Lecture 10
Computational tricks and Techniques

Forrest Brewer
Embedded Computation

- Computation involves Tradeoffs
  - Space vs. Time vs. Power vs. Accuracy
- Solution depends on resources as well as constraints
  - Availability of large space for tables may obviate calculations
  - Lack of sufficient scratch-pad RAM for temporary space may cause restructuring of whole design
  - Special Processor or peripheral features can eliminate many steps – but tend to create resource arbitration issues
- Some computations admit incremental improvement
  - Often possible to iterate very few times on average
    - Timing is data dependent
    - If careful, allows for soft-fail when resources are exceeded
Overview

• Basic Operations
  – Multiply (high precision)
  – Divide

• Polynomial representation of functions
  – Evaluation
  – Interpolation
  – Splines

• Putting it together
Multiply

- It is not so uncommon to work on a machine without hardware multiply or divide support
  - Can we do better than $O(n^2)$ operations to multiply n-bit numbers?

- Idea 1: Table Lookup
  - Not so good – for 8-bit x 8-bit we need 65k 16-bit entries
    - By symmetry, we can reduce this to 32k entries – still very costly – we often have to produce 16-bit x 16-bit products…
Multiply – cont’d

- Idea 2: Table of Squares
  - Consider: \( \left( \frac{u + v}{2} \right) \left( \frac{u - v}{2} \right) = \left( \frac{1}{4} \right) (u^2 - v^2) \)
  - Given a and b, we can write: \( u = a + b, v = a - b \)
  - Then to find ab, we just look up the squares and subtract…
  - This is much better since the table now has \( 2^{n+1} - 2 \) entries
  - Eg consider n=7-bits, then the table is \( \{u^2: u=1..254\} \)
    - If \( a = 17 \) and \( b = 18 \), \( u = 35, v = 1 \) so \( u^2 - v^2 = 1224 \), dividing by 4 (shift!) we get 306
    - Total cost is: 1 add, 2 sub, 2 lookups and 2 shifts…
      (this idea was used by Mattel, Atari, Nintendo…)
Multiply – still more

- Often, you need double precision multiply even if your machine supports multiply
  - Is there something better than 4 single-precision multiplies to get double precision?
  - Yes – can do it in 3 multiplies:
    \[
    u = 2^n u_1 + u_2 , v = 2^n v_1 + v_2
    \]
    \[
    uv = (2^{2n} + 2^n)u_1 v_1 + 2^n (u_1 - u_0)(v_0 - v_1) + (2^n + 1)u_0 v_0
    \]
    - Note that you only have to multiply \(u_1 v_1\), \((u_1 - u_0)(v_0 - v_1)\) and \(u_0 v_0\)
      Every thing else is just add, sub and shift –

- More importantly, this trick can be done recursively
  - Leads to \(O(n^{1.58})\) multiply for large numbers
Multiply – the (nearly) final word

- The fastest known algorithm for multiply is based on FFT! (Schönhage and Strassen ‘71)
  - The tough problem in multiply is dealing with the partial products which are of the form: $u_r v_0 + u_{r-1} v_1 + u_{r-2} v_2 + \ldots$
  - This is the form of a convolution – but we have a neat trick for convolution…

$$f \ast g = F^{-1}\{F\{g\} \cdot F\{h\}\}$$

- This trick reduces the cost of the partials to $O(n \ln(n))$
  - Overall runtime is $O(n \ln(n) \ln(\ln(n)))$
- Practically, the trick wins for $n > 200$ bits…
  - 2007 Furer showed $O(n \ln(n) 2^{\lceil \log(n) \rceil})$ – this wins for $n > 10^{10}$ bits..
Division

- There are hundreds of ‘fast’ division algorithms but none are particularly fast
  - Often can re-scale computations so that divisions are reduced to shift – need to pre-calculate constants.

- Newton’s Method can be used to find $1/v$
  - Start with small table $1/v$ given $v$. (Here we assume $v>1$)
  - Let $x_0$ be the first guess
  - Iterate: $x_{n+1} = 2x_n - v x_n^2$
  - Note: if

$$x_n = (1/v) + \varepsilon, \quad x_{n+1} = 2/v + 2\varepsilon - (1/v) - (2\varepsilon) - v\varepsilon^2 = 1/v - v\varepsilon^2$$

  - E.g. $v=7$, $x_0=0.1$ $x_1=0.13$ $x_2=0.1417$ $x_3=0.142848$
    $x_4=0.1428571422$ – doubles number of bits each iteration
  - Win here is if table is large enough to get 2 or more digits
Polynomial evaluation

- When represented in monomial basis, polynomial
  \[ p_{n-1}(t) = x_1 + x_2 t + \cdots + x_n t^{n-1} \]
  can be evaluated efficiently using Horner's nested evaluation scheme
  \[ p_{n-1}(t) = x_1 + t(x_2 + t(x_3 + t(\cdots (x_{n-1} + tx_n) \cdots ))) \]
  which requires only \( n \) additions and \( n \) multiplications
- For example,
  \[ 1 - 4t + 5t^2 - 2t^3 + 3t^4 = 1 + t(-4 + t(5 + t(-2 + 3t))) \]
- Other manipulations of interpolating polynomial, such as differentiation or integration, are also relatively easy with monomial basis representation
Interpolation by Polynomials

- Function values listed in short table – how to approximate values not in the table?
- Two basic strategies
  - Fit all points into a (big?) polynomial
    - Lagrange and Newton Interpolation
    - Stability issues as degree increases
  - Fit subsets of points into several overlapping polynomials (splines)
    - Simpler to bound deviations since get new polynomial each segment
    - 1st order is just linear interpolation
    - Higher order allows matching of slopes of splines at nodes
    - Hermite Interpolation matches values and derivatives at each node
- Sometimes polynomials are a poor choice—
  - Asymptotes and Meromorphic functions
  - Rational fits (quotients of polynomials) are good choice
  - Leads to non-linear fitting equations
What is Interpolation?

Given \((x_0,y_0), (x_1,y_1), \ldots, (x_n,y_n)\), find the value of ‘y’ at a value of ‘x’ that is not given.
Interpolation

- Basic interpolation problem: for given data
  \[(t_1, y_1), (t_2, y_2), \ldots, (t_m, y_m)\] with \(t_1 < t_2 < \cdots < t_m\)
  determine function \(f: \mathbb{R} \to \mathbb{R}\) such that
  \[f(t_i) = y_i, \quad i = 1, \ldots, m\]

- \(f\) is *interpolating function*, or *interpolant*, for given data

- Additional data might be prescribed, such as slope of interpolant at given points

- Additional constraints might be imposed, such as smoothness, monotonicity, or convexity of interpolant

- \(f\) could be function of more than one variable, but we will consider only one-dimensional case
**Basis functions**

- Family of functions for interpolating given data points is spanned by set of *basis functions* $\phi_1(t), \ldots, \phi_n(t)$

- Interpolating function $f$ is chosen as linear combination of basis functions,

\[
  f(t) = \sum_{j=1}^{n} x_j \phi_j(t)
\]

- Requiring $f$ to interpolate data $(t_i, y_i)$ means

\[
  f(t_i) = \sum_{j=1}^{n} x_j \phi_j(t_i) = y_i, \quad i = 1, \ldots, m
\]

which is system of linear equations $Ax = y$ for $n$-vector $x$ of parameters $x_j$, where entries of $m \times n$ matrix $A$ are given by $a_{ij} = \phi_j(t_i)$
Polynomial interpolation

- Simplest and most common type of interpolation uses polynomials
- Unique polynomial of degree at most $n-1$ passes through $n$ data points $(t_i, y_i)$, $i = 1, \ldots, n$, where $t_i$ are distinct

\begin{itemize}
  \item \textbf{Monomial basis functions}
  \[
  \phi_j(t) = t^{j-1}, \quad j = 1, \ldots, n
  \]
  give interpolating polynomial of form
  \[
  p_{n-1}(t) = x_1 + x_2 t + \cdots + x_n t^{n-1}
  \]
  with coefficients $x$ given by $n \times n$ linear system
  \[
  Ax = \begin{bmatrix}
  1 & t_1 & \cdots & t_1^{n-1} \\
  1 & t_2 & \cdots & t_2^{n-1} \\
  \vdots & \vdots & \ddots & \vdots \\
  1 & t_n & \cdots & t_n^{n-1}
  \end{bmatrix}
  \begin{bmatrix}
  x_1 \\
  x_2 \\
  \vdots \\
  x_n
  \end{bmatrix}
  =
  \begin{bmatrix}
  y_1 \\
  y_2 \\
  \vdots \\
  y_n
  \end{bmatrix}
  = y
  \]
  \item Matrix of this form is called \textit{Vandermonde matrix}
\end{itemize}
Example

- Determine polynomial of degree two interpolating three data points \((-2, -27), (0, -1), (1, 0)\).

- Using monomial basis, linear system is

\[
A x = \begin{bmatrix}
1 & t_1 & t_1^2 \\
1 & t_2 & t_2^2 \\
1 & t_3 & t_3^2
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3
\end{bmatrix} = \begin{bmatrix}
y_1 \\
y_2 \\
y_3
\end{bmatrix} = y
\]

- For these particular data, system is

\[
\begin{bmatrix}
1 & -2 & 4 \\
1 & 0 & 0 \\
1 & 1 & 1
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3
\end{bmatrix} = \begin{bmatrix}
-27 \\
-1 \\
0
\end{bmatrix}
\]

whose solution is \(x = [-1, 5, -4]^T\), so interpolating polynomial is

\[
p_2(t) = -1 + 5t - 4t^2
\]

\(O(n^3)\) operations to solve linear system
Conditioning

- For monomial basis, matrix A is increasingly ill-conditioned as degree increases
- Ill-conditioning does not prevent fitting data points well, since residual for linear system solution will be small
- It means that values of coefficients are poorly determined
- Change of basis still gives same interpolating polynomial for given data, but representation of polynomial will be different

Conditioning with monomial basis can be improved by shifting and scaling independent variable $t$

$$
\phi_j(t) = \left( \frac{t - c}{d} \right)^{j-1}
$$

where, $c = (t_1 + t_n)/2$ is midpoint and $d = (t_n - t_1)/2$ is half of range of data

Still not well-conditioned, Looking for better alternative
Lagrange interpolation

For given set of data points \((t_i, y_i), i = 1, \ldots, n\), Lagrange basis functions are defined by

\[
l_j(t) = \prod_{k=1, k \neq j}^{n} \frac{(t - t_k)}{(t_j - t_k)}, \quad j = 1, \ldots, n
\]

For Lagrange basis,

\[
l_j(t_i) = \begin{cases} 
1 & \text{if } i = j \\ 
0 & \text{if } i \neq j 
\end{cases}, \quad i, j = 1, \ldots, n
\]

so matrix of linear system \(Ax = y\) is identity matrix

Thus, Lagrange polynomial interpolating data points \((t_i, y_i)\) is given by

\[
p_{n-1}(t) = y_1 \ell_1(t) + y_2 \ell_2(t) + \cdots + y_n \ell_n(t)
\]

Easy to determine, but expensive to evaluate, integrate and differentiate comparing to monomials
Example

- Use Lagrange interpolation to determine interpolating polynomial for three data points \((-2, -27), (0, -1), (1, 0)\)

- Lagrange polynomial of degree two interpolating three points \((t_1, y_1), (t_2, y_2), (t_3, y_3)\) is given by \(p_2(t) = \)

  \[
  y_1 \frac{(t - t_2)(t - t_3)}{(t_1 - t_2)(t_1 - t_3)} + y_2 \frac{(t - t_1)(t - t_3)}{(t_2 - t_1)(t_2 - t_3)} + y_3 \frac{(t - t_1)(t - t_2)}{(t_3 - t_1)(t_3 - t_2)}
  \]

- For these particular data, this becomes

  \[
  p_2(t) = -27 \frac{t(t - 1)}{(-2)(-2 - 1)} + (-1) \frac{(t + 2)(t - 1)}{(2)(-1)}
  \]
Piecewise interpolation (Splines)

- Fitting single polynomial to large number of data points is likely to yield unsatisfactory oscillating behavior in interpolant

- Piecewise polynomials provide alternative to practical and theoretical difficulties with high-degree polynomial interpolation. Main advantage of piecewise polynomial interpolation is that large number of data points can be fit with low-degree polynomials

- In piecewise interpolation of given data points \((t_i, y_i)\), different functions are used in each subinterval \([t_i, t_{i+1}]\)

- Abscissas \(t_i\) are called knots or breakpoints, at which interpolant changes from one function to another

- Simplest example is piecewise linear interpolation, in which successive pairs of data points are connected by straight lines

- Although piecewise interpolation eliminates excessive oscillation and nonconvergence, it may sacrifice smoothness of interpolating function

- We have many degrees of freedom in choosing piecewise polynomial interpolant, however, which can be exploited to obtain smooth interpolating function despite its piecewise nature
Why Splines?

\[ f(x) = \frac{1}{1 + 25x^2} \]

Table: Six equidistantly spaced points in [-1, 1]

<table>
<thead>
<tr>
<th>x</th>
<th>( \frac{1}{1 + 25x^2} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.0</td>
<td>0.038461</td>
</tr>
<tr>
<td>-0.6</td>
<td>0.1</td>
</tr>
<tr>
<td>-0.2</td>
<td>0.5</td>
</tr>
<tr>
<td>0.2</td>
<td>0.5</td>
</tr>
<tr>
<td>0.6</td>
<td>0.1</td>
</tr>
<tr>
<td>1.0</td>
<td>0.038461</td>
</tr>
</tbody>
</table>

Figure: 5th order polynomial vs. exact function

http://numericalmethods.eng.usf.edu
Why Splines?

Figure: Higher order polynomial interpolation is dangerous

http://numericalmethods.eng.usf.edu
Spline orders

- **Linear spline**
  - Derivatives are not continuous
  - Not smooth

- **Quadratic spline**
  - Continuous 1\textsuperscript{st} derivatives

- **Cubic spline**
  - Continuous 1\textsuperscript{st} & 2\textsuperscript{nd} derivatives
  - Smoother
Linear Interpolation

Given \((x_0, y_0) (x_1, y_1) \ldots \ldots, (x_{n-1}, y_{n-1}) (x_n, y_n)\), fit linear splines to the data. This simply involves forming the consecutive data through straight lines. So if the above data is given in an ascending order, the linear splines are given by \((y_i = f(x_i))\)

Figure: Linear splines
Quadratic Spline

\[ f(x) = a_1x^2 + b_1x + c_1 \]
\[ f(x_0) \]

\[ f(x) = a_2x^2 + b_2x + c_2 \]
\[ f(x_1) \]

\[ f(x) = a_3x^2 + b_3x + c_3 \]
\[ f(x_2) \]

\[ f(x) = f(x_3) \]

Interval 1

\[ i = 0 \]

Interval 2

\[ i = 1 \]

Interval 3

\[ i = 2 \]

\[ i = 3 \]
Cubic Spline

- Spline of Degree 3

- A function $s$ is called a spline of degree 3 if
  - The domain of $s$ is an interval $[a, b]$.
  - $s$, $s'$ and $s''$ are continuous functions on $[a, b]$.
  - There are points $t_i$ (called knots) such that
    \[a = t_0 < t_1 < \ldots < t_n = b\]
    and $Q$ is a polynomial of degree at most 3 on each subinterval $[t_i, t_{i+1}]$. 
Cubic Spline ($4n$ conditions)

1. Interpolating conditions ($2n$ conditions).

2. Continuous 1st derivatives ($n-1$ conditions)
   - The 1st derivatives at the interior knots must be equal.

3. Continuous 2nd derivatives ($n-1$ conditions)
   - The 2nd derivatives at the interior knots must be equal.

4. Assume the 2nd derivatives at the end points are zero ($2$ conditions).
   - This condition makes the spline a "natural spline".
Hermite cubic Interpolant

**Hermite cubic interpolant**: piecewise cubic polynomial
interpolant with continuous first derivative

- Piecewise cubic polynomial with $n$ knots has $4(n - 1)$
  parameters to be determined
- Requiring that it interpolate given data gives $2(n - 1)$
equations
- Requiring that it have one continuous derivative gives $n$
  $- 2$ additional equations, or total of $3n - 4$, which still
leaves $n$ free parameters
- Thus, Hermite cubic interpolant is not unique, and
remaining free parameters can be chosen so that result
satisfies additional constraints
Spline example

- Determine natural cubic spline interpolating three data points \((t_i, y_i), i = 1, 2, 3\)

- Required interpolant is piecewise cubic function defined by separate cubic polynomials in each of two intervals \([t_1, t_2]\) and \([t_2, t_3]\)

- Denote these two polynomials by

\[
p_1(t) = \alpha_1 + \alpha_2 t + \alpha_3 t^2 + \alpha_4 t^3
\]

\[
p_2(t) = \beta_1 + \beta_2 t + \beta_3 t^2 + \beta_4 t^3
\]

- Eight parameters are to be determined, so we need eight equations
Example

- Requiring first cubic to interpolate data at end points of first interval \([t_1, t_2]\) gives two equations
  \[
  \alpha_1 + \alpha_2 t_1 + \alpha_3 t_1^2 + \alpha_4 t_1^3 = y_1
  \]
  \[
  \alpha_1 + \alpha_2 t_2 + \alpha_3 t_2^2 + \alpha_4 t_2^3 = y_2
  \]

- Requiring second cubic to interpolate data at end points of second interval \([t_2, t_3]\) gives two equations
  \[
  \beta_1 + \beta_2 t_2 + \beta_3 t_2^2 + \beta_4 t_2^3 = y_2
  \]
  \[
  \beta_1 + \beta_2 t_3 + \beta_3 t_3^2 + \beta_4 t_3^3 = y_3
  \]

- Requiring first derivative of interpolant to be continuous at \(t_2\) gives equation
  \[
  \alpha_2 + 2\alpha_3 t_2 + 3\alpha_4 t_2^2 = \beta_2 + 2\beta_3 t_2 + 3\beta_4 t_2^2
  \]
Example

- Requiring second derivative of interpolant function to be continuous at $t_2$ gives equation
  \[ 2\alpha_3 + 6\alpha_4 t_2 = 2\beta_3 + 6\beta_4 t_2 \]

- Finally, by definition natural spline has second derivative equal to zero at endpoints, which gives two equations
  \[ 2\alpha_3 + 6\alpha_4 t_1 = 0 \]
  \[ 2\beta_3 + 6\beta_4 t_3 = 0 \]

- When particular data values are substituted for $t_i$ and $y_i$, system of eight linear equations can be solved for eight unknown parameters $\alpha_i$ and $\beta_i$
Hermite vs. spline

- Choice between Hermite cubic and spline interpolation depends on data to be fit
- If smoothness is of paramount importance, then spline interpolation wins
- Hermite cubic interpolant allows flexibility to preserve monotonicity if original data are monotonic
Function Representation

- Put together minimal table, polynomial splines/fit
- Discrete error model
- Tradeoff table size/run time vs. accuracy
Embedded Processor Function Evaluation

--Dong-U Lee/UCLA 2006

- Approximate functions via polynomials
- Minimize resources for given target precision
- Processor fixed-point arithmetic
- Minimal number of bits to each signal in the data path
- Emulate operations larger than processor word-length
Function Evaluation

- Typically in three steps
  - (1) reduce the input interval \([a,b]\) to a smaller interval \([a’,b’]\)
  - (2) function approximation on the range-reduced interval
  - (3) expand the result back to the original range

- Evaluation of \(\log(x)\):

\[
\log(M_x \times 2^{E_x}) = \log(M_x) + E_x \times \log(2)
\]

where \(M_x\) is the mantissa over the \([1,2)\) and \(E_x\) is the exponent of \(x\)
Polynomial Approximations

- Single polynomial
  - Whole interval approximated with a single polynomial
  - Increase polynomial degree until the error requirement is met

- Splines (piecewise polynomials)
  - Partition interval into multiple segments: fit polynomial for each segment
  - Given polynomial degree, increase the number of segments until the error requirement is met
Computation Flow

- Input interval is split into $2^{B_x0}$ equally sized segments
- Leading $B_{x0}$ bits serve as coefficient table index
- Coefficients computed to
- Determine minimal bit-widths $\rightarrow$ minimize execution time
- $x_1$ used for polynomial arithmetic normalized over $[0,1)$
Approximation Methods

- Degree-3 Taylor, Chebyshev, and minimax approximations of \( \log(x) \)
Design Flow Overview

- Written in MATLAB
- Approximation methods
  - Single Polynomial
  - Degree-$d$ splines
- Range analysis
  - Analytic method based on roots of the derivative
- Precision analysis
  - Simulated annealing on analytic error expressions
Error Sources

- Three main error sources:
  1. Inherent error $E_\infty$ due to approximating functions
  2. Quantization error $E_Q$ due to finite precision effects
  3. Final output rounding step, which can cause a maximum of 0.5 ulp

- To achieve 1 ulp accuracy at the output, $E_\infty + E_Q \leq 0.5$ ulp
  - Large $E_\infty$
    - Polynomial degree reduction (single polynomial) or required number of segments reduction (splines)
    - However, leads to small $E_Q$, leading to large bit-widths
  - Good balance: allocate a maximum of 0.3 ulp for $E_\infty$ and the rest for $E_Q$
Range Analysis

- Inspect dynamic range of each signal and compute required number of integer bits
- Two’s complement assumed, for a signal \( x \):

\[
B_x
\]

- Integer Part   Fractional Part

\[
IB_x \quad FB_x
\]

- For a range \( x = [x_{\text{min}}, x_{\text{max}}] \), \( IB_x \) is given by

\[
IB_x = \left\lfloor \log_2(\max(|x_{\text{min}}|, |x_{\text{max}}|)) \right\rfloor + \alpha
\]

where \( \alpha = \begin{cases} 
1, & \text{mod}(\log_2(x_{\text{max}}), 1) \neq 0 \\
2, & \text{mod}(\log_2(x_{\text{max}}), 1) = 0
\end{cases} \)
Range Determination

- Examine local minima, local maxima, and minimum and maximum input values at each signal
- Works for designs with differentiable signals, which is the case for polynomials

![Graph showing range determination](image)

range = \([y_2, y_5]\)
Range Analysis Example

- Degree-3 polynomial approximation to $\log(x)$
- Able to compute exact ranges
- $IB$ can be negative as shown for $C_3$, $C_0$, and $D_4$: leading zeros in the fractional part

<table>
<thead>
<tr>
<th>Signal</th>
<th>Range</th>
<th>$IB$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_4$</td>
<td>$[0.00000,0.11006]$</td>
<td>-2</td>
</tr>
<tr>
<td>$D_3$</td>
<td>$[-0.40048,-0.29042]$</td>
<td>0</td>
</tr>
<tr>
<td>$D_2$</td>
<td>$[-0.29042,0.00000]$</td>
<td>0</td>
</tr>
<tr>
<td>$D_1$</td>
<td>$[0.69320,0.98362]$</td>
<td>1</td>
</tr>
<tr>
<td>$D_0$</td>
<td>$[0.00000,0.69320]$</td>
<td>1</td>
</tr>
<tr>
<td>$y$</td>
<td>$[0.00044,0.69363]$</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>$C_3$</th>
<th>$C_2$</th>
<th>$C_1$</th>
<th>$C_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0.11006</td>
<td>-0.40048</td>
<td>0.98362</td>
<td>0.00044</td>
</tr>
<tr>
<td>$IB$</td>
<td>-2</td>
<td>0</td>
<td>1</td>
<td>-10</td>
</tr>
</tbody>
</table>
Precision Analysis

- Determine minimal $FB$s of all signals while meeting error constraint at output
- Quantization methods
  - Truncation: $2^{-FB}$ (1 ulp) maximum error
  - Round-to-nearest: $2^{-FB-1}$ (0.5 ulp) maximum error
- To achieve 1 ulp accuracy at output, round-to-nearest must be performed at output
- Free to choose either method for internal signals: Although round-to-nearest offers smaller errors, it requires an extra adder, hence truncation is chosen
Error Models of Arithmetic Operators

- Let $\tilde{x}$ be the quantized version and $\epsilon_{\tilde{x}}$ be the error due to quantizing a signal $x$.
- Addition/subtraction

$$\tilde{z} = \tilde{x} \pm \tilde{y} = x \pm y + \epsilon_{\tilde{x}} + \epsilon_{\tilde{y}} + 2^{-FB_{\tilde{z}}}$$

$$\Rightarrow \epsilon_{\tilde{z}} = \epsilon_{\tilde{x}} + \epsilon_{\tilde{y}} + 2^{-FB_{\tilde{z}}}$$

- Multiplication

$$\tilde{z} = \tilde{x}\tilde{y}$$

$$= xy + a\epsilon_{\tilde{y}} + b\epsilon_{\tilde{x}} + \epsilon_{\tilde{x}}\epsilon_{\tilde{y}} + 2^{-FB_{\tilde{z}}}$$

$$\Rightarrow \epsilon_{\tilde{z}} = x\epsilon_{\tilde{y}} + y\epsilon_{\tilde{x}} + \epsilon_{\tilde{x}}\epsilon_{\tilde{y}} + 2^{-FB_{\tilde{y}}}$$
Precision Analysis for Polynomials

- Degree-3 polynomial example, assuming that coefficients are rounded to the nearest:

\[
\epsilon_D = 2^{-FB_{C_3}} + 2^{-FB_{D_4}}
\]
\[
\epsilon_D = \epsilon_D + 2^{-FB_{C_2}} + 2^{-FB_{D_3}}
\]
\[
\epsilon_D = \epsilon_D + 2^{-FB_{D_2}}
\]
\[
\epsilon_D = \epsilon_D + 2^{-FB_{C_1}} + 2^{-FB_{D_1}}
\]
\[
\epsilon_D = \epsilon_D + 2^{-FB_{D_0}}
\]
\[
\epsilon_D = \epsilon_D + 2^{-FB_{C_0}} + 2^{-FB_y} + \epsilon_\infty
\]

\[
\epsilon_D = \sum_{i=0}^{3} 2^{-FB_{C_i}} + \sum_{i=0}^{4} 2^{-FB_{D_i}} + 2^{-FB_y} + \epsilon_\infty
\]
Uniform Fractional Bit-Width

- Obtain 8 fractional bits with 1 ulp ($2^{-8}$) accuracy

$$
\epsilon_y \leq 2^{-F B_y} 
$$

$$
\Rightarrow \quad 2^{-8} \geq \sum_{i=0}^{3} 2^{-F B C_i} + \sum_{i=0}^{4} 2^{1-F B D_i} + 2 \times \epsilon_\infty
$$

- Suboptimal but simple solution is the uniform fractional bit-width (UFB)

$$
2^{-8} \geq 2^{2-UFB} + 2^{1-UFB} + 2^{3-UFB} \\
+ 2 \times 4.36706 \times 10^{-4}
$$

$$
\Rightarrow UFB = 13 \text{ bits}
$$
Non-Uniform Fractional Bit-Width

- Let the fractional bit-widths to be different
- Use adaptive simulated annealing (ASA), which allows for faster convergence times than traditional simulated annealing
  - Constraint function: error inequalities
  - Cost function: latency of multiword arithmetic
- Bit-widths must be a multiple of the natural processor word-length $n$
  \[ FB_x = n - IB_x + i \times n \quad \text{where} \quad i \in \mathbb{Z}^* \]
- On an 8-bit processor, if signal $IB_x = 1$, then $FB_x = \{ 7, 15, 23, \ldots \}$
Bit-Widths for Degree-3 Example

<table>
<thead>
<tr>
<th>Signal</th>
<th>8-bit Constrained</th>
<th>C Constrained</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B</td>
<td>IB</td>
</tr>
<tr>
<td>$C_3$</td>
<td>8</td>
<td>-2</td>
</tr>
<tr>
<td>$C_2$</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>$C_1$</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>$C_0$</td>
<td>8</td>
<td>-10</td>
</tr>
<tr>
<td>$D_4$</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>$D_3$</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>$D_2$</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>$D_1$</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>$D_0$</td>
<td>24</td>
<td>6</td>
</tr>
<tr>
<td>$y$</td>
<td>16</td>
<td>8</td>
</tr>
</tbody>
</table>

Total Bit-Width | Integer Bit-Width | Fractional Bit-Width
---|---|---

Shifts for binary point alignment
Fixed-Point to Integer Mapping

- Fixed-point libraries for the C language do not provide support for negative integer bit-widths
  → emulate fixed-point via integers with shifts
Multiplications in C language

- In standard C, a 16-bit by 16-bit multiplication returns the least significant 16 bits of the full 32-bit result → undesirable since access to the full 32 bits is required
- Solution 1: pad the two operands with 16 leading zeros and perform 32-bit by 32-bit multiplication
- Solution 2: use special C syntax to extract full 32-bit result from 16-bit by 16-bit multiplication

```c
int32_t z = (int32_t)(int16_t)x * (int32_t)(int16_t)y
```

More efficient than solution 1 and works on both Atmel and TI compilers
Automatically Generated C Code for Degree-3 Polynomial

```c
inline int16_t log_poly3_9bit(int16_t x) {
    static const int8_t c3 = 113;
    static const int16_t c2 = -26246;
    static const int16_t c1 = 32231;
    static const int16_t c0 = 29307;
    static const int32_t r = 131072;

    static int16_t d4, d3, d2, d1, y;
    static int32_t d0;

    d4 = (int16_t)(((int32_t)(int16_t)x
        + (int32_t)(int8_t)c3) >> 2);
    d3 = (int16_t)(((int16_t)(int16_t)d4
        + (int16_t)(int16_t)c2) >> 0);
    d2 = (int16_t)(((int32_t)(int16_t)x
        + (int32_t)(int16_t)d3) >> 9);
    d1 = (int16_t)(((int16_t)(int16_t)d2
        + (int16_t)(int16_t)c1) >> 0);
    d0 = (int32_t)(((int32_t)(int16_t)x
        + (int32_t)(int16_t)d1) << 3);
    y = (int16_t)(((int32_t)(int32_t)d0
        + (int32_t)(int32_t)c0
        + (int32_t)(int32_t)r) >> 18);

    return y;
}
```

- Casting for controlling multi-word arithmetic (inttypes.h)
- Shifts after each operation for quantization
- \( r \) is a constant used for final rounding
Automatically Generated C Code for Degree-3 Splines

```c
inline int16_t log_splines3_16bit(int16_t x) {
    static const int16_t c3_table[4] = { 31031, 16933, 10237, 6655 };
    static const int16_t c2_table[4] = { -31870, -20579, -14365, -10589 };
    static const int32_t c1_table[4] = { 2146092910, 1717372057, 1431343304, 1226958295 };
    static const int32_t c0_table[4] = { 10372, 479201753, 870732054, 1201767086 };

    static int8_t x0; static int16_t d4, d3, d1, c3, c2, x1, y; static int32_t d2, d0, c1, c0;

    x0 = x >> 13;
    x1 = x & 8191;

    c3 = c3_table[x0]; c2 = c2_table[x0];
    c1 = c1_table[x0]; c0 = c0_table[x0];

    d4 = (int16_t)(((int32_t)(int16_t)x1 * (int32_t)(int16_t)c3) >> 16);
    d3 = (int16_t)(((int32_t)(int16_t)x1 * (int32_t)(int16_t)c2) >> 0);
    d2 = (int32_t)(((int32_t)(int16_t)x1 * (int32_t)(int16_t)d4) >> 0);
    d1 = (int16_t)(((int32_t)(int32_t)d2 + (int32_t)(int32_t)c1) >> 16);
    d0 = (int32_t)(((int32_t)(int16_t)x1 * (int32_t)(int16_t)d1) << 1);
    y = (int16_t)(((int32_t)(int32_t)d0 + (int32_t)(int32_t)c0 + (int32_t)(int16_t)r) >> 16);

    return y;
}
```

- Accurate to 15 fractional bits ($2^{-15}$)
- 4 segments used
- 2 leading bits of $x$ of the table index
- Over 90% are exactly rounded less than $\frac{1}{2}$ ulp error
Experimental Validation

- Two commonly-used embedded processors
  - Atmel ATmega 128 8-bit MCU
    - Single ALU & a hardware multiplier
    - Instructions execute in one cycle except multiplier (2 cycles)
    - 4 KB RAM
    - Atmel AVR Studio 4.12 for cycle-accurate simulation
  - TI TMS320C64x 16-bit fixed-point DSP
    - VLIW architecture with six ALUs & two hardware multipliers
    - ALU: multiple additions/subtractions/shifts per cycle
    - Multiplier: 2x 16b-by-16b / 4x 8b-by-8b per cycle
    - 32 KB L1 & 2048 KB L2 cache
    - TI Code Composer Studio 3.1 for cycle-accurate simulation

- Same C code used for both platforms
Table Size Variation

- Single polynomial approximation shows little area variation
- Rapid growth with low degree splines due to increasing number of segments
NFB vs UFB Comparisons

- Non-uniform fractional bit widths (NFB) allow reduced latency and code size relative to uniform fractional bit-width (UFB)
Latency Variations

ATmega128

TMS320C64x
Code Size Variations

ATmega128

TMS320C64x
Code Size: Data and Instructions

Degree-2 Splines Code Size

Size [KB]

Precision [bits]

Upper part: Data

Lower part: Instructions

ATmega128
TMS320C64x
Comparisons Against Floating-Point

- Significant savings in both latency and code size

<table>
<thead>
<tr>
<th>Function</th>
<th>Atmel ATmega128 8-bit MCU</th>
<th>TI TMS320C64x 16-bit DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12-bit Precision (Degree-1)</td>
<td>16-bit Precision (Degree-2)</td>
</tr>
<tr>
<td>ln(x)</td>
<td>115</td>
<td>0.56</td>
</tr>
<tr>
<td>sin(x)</td>
<td>114</td>
<td>0.56</td>
</tr>
<tr>
<td>2^x</td>
<td>109</td>
<td>0.55</td>
</tr>
<tr>
<td>ln(x)</td>
<td>106</td>
<td>1.57</td>
</tr>
<tr>
<td>sin(x)</td>
<td>104</td>
<td>1.49</td>
</tr>
<tr>
<td>2^x</td>
<td>103</td>
<td>1.47</td>
</tr>
</tbody>
</table>
Application to Gamma Correction

- Evaluation of $f(x) = x^{0.8}$ on ATmega128 MCU

<table>
<thead>
<tr>
<th>Method</th>
<th>Simulated Latency</th>
<th>Measured Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cycles</td>
<td>Normalized</td>
</tr>
<tr>
<td>32-bit floating-point</td>
<td>10784</td>
<td>13.6</td>
</tr>
<tr>
<td>$2^{-8}$ fixed-point*</td>
<td>791</td>
<td>1</td>
</tr>
</tbody>
</table>

* Degree-1 splines

No visible difference