The Return of Table-Based Computing

Behrooz Parhami
University of California, Santa Barbara
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. All rights reserved for the author. ©2018 Behrooz Parhami

<table>
<thead>
<tr>
<th>Edition</th>
<th>Released</th>
<th>Revised</th>
<th>Revised</th>
<th>Revised</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>Fall 2018</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

Diagram:

- **Operand(s)**: $u$ bits
- **Pre-processing logic**
- **Smaller table(s)**
- **Post-processing logic**
- **Result(s)**: $v$ bits
Memory Cost Reduction Trends

The diagram illustrates the price per terabyte (S/TB) of various storage technologies over time. The x-axis represents the year, ranging from 1955 to 2020, while the y-axis shows the price in $1T, $1B, $1M, $1K, and $1, allowing for a clear comparison of cost trends over decades for technologies such as flip-flops, cores, ICs on boards, floppy drives, small drives, DIMMs, SIMMs, flash, and SSDs. The downward trend lines elaborate on the decreasing cost of memory over time.
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 $\rightarrow$ Hybrid solutions $\rightarrow$ 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 \( \Delta = L_y - L_x \)

\[
L_z = \log(x \pm y) \\
= \log(x (1 \pm y/x)) \\
= \log x + \log(1 \pm y/x) \\
= L_x + \log(1 \pm \log^{-1}\Delta)
\]

\[
L_x + \phi^+(\Delta) \quad \text{and} \quad L_x + \phi^-(\Delta)
\]

Pre-process \( L_y \) and \( L_x \) using \( \phi^+ \) and \( \phi^- \) tables.

Post-process to get \( L_z \) from \( \Delta = L_y - L_x \).
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

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

<table>
<thead>
<tr>
<th>$i$</th>
<th>$x_{lo}$</th>
<th>$x_{hi}$</th>
<th>$a^{(i)}$</th>
<th>$b^{(i)}/4$</th>
<th>Max error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.00</td>
<td>1.25</td>
<td>0.004487</td>
<td>0.321928</td>
<td>± 0.004487</td>
</tr>
<tr>
<td>1</td>
<td>1.25</td>
<td>1.50</td>
<td>0.324924</td>
<td>0.263034</td>
<td>± 0.002996</td>
</tr>
<tr>
<td>2</td>
<td>1.50</td>
<td>1.75</td>
<td>0.587105</td>
<td>0.222392</td>
<td>± 0.002142</td>
</tr>
<tr>
<td>3</td>
<td>1.75</td>
<td>2.00</td>
<td>0.808962</td>
<td>0.192645</td>
<td>± 0.001607</td>
</tr>
</tbody>
</table>
Approximation of reciprocal \(1/x\) and reciprocal square root \((1/\sqrt{x})\) 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]

\[
f(x) = c + bv + av^2
\]

1 square

2 mult's

2 adds

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.

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.

$$(x + y) \mod p = (x \mod p + y \mod p) \mod p$$

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

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

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

(b) Linear approximation

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.

Bipartite tables: Main idea
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]

<table>
<thead>
<tr>
<th>Bits</th>
<th>CLBs</th>
<th>LUTs</th>
<th>Gates</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>12</td>
<td>21</td>
<td>~18K</td>
<td>15 ns</td>
</tr>
<tr>
<td>12</td>
<td>25</td>
<td>40</td>
<td>~37K</td>
<td>22 ns</td>
</tr>
<tr>
<td>16</td>
<td>42</td>
<td>73</td>
<td>~63K</td>
<td>40 ns</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Bits</th>
<th>CLBs</th>
<th>LUTs</th>
<th>Gates</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>10</td>
<td>17</td>
<td>~12K</td>
<td>9 ns</td>
</tr>
<tr>
<td>12</td>
<td>22</td>
<td>39</td>
<td>~26K</td>
<td>20 ns</td>
</tr>
<tr>
<td>16</td>
<td>39</td>
<td>71</td>
<td>~47K</td>
<td>37 ns</td>
</tr>
</tbody>
</table>

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
The Return of Table-Based Computing

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 $4k$-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