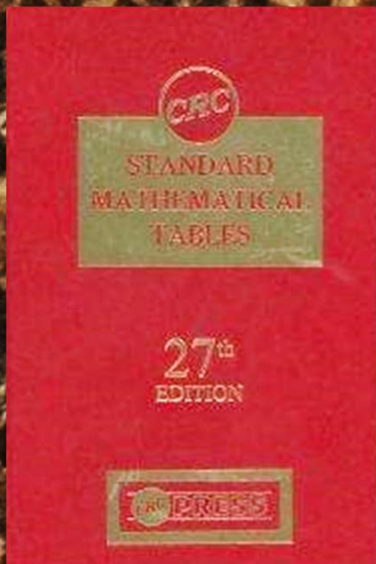
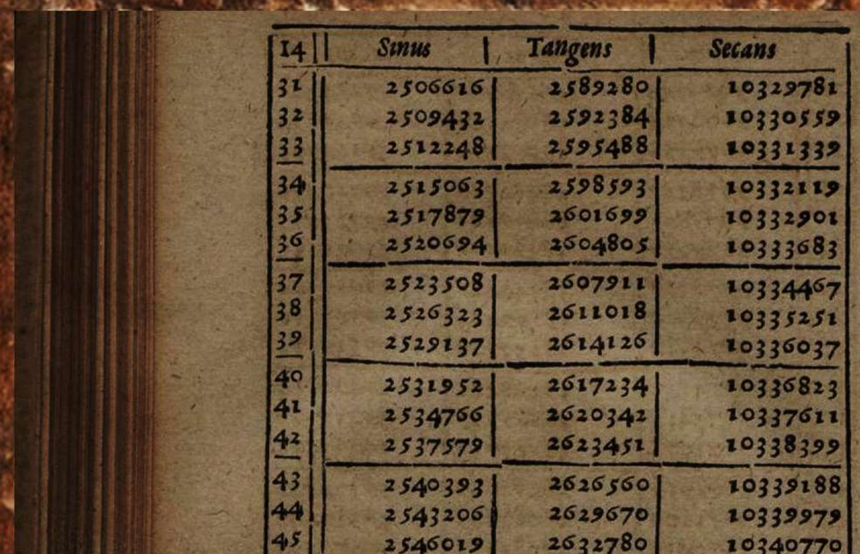
# The Return of Table-Based Computing




**Behrooz Parhami**

University of California, Santa Barbara



| 14 | Sinus | Tangens | Secans |
|---|---|---|---|
| 31 | 2506616 | 2589280 | 10329781 |
| 32 | 2509432 | 2592384 | 10330559 |
| 33 | 2512248 | 2595488 | 10331339 |
| 34 | 2515063 | 2598593 | 10332119 |
| 35 | 2517879 | 2601699 | 10332901 |
| 36 | 2520694 | 2604805 | 10333683 |
| 37 | 2523508 | 2607911 | 10334467 |
| 38 | 2526323 | 2611018 | 10335251 |
| 39 | 2529137 | 2614126 | 10336037 |
| 40 | 2531952 | 2617234 | 10336823 |
| 41 | 2534766 | 2620342 | 10337611 |
| 42 | 2537579 | 2623451 | 10338399 |
| 43 | 2540393 | 2626560 | 10339188 |
| 44 | 2543206 | 2629670 | 10339979 |
| 45 | 2546019 | 2632780 | 10340770 |

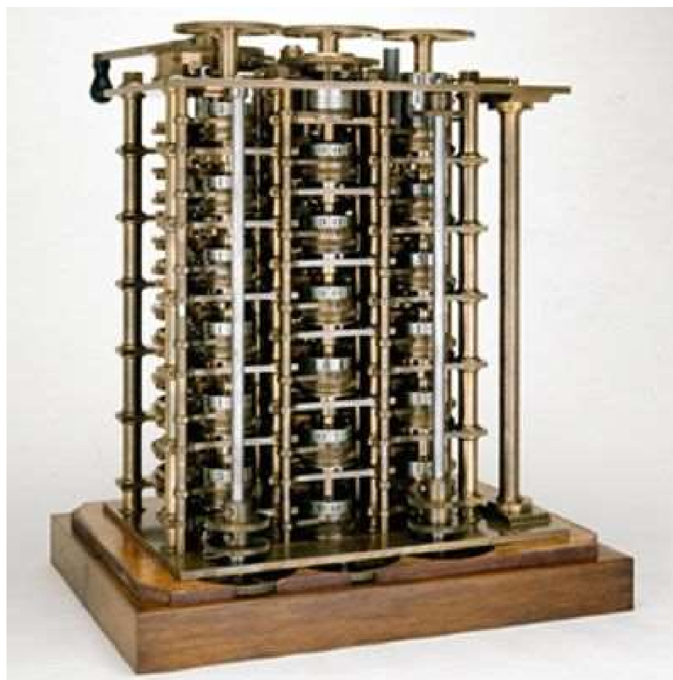
# About This Presentation

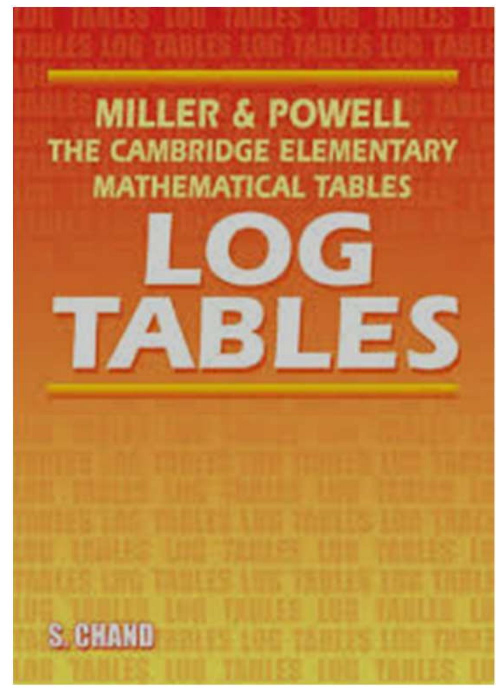
This slide show was first developed in fall of 2018 for an October 2018 talk at the 52nd Asilomar Conference on Signals, Systems, and Computers), Pacific Grove, CA, USA. 

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

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

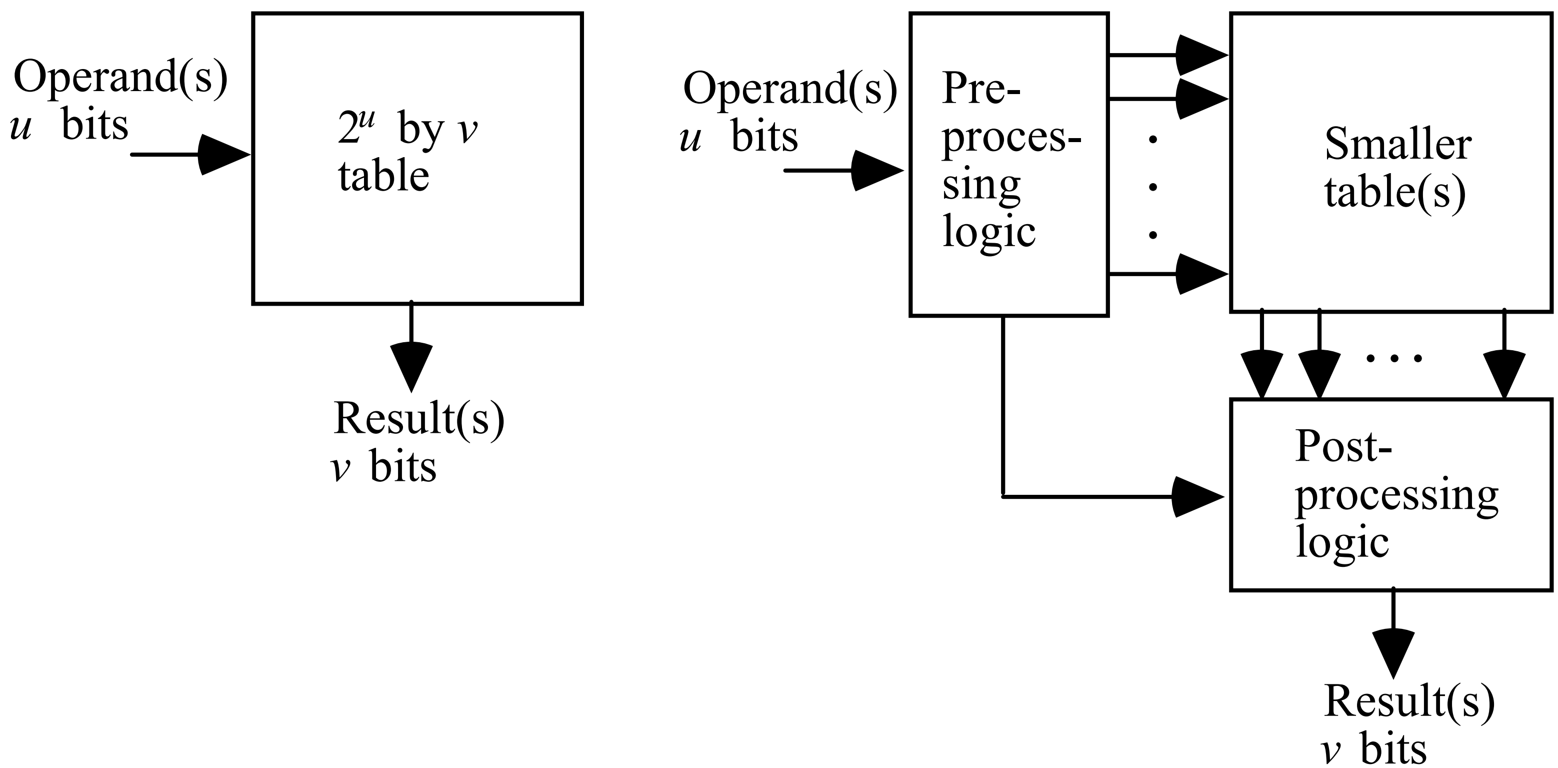
$= 0.549\ 236$

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

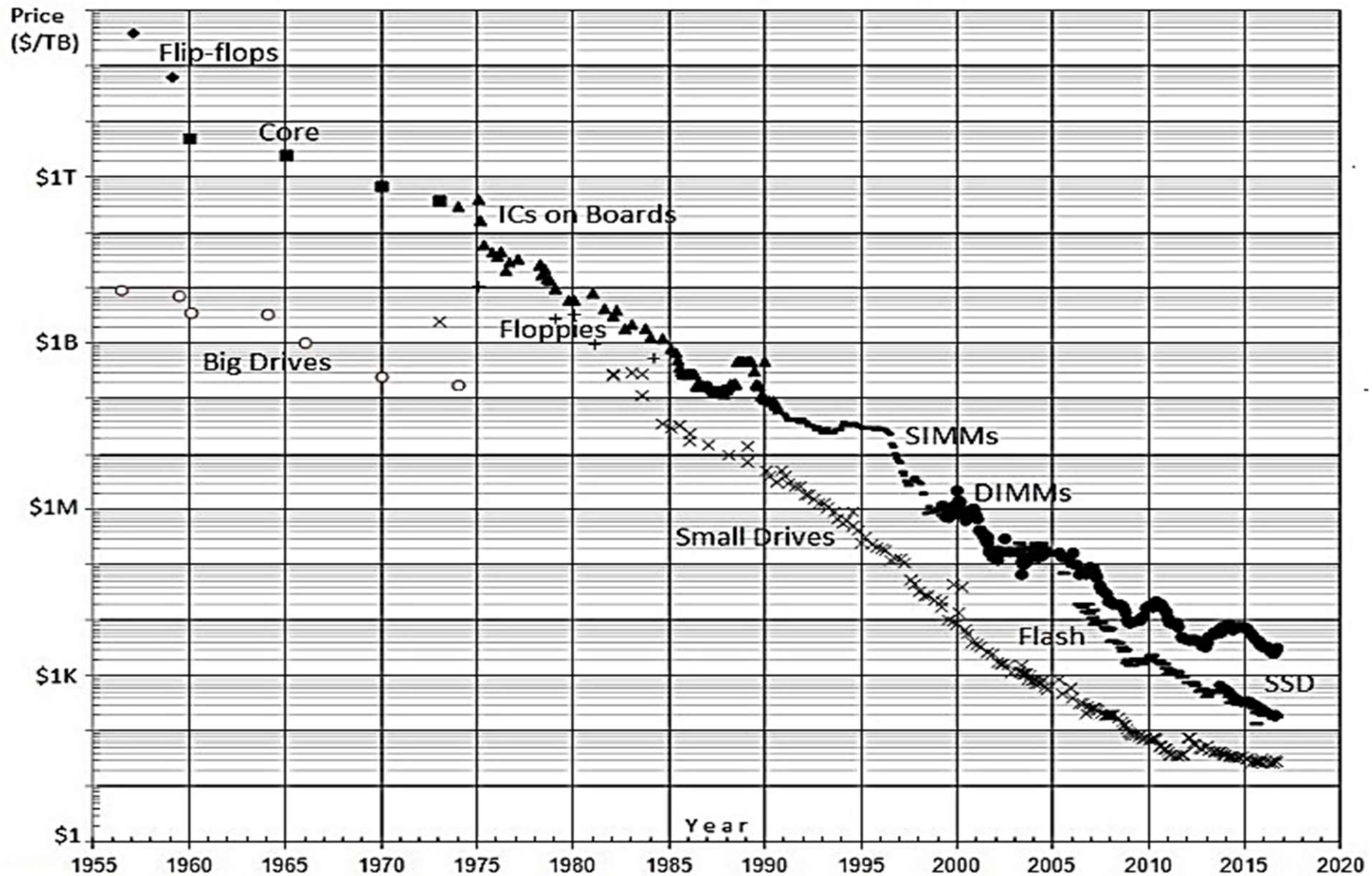
# Direct and Indirect Table-Lookup

**Direct lookup:** Operands serve as address bits into table

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



# Memory Cost Reduction Trends



# 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

# Example for Table Size Reduction

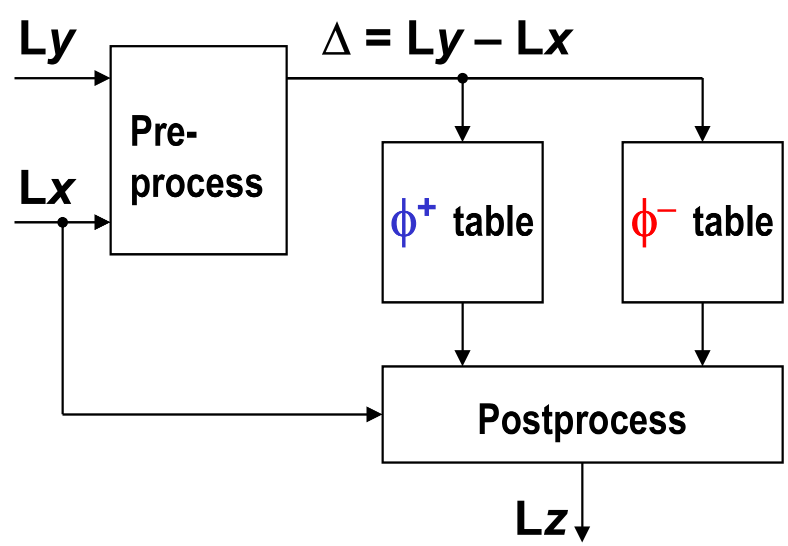
**Strategy: Reduce the table size by using an auxiliary unary function to evaluate a desired binary function**

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

**Solution: Let $\Delta$ = L$y$ – L$x$**

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

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

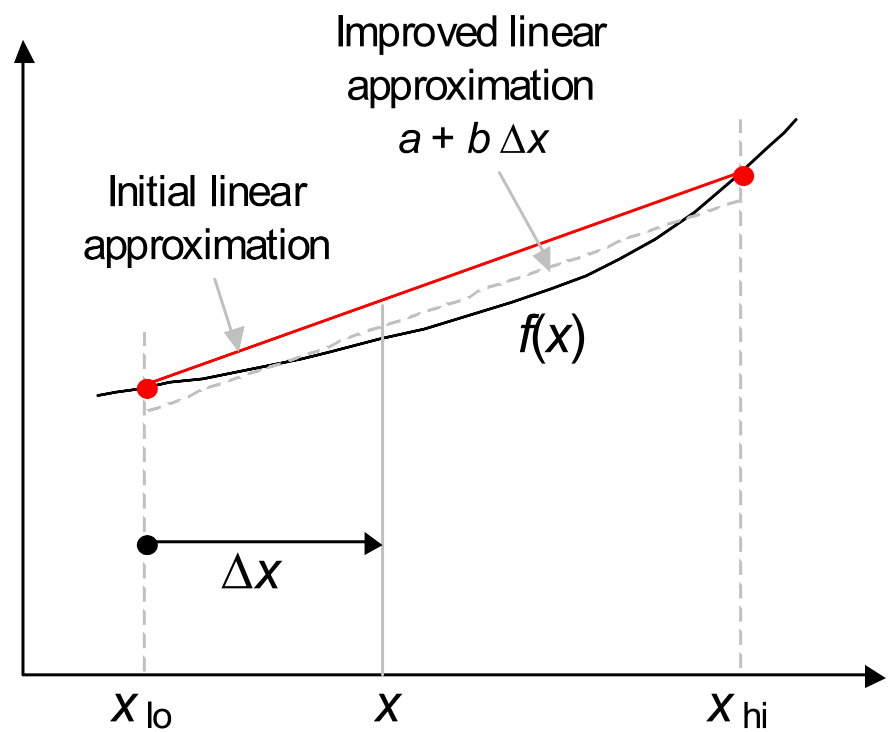


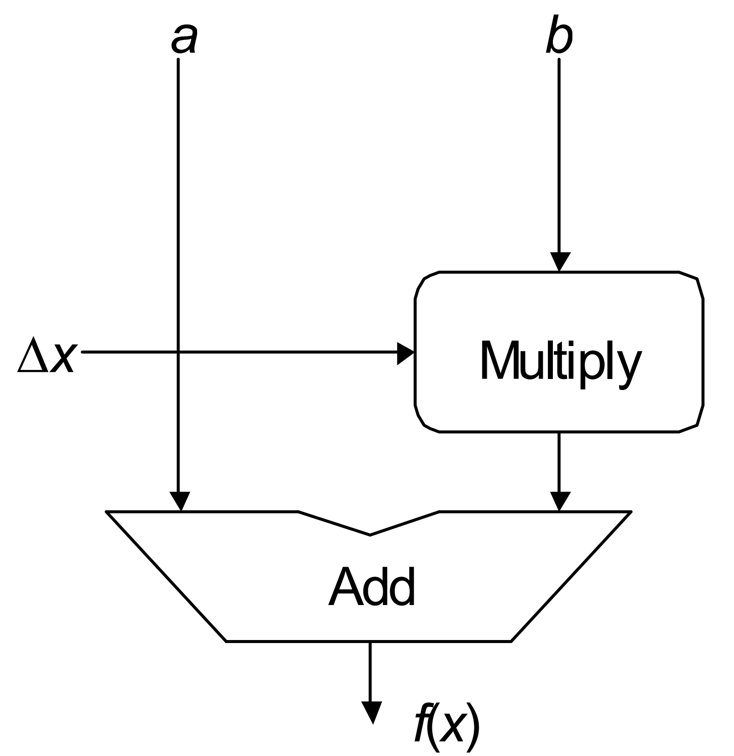
# Interpolating Memory Unit

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

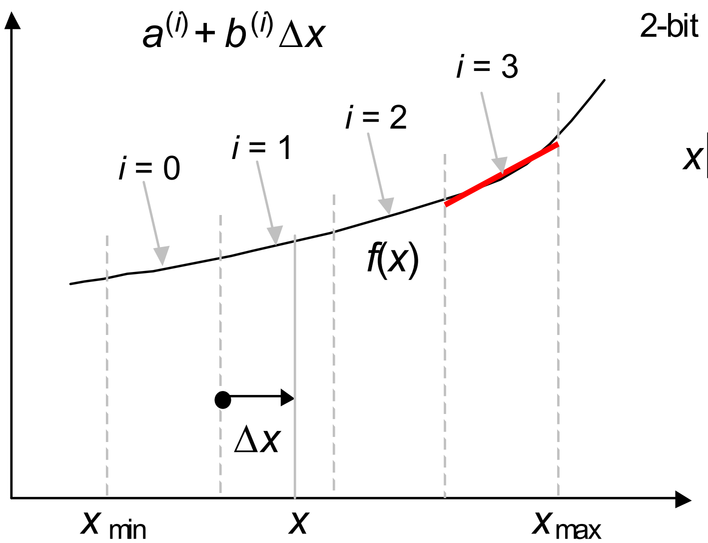
$$f(x) = f(x_{lo}) + \frac{x - x_{lo}}{x_{hi} - x_{lo}}[f(x_{hi}) - f(x_{lo})]$$

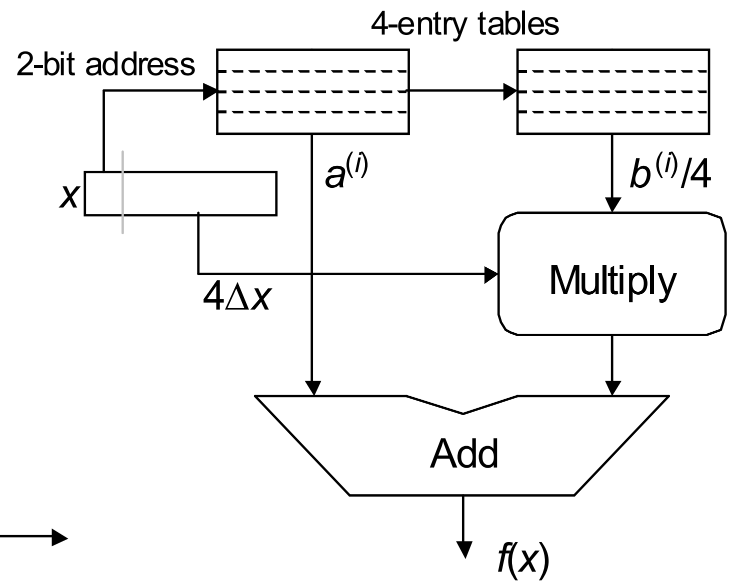
**4 adds, 1 divide, 1 multiply**
**(2 adds) (1 shift)**



# Linear Interpolation with 4 Subintervals



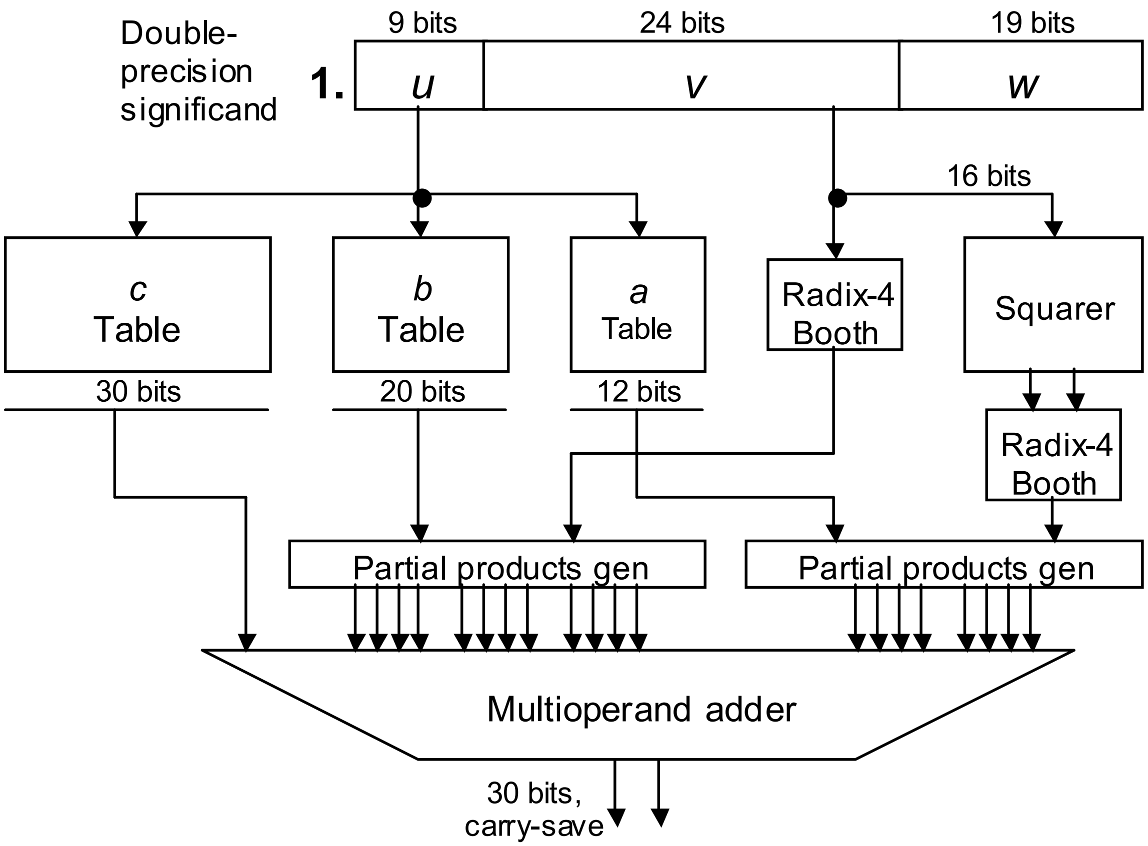

**Linear interpolation for computing $f(x)$ using 4 subintervals.**

**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]**




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

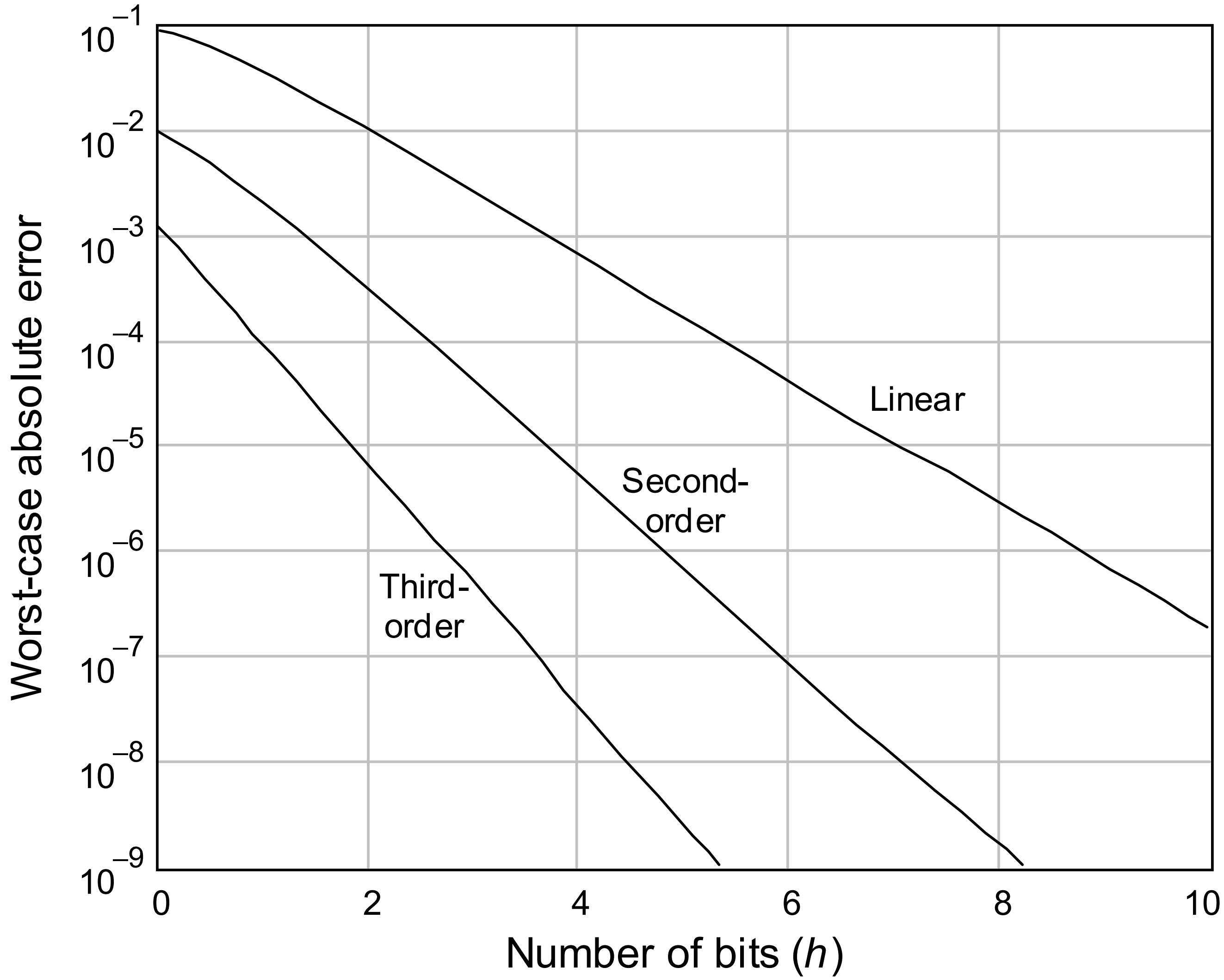
**2 adds**

**2 mult's**

**1 square**

**Comparable to a multiplier**

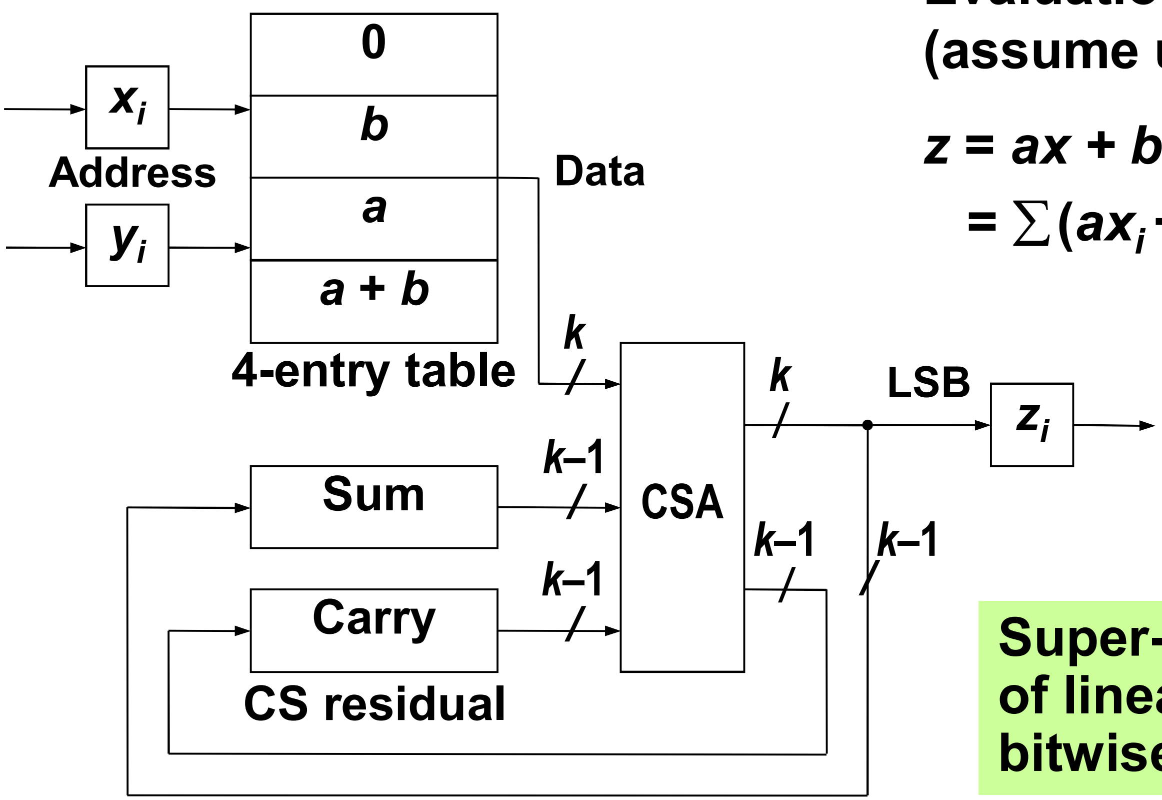
# Trade-offs in Cost, Speed, and Accuracy




**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**

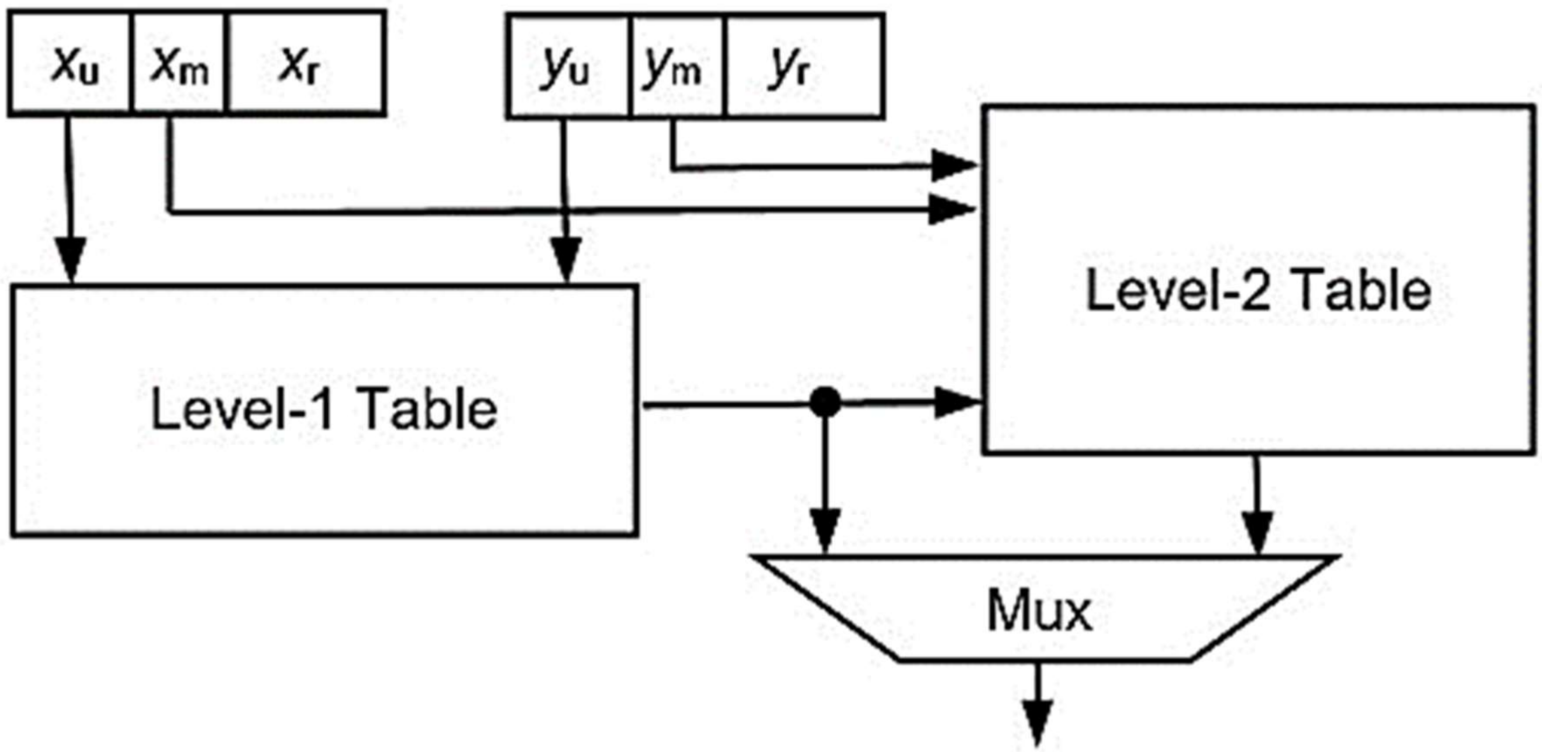



**Evaluation of linear expressions (assume unsigned values)**

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

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

# Two-Level Table for Approximate Sum



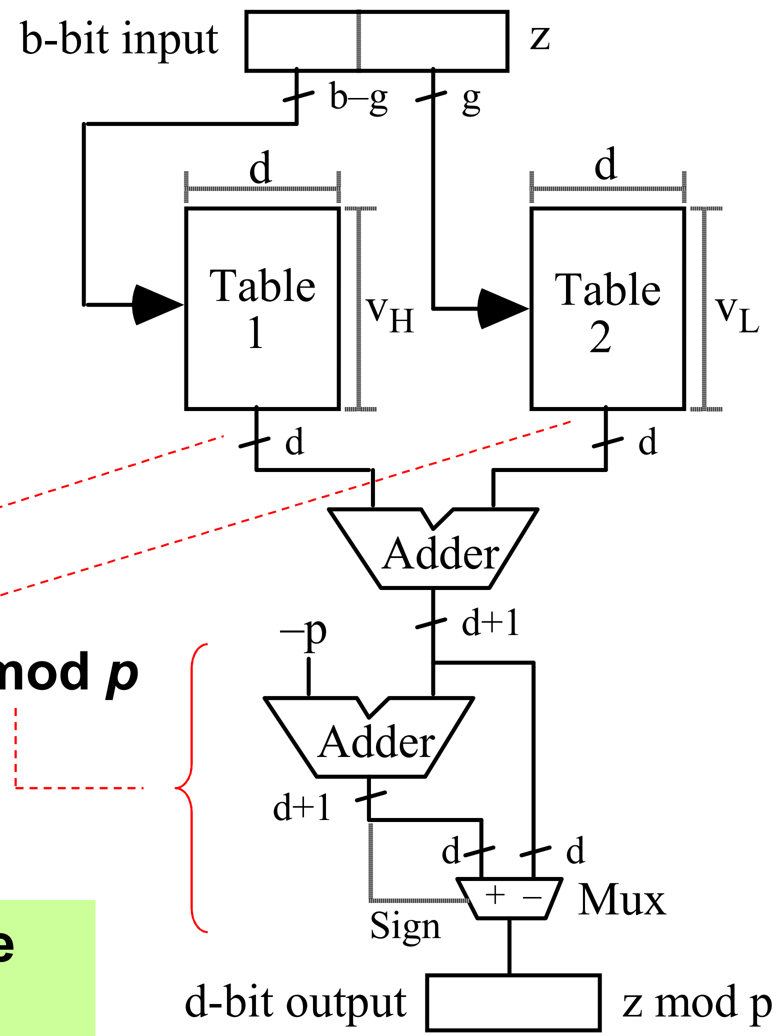

**Level-1 table provides a rough approximation for the sum**

**Level-2 table refines the sum for a greater precision**

# Modular Reduction: Computing *z* mod *p*

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

**$(x + y) \bmod p = (x \bmod p + y \bmod \mathrm{p}) \bmod p$**



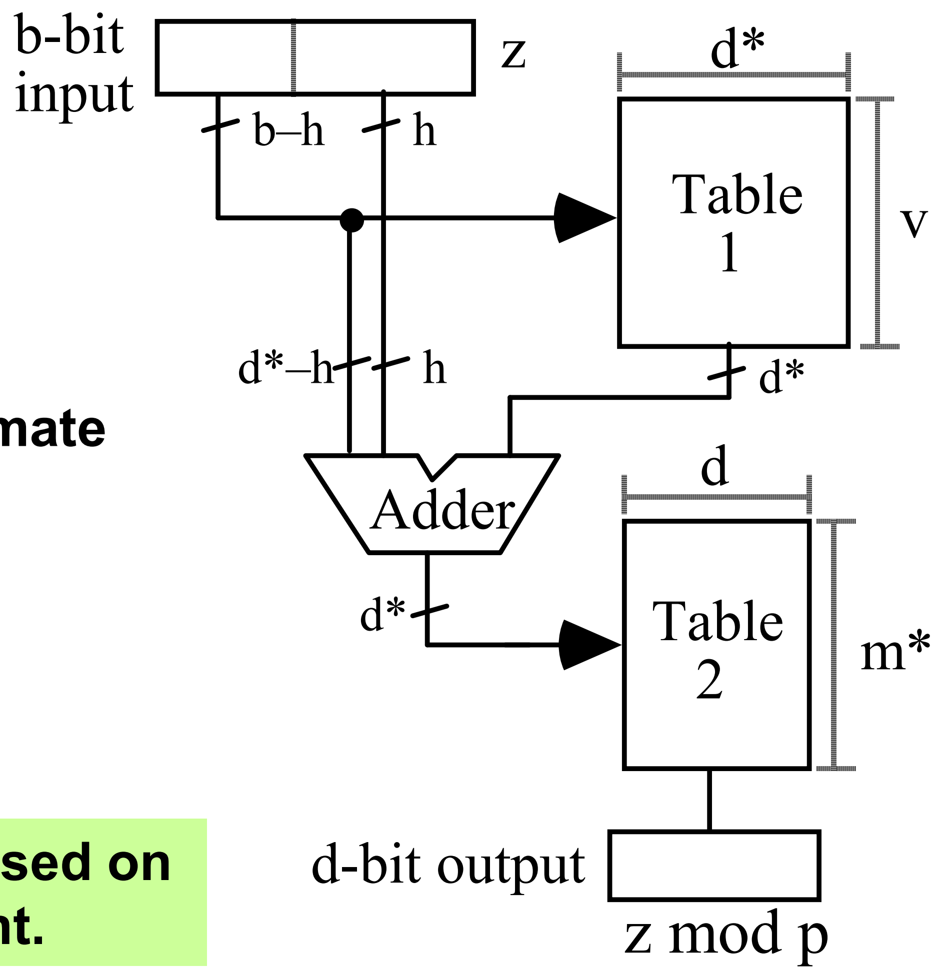

**Two-table modular reduction scheme based on divide-and-conquer.**

# Another 2-Level Table for Mod Reduction

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

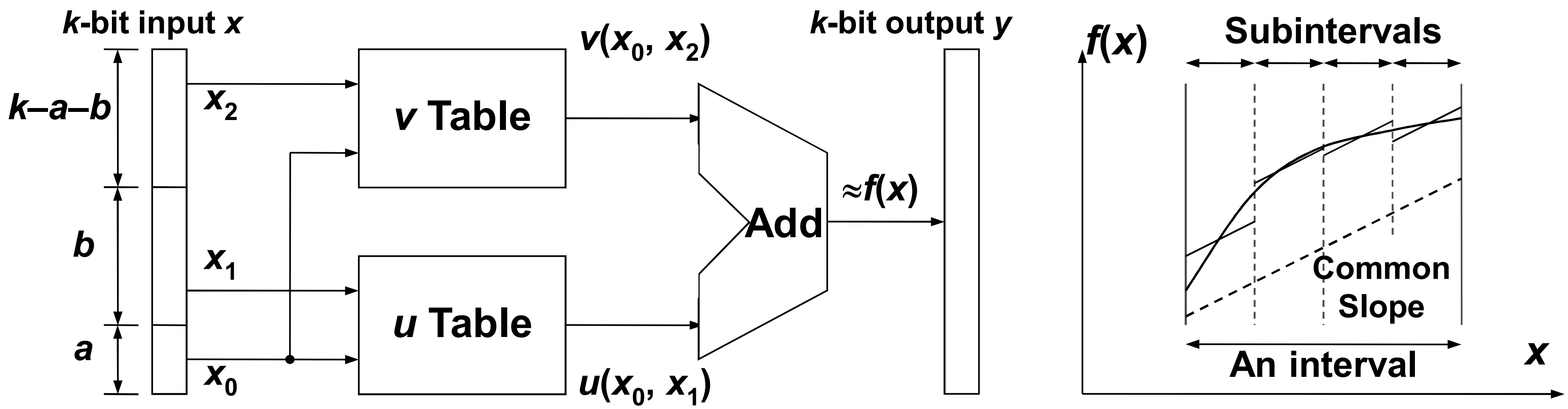
**Table 1 provides a rough estimate for the final result**

**Table 2 refines the estimate**




**Modular reduction based on successive refinement.**

# Bipartite and Multipartite Lookup Tables




(a) Hardware realization (b) Linear approximation

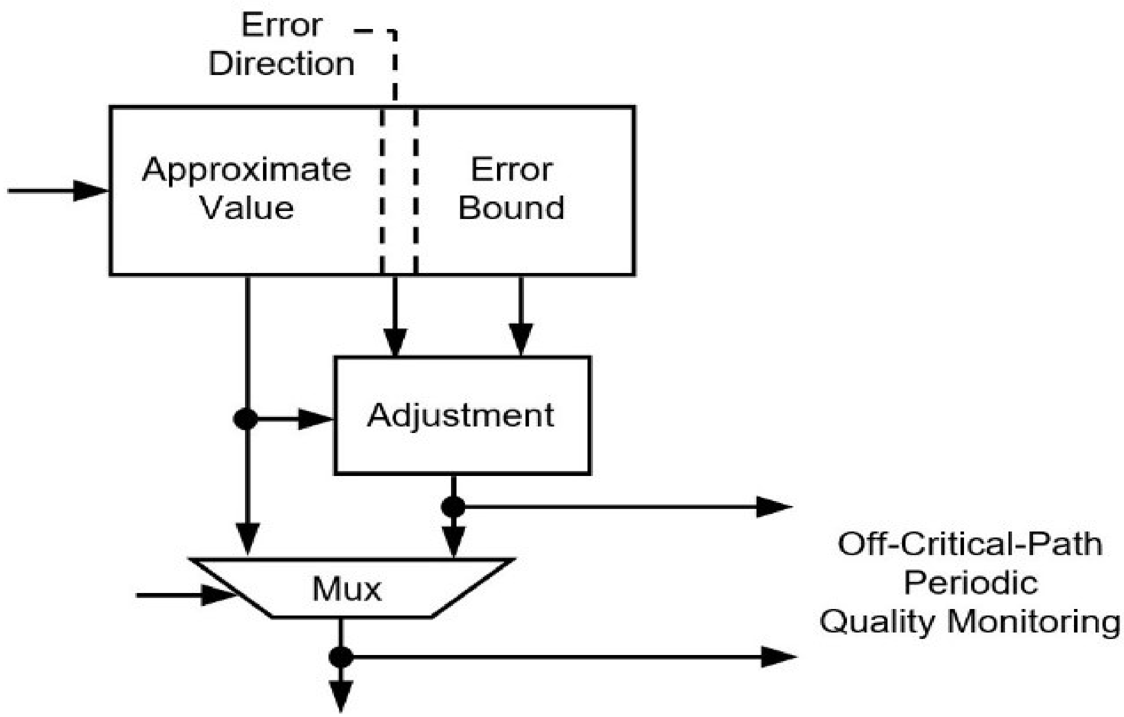
Divide the domain of interest into $2^a$ intervals, each of which is further divided into $2^b$ smaller subintervals

**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

# 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**

# 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**

# Conclusions and Future Work

**Use of tables is expanding:** Memory cost ↓ Memory size ↑

**Benefits of Returning to Table-Based Computing:**

Fast approximation + added precision as needed


Knowable error direction and magnitude

Table-size/latency/precision trade-offs

Avoid waste from recomputation

**Future work and more detailed comparisons**

Assessment of speed benefits in application contexts


Quantifying cost and energy reduction

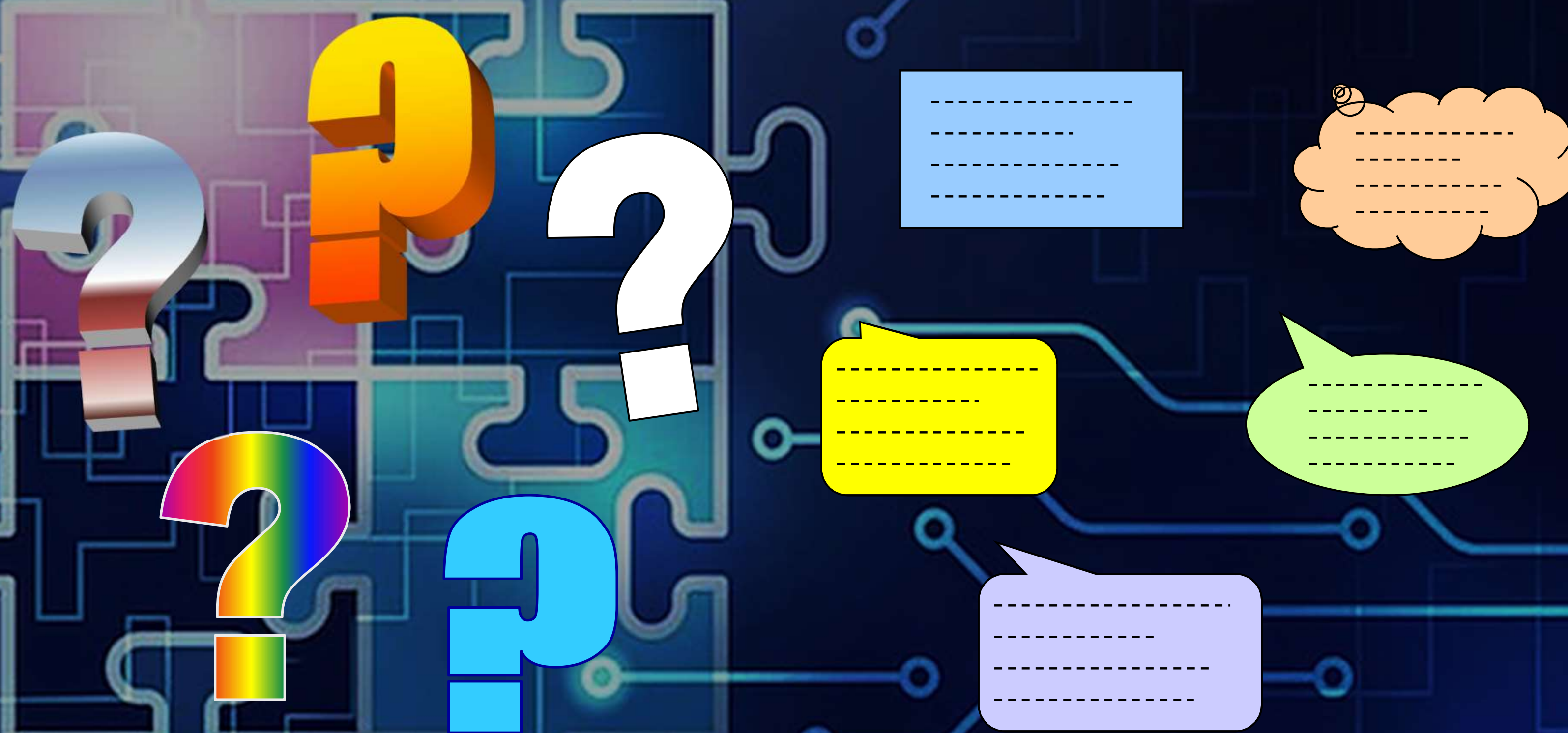
Bit-level table optimization methods

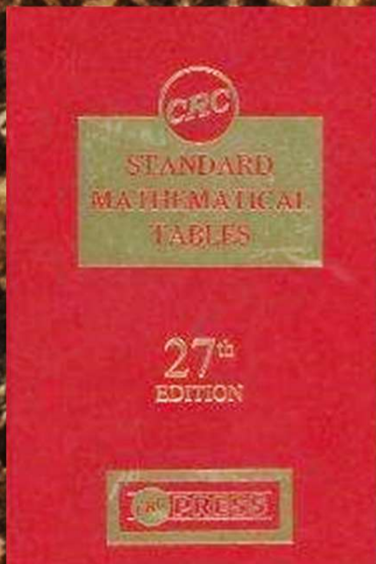
Sparse and associative tables

# Questions or Comments?

parhami@ece.ucsb.edu

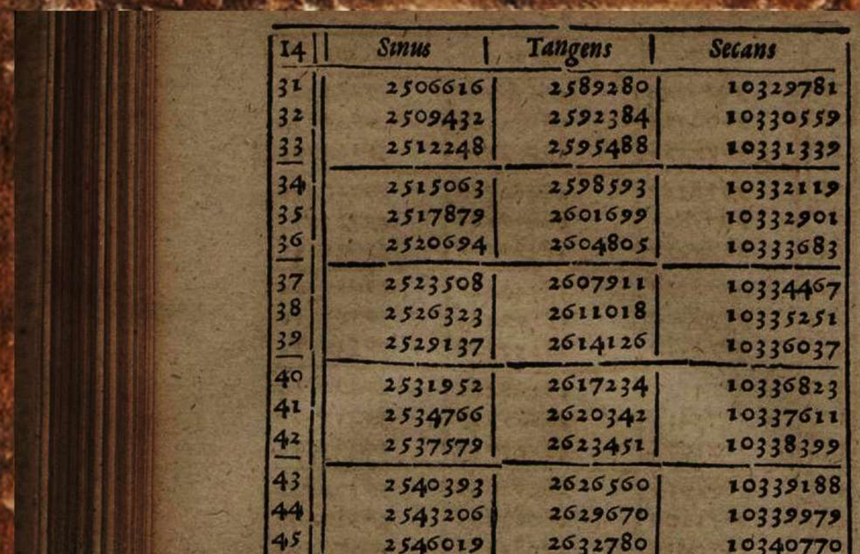
http://www.ece.ucsb.edu/~parhami/






# The Return of Table-Based Computing

## Back-Up Slides



| 14 | Sinus | Tangens | Secans |
|---|---|---|---|
| 31 | 2506616 | 2589280 | 10329781 |
| 32 | 2509432 | 2592384 | 10330559 |
| 33 | 2512248 | 2595488 | 10331339 |
| 34 | 2515063 | 2598593 | 10332119 |
| 35 | 2517879 | 2601699 | 10332901 |
| 36 | 2520694 | 2604805 | 10333683 |
| 37 | 2523508 | 2607911 | 10334467 |
| 38 | 2526323 | 2611018 | 10335251 |
| 39 | 2529137 | 2614126 | 10336037 |
| 40 | 2531952 | 2617234 | 10336823 |
| 41 | 2534766 | 2620342 | 10337611 |
| 42 | 2537579 | 2623451 | 10338399 |
| 43 | 2540393 | 2626560 | 10339188 |
| 44 | 2543206 | 2629670 | 10339979 |
| 45 | 2546019 | 2632780 | 10340770 |

**Behrooz Parhami**

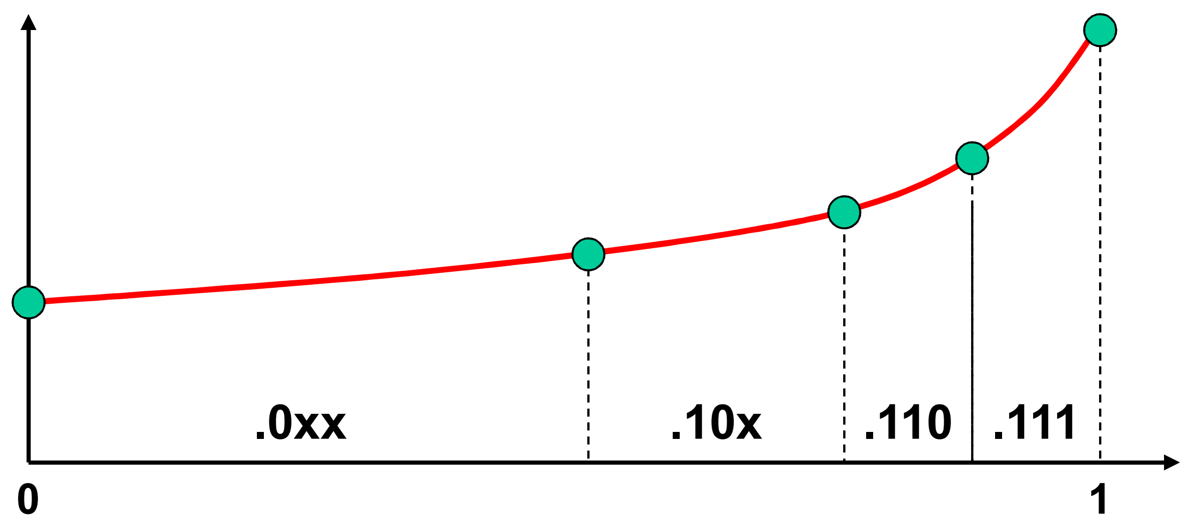
University of California,
Santa Barbara

# Interpolation with Nonuniform Intervals

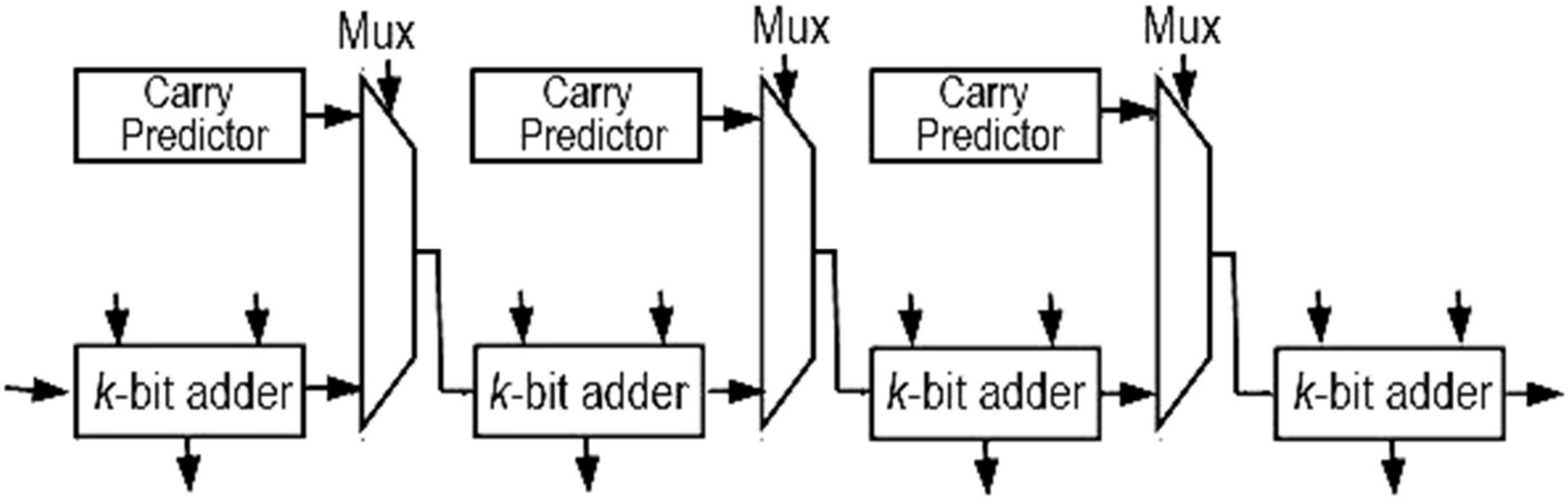
One way to use interpolation with nonuniform intervals to successively divide ranges and subranges of interest into 2 parts, with finer divisions used where the function exhibits greater curvature (nonlinearity)

In this way, a number of leading bits can be used to decide which subrange is applicable

The [0,1) range divided into 4 nonuniform intervals



# 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**