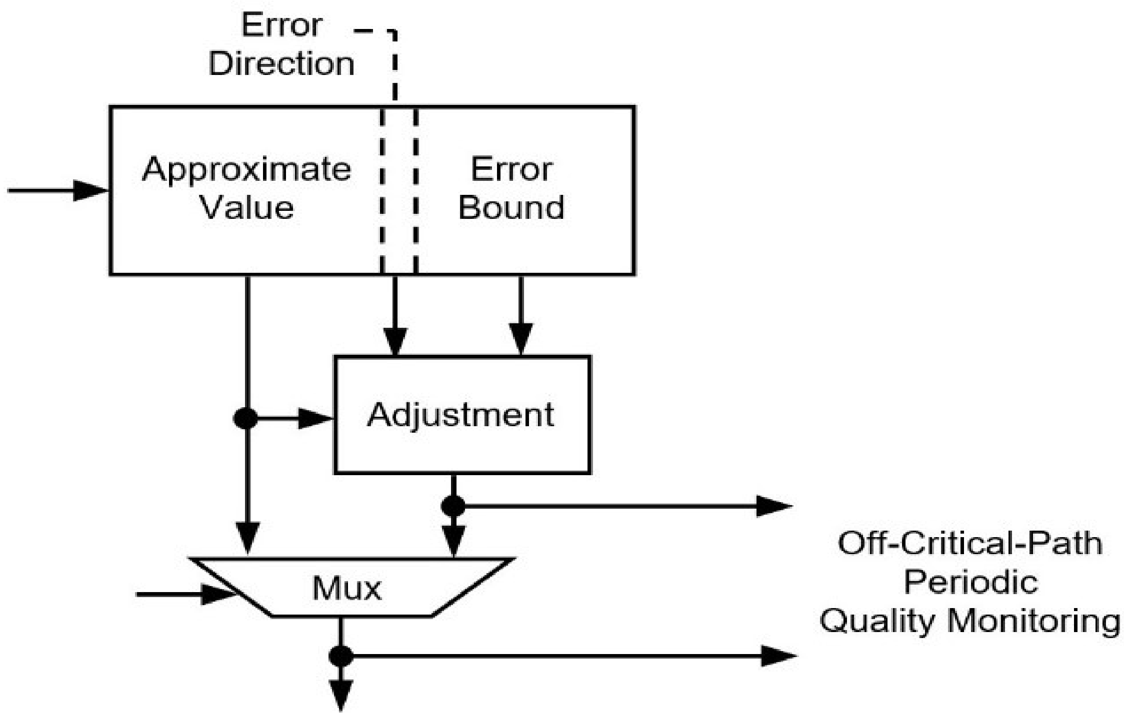


An approximate 4*k*-bit addition scheme

Carry predictor is correct most of the time, leading to addition time dictated by the shorter *k*-bit adders

The adder can also perform precise addition, if required

Adaptive Table-Based Computing



Approximate value is read out from the top table, which also supplies an error direction and an accurate error bound

The more precise value is compared with the approximate value off the critical path for periodic quality monitoring

Conclusions and Future Work

Too much precision is wasteful: Chip area and energy

Benefits of table-based approximate computing:

Tables are well-matched to approximate computing

Error direction and magnitude are knowable

Table-size reduction methods exist

Memories get bigger/cheaper

Future work and application-specific evaluations

Design of various kinds of neural networks

Mixed D/A representations and processing

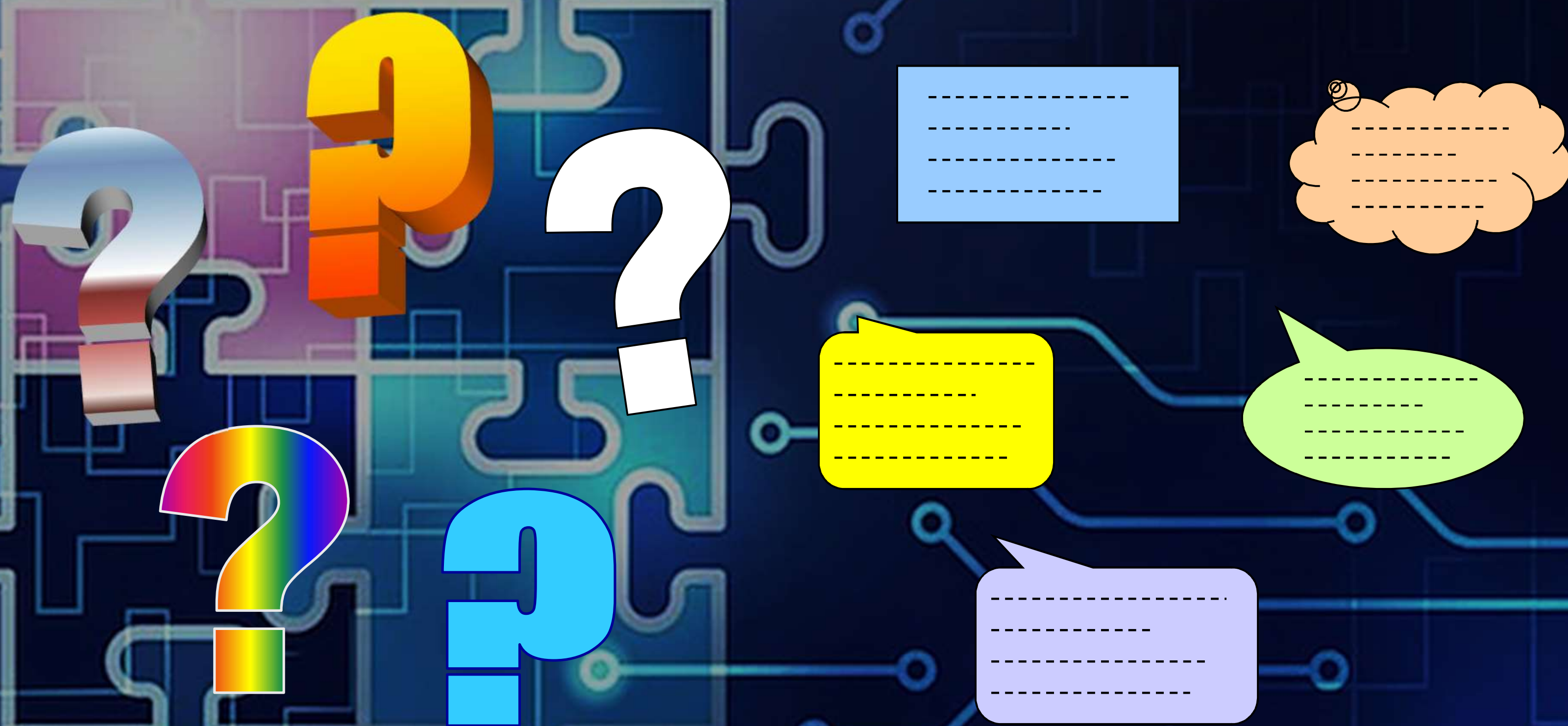
Bit-level optimization methods

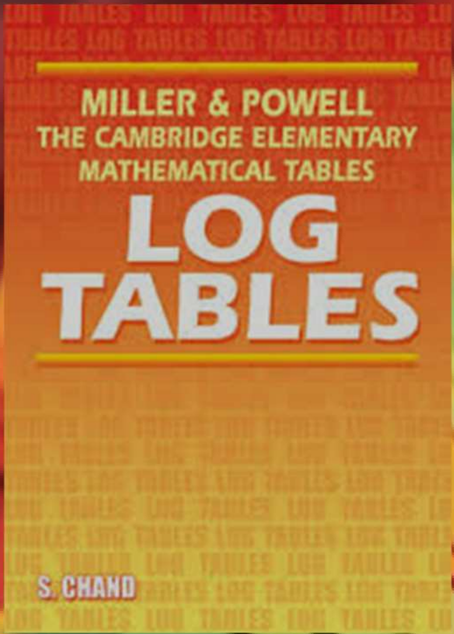
Iterative refinement strategies

Questions or Comments?

parhami@ece.ucsb.edu

http://www.ece.ucsb.edu/~parhami/





A Case for Table-Based Approximate Computing

Back-Up Slides

Behrooz Parhami
University of California,
Santa Barbara

Tables in Primary and Supporting Roles

Tables are used in two ways:

As main computing mechanism

In supporting role (e.g., as in initial estimate for division)

Boundary between two uses is fuzzy

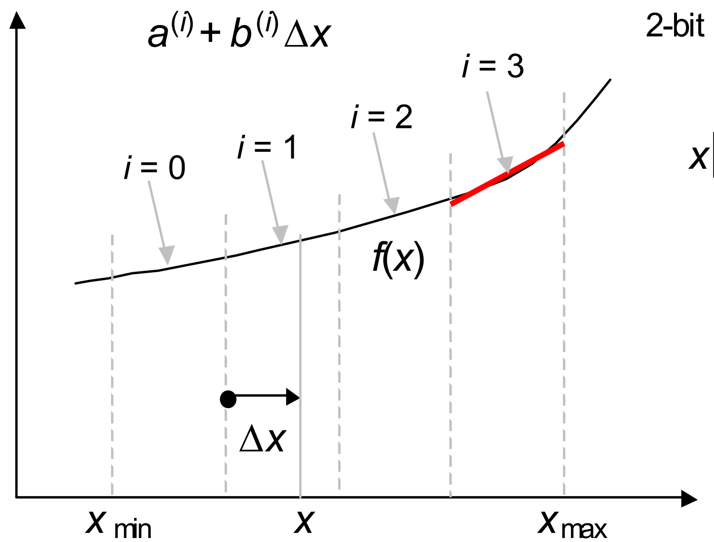
Pure logic → Hybrid solutions ← Pure tabular

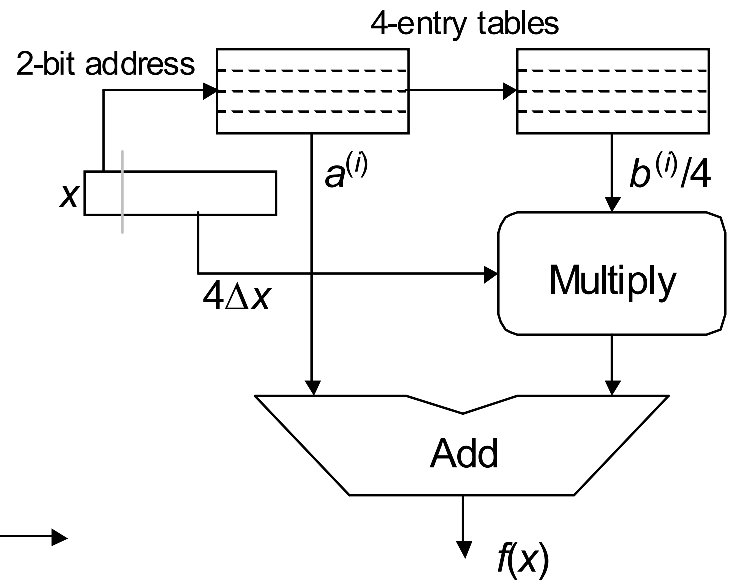
Historically, we started with the goal of designing logic circuits for particular arithmetic computations and ended up using tables to facilitate or speed up certain steps

From the other side, we aim for a tabular implementation and end up using peripheral logic circuits to reduce the table size

Some solutions can be derived starting at either endpoint

Linear Interpolation with 4 Subintervals





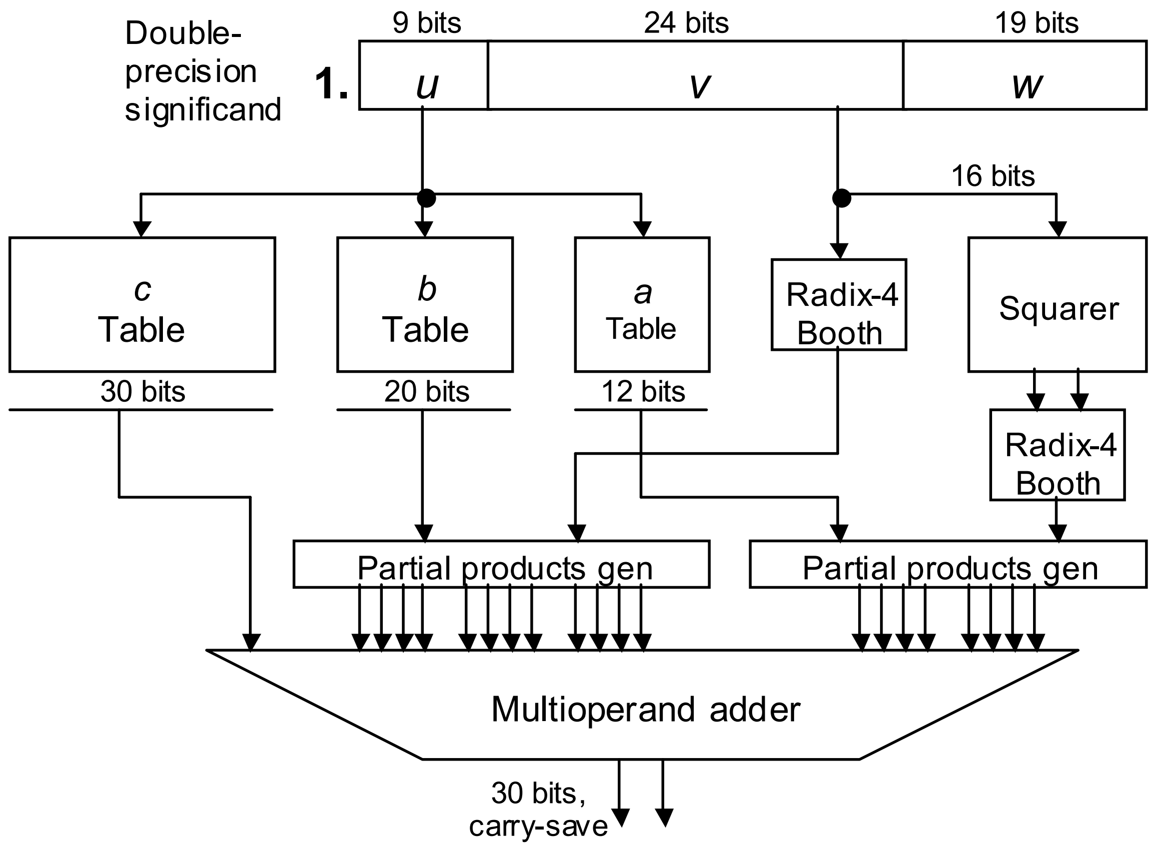
Linear interpolation for computing $f(x)$ using 4 subintervals.

Approximating $\log_2 x$ for x in [1, 2) using linear interpolation within 4 subintervals.

i	x_{lo}	x_{hi}	$a^{(i)}$	$b^{(i)}/4$	Max error
0	1.00	1.25	0.004 487	0.321 928	± 0.004 487
1	1.25	1.50	0.324 924	0.263 034	± 0.002 996
2	1.50	1.75	0.587 105	0.222 392	± 0.002 142
3	1.75	2.00	0.808 962	0.192 645	± 0.001 607

Second-Degree Interpolation Example

Approximation of reciprocal (1/*x*) and reciprocal square root (1/√*x*) functions with 29-30 bits of precision, so that a long floating-point result can be obtained with just one iteration at the end [Pine02]



$f(x) = c + bv + av^2$

2 adds

2 mult's

1 square

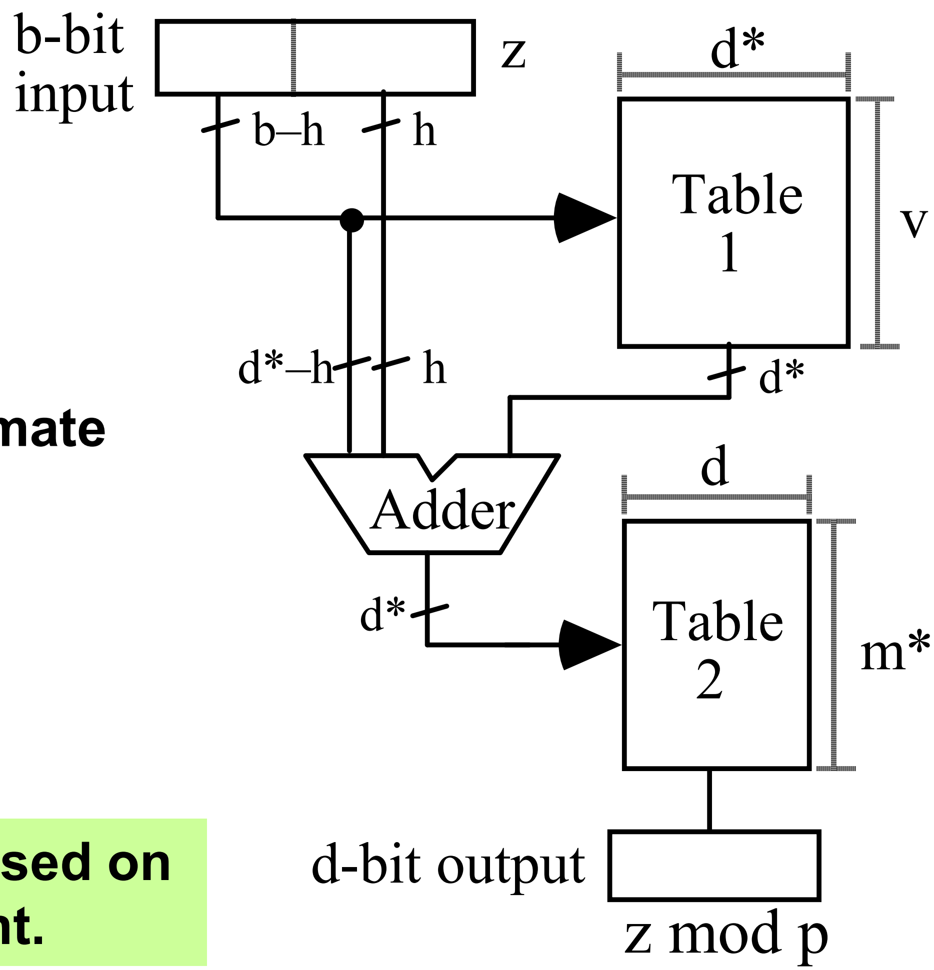
Comparable to a multiplier

Two-Level Table for Modular Reduction

Divide the argument *z* into (*b* – *h*)-bit upper part (*x*) and *h*-bit lower part (*y*), where *x* ends with *h* zeros

Table 1 provides a rough estimate for the final result

Table 2 refines the estimate



Modular reduction based on successive refinement.

FPGA-Based Integer Square-Rooters

Table 1 FPGA-based integer square-rooters [20]

Bits	CLBs	LUTs	Gates	Delay
8	12	21	~18K	15 ns
12	25	40	~37K	22 ns
16	42	73	~63K	40 ns

Table 2 FPGA-based integer square-rooters [21]

Bits	CLBs	LUTs	Gates	Delay
8	10	17	~12K	9 ns
12	22	39	~26K	20 ns
16	39	71	~47K	37 ns

The more computationally complex the function, the greater the cost and latency benefits of using table-based schemes