Speeding Up Floating-Point Division With In-lined Iterative Algorithms

Robert Enenkel, Allan Martin
IBM® Toronto Lab
Outline

- Hardware floating-point division
- The case for software division
- Software division algorithms
- Special cases/tradeoffs
- Performance results
- Automatic generation
Hardware Division

- PPC fdiv, fdivs

Advantages
- accurate (correctly rounded)
- handles exceptional cases (Inf, NaN)
- lower latency than SW

Disadvantages
- occupies FPU completely
- inhibits parallelism
Alternatives to HW division

- Vector libraries
  - MASS
  - higher overhead, greater speedup
- In-lined software division
  - low overhead, medium speedup
Rationale for Software Division

- Write SW division algorithm in terms of HW arithmetic instructions
  - *Newton's method or Taylor series*
- Latency will be higher than HW division
- But...SW instructions can be interleaved, so throughput may be better
- Requires enough independent instructions to interleave
  - *loop of divisions*
  - *other work*
Newton's Method

- To find $x$ such that $f(x) = 0$,
- Initial guess $x_0$
- $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$, $n=0, 1, 2,...$
- Provided $x_0$ is close enough
  - $f$ $x_n$ converges to $x$
  - $f$ It converges quadratically $|x_{n+1} - x| < c|x_n - x|^2$
  - $f$ Number of bits of accuracy doubles with each iteration

© Copyright IBM Corp. 2005
Newton's Method
Newton Iteration for Division

- For $1/b$, let $f(x) = 1/x - b$
- For $a/b$, use $a*(1/b)$ or $f(x) = a/x - b$
- Algorithm for $1/b$
  
  \[
  f \quad x_0 \sim 1/b \text{ initial guess}
  \]
  
  \[
  f \quad e_0 = 1 - b*y_0
  \]
  
  \[
  f \quad x_1 = x_0 + e_0*x_0
  \]
  
  \[
  f \quad e_1 = e_0*e_0
  \]
  
  \[
  f \quad x_2 = x_1 + e_1*x_1
  \]
  
  \[
  f \quad \text{etc...}
  \]
How Many Iterations Needed?

- **Power5 reciprocal estimate instructions**
  - $FRES$ (single precision), $FRE$ (double prec.)
  - $|relative\ error| \leq 2^{-8}$

- **Floating-point precision**
  - single: 24 bits
  - double: 53 bits

- **Newton iterations**
  - error: $2^{-16}, 2^{-32}, 2^{-64}, 2^{-128}$
  - single: 2 iterations for 1 ulp
  - double: 3 iterations for 1 ulp
  - +1 iteration for correct rounding (0.5 ulps)
Taylor Series for Reciprocal

- $x_0 \sim 1/b$ initial guess
- $e = 1 - b \cdot x_0$
- $1/b = x_0/(b \cdot x_0) = x_0 \cdot (1/(1-e))$
  
  $= x_0 \cdot (1 + e + e^2 + e^3 + e^4 + ...)$

- Algorithm (6 terms)
  
  $f\ e = 1 - d \cdot x_0$
  $f\ t_1 = 0.5 + e \cdot e$
  $f\ q_1 = x_0 + x_0 \cdot e$
  $f\ t_2 = 0.75 + t_1 \cdot t_1$
  $f\ t_3 = q_1 \cdot e$
  $f\ q_2 = x_0 + t_2 \cdot t_3$
Speed/Accuracy tradeoff

- IBM compilers have -qstrict/-qnostrict
  - qstrict: SW result should match HW division exactly
  - qnostrict: SW result may be slightly less accurate for speed
Exceptions

- Even when a/b is representable...
- 1/b may underflow
  - $f$ $a \sim b \sim$ huge, $a/b \sim 1$, $1/b$ denormalized
  - $f$ Causes loss of accuracy
- 1/b may overflow
  - $f$ a, b denormalized, $a/b \sim 1$, $1/b = \text{Inf}$
  - $f$ Causes SW algorithm to produce NaN
- Handle with tests in algorithm
  - $f$ Use HW divide for exceptional cases
Algorithm variations

- User callable built-in functions
  - \texttt{swdiv(a,b)}: double precision, checking
  - \texttt{swdivs(a,b)}: single precision, checking
  - \texttt{swdiv\_nochk(a,b)}: double, non-checking
  - \texttt{swdivs\_nochk(a,b)}: single, non-checking

- Accuracy of \texttt{swdiv}, \texttt{swdiv\_nochk} depends on \texttt{-qstrict/-qnostrict}

- \texttt{\_nochk} versions faster but have argument restrictions
# Accuracy and Performance

<table>
<thead>
<tr>
<th></th>
<th>Power5 speedup ratio</th>
<th>Power4 speedup ratio</th>
<th>Power5 ulps max error</th>
<th>Power4 ulps max error</th>
</tr>
</thead>
<tbody>
<tr>
<td>swdivs</td>
<td>1.07</td>
<td>1.05</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>swdivs_nochk</td>
<td>1.46</td>
<td>1.28</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>swdiv strict</td>
<td>1.05</td>
<td></td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>swdiv nostrict</td>
<td>1.50</td>
<td></td>
<td></td>
<td>1.5</td>
</tr>
<tr>
<td>swdiv_nochk strict</td>
<td>1.51</td>
<td></td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>swdiv_nochk nostrict</td>
<td>1.77</td>
<td></td>
<td>1.5</td>
<td></td>
</tr>
</tbody>
</table>
Automatic Generation of Software Division

- The `swdivs` and `swdiv` algorithms can also be automatically generated by the compiler.
- Compiler can detect situations where throughput is more important than latency.
Automatic Generation of Software Division

- In straight-line code, we use a heuristic that calculates how much FP can be executed in parallel.
  - \( f \) independent instructions are good, especially other divides.
  - \( f \) dependent instructions are bad (they increase latency).
Automatic Generation of Software Division

- In modulo scheduled loops software-divide code can be pipelined, interleaving multiple iterations
- Divides are expanded if divide does not appear in a recurrence (cyclic data-dependence)
Summary

- **Software divide algorithms**
  - user callable
  - compiler generated
- **Loops of divides**
  - up to 1.77x speedup
- **UMT2K benchmark**
  - 1.19x speedup