Simulation

Philip Leong
Computer Engineering Laboratory
School of Electrical and Information Engineering
The University of Sydney
1. Introduction to Reconfigurable Computing
   - what is reconfigurable computing, applications, areas for research
2. Abstractions for Implementation
   - microcode, map-reduce
3. Monte Carlo Simulation
   - uniform and Gaussian random number generation, Monte Carlo arithmetic
4. FPGA Architecture
   - floating-point FPGAs, process variation
Random number generators (RNGs) extensively used in cryptography and simulation.

Can be:
- Non-deterministic, i.e., true random number generator (TRNG)
- Deterministic, i.e., pseudo random number generator (PRNG)

Distribution can be arbitrary:
- Will describe uniform and Gaussian (or normal)

Monte Carlo simulation is a large class of problems using RNGs
Overview

- A true random number generator
- Gaussian random numbers
- Monte Carlo arithmetic
True Random Number Generator

› Introduction
› Ring Oscillator based noise source
› Alternating Sequence Generator
› Results
› Conclusion
Propose random number generators for these applications
- True RNG (RRNG) based on oscillator phase noise
A cryptographically secure random bit generator (CSRBG) is one which produces sequences for which there is no polynomial time algorithm which, on input of the first $l$ bits of the output sequences, can predict the $(l+1)$st bit of $s$ with a probability significantly greater than 0.5.
Applications

› Designed for embedded FPGA applications
  - Small
  - Fast

› For cryptographic applications
  - Secure
Physical random number generators

- Oscillator sampling
- Direct amplification of transistor/resistor noise
- Discrete time chaos
• One method to produce a real random number source in an FPGA is to sample a high frequency signal with a low quality low frequency clock.

• If the phase noise of the low frequency oscillator is of the same order as the period of the high frequency clock, output is quite random.
Ring Oscillator based noise source
Increased Randomness via Clock Doubling
Poker Test (ONS only)

- Checks that each m-bit string occurs the correct number of times
- Passed if the result is between 1.03 and 57.4

(iii) Poker test

Let $m$ be a positive integer such that \( \lceil \frac{n}{m} \rceil \geq 5 \cdot (2^m) \), and let $k = \lceil \frac{n}{m} \rceil$. Divide the sequence $s$ into $k$ non-overlapping parts each of length $m$, and let $n_i$ be the number of occurrences of the $i$th type of sequence of length $m$, $1 \leq i \leq 2^m$. The poker test determines whether the sequences of length $m$ each appear approximately the same number of times in $s$, as would be expected for a random sequence. The statistic used is

\[
X_3 = \frac{2^m}{k} \left( \sum_{i=1}^{2^m} n_i^2 \right) - k
\]

which approximately follows a $\chi^2$ distribution with $2^m - 1$ degrees of freedom. Note that the poker test is a generalization of the frequency test: setting $m = 1$ in the poker test yields the frequency test.

From Menezes, “Handbook of Applied Cryptography”
Poker Test (ONS only)
At this point, have a mediocre TRNG that doesn’t pass Poker test

Can turn into a good RNG by reducing sampling clock rate and passing through a parity filter to deskew non-uniform distribution (Tsoi, Leung and Leong, FCCM03)

- slow

How can we make a fast one?
• Selected among many stream ciphers because of its compact implementation

• Recent research has shown weaknesses in this cipher (but any other can be used)
Modified Alternating Step Generator
127 and 129 bit LFSRs based on primitive polynomials with approx equal 1 and 0 coefficients

- \( LFSR1(x) = x^{127} + x^{102} + x^{61} + x^{59} + x^{58} + x^{57} + x^{56} + x^{55} + x^{54} + x^{53} + x^{52} + x^{51} + x^{50} + x^{49} + x^{48} + x^{47} + x^{46} + x^{45} + x^{44} + x^{43} + x^{42} + x^{41} + x^{40} + x^{39} + x^{38} + x^{37} + x^{36} + x^{35} + x^{34} + x^{33} + x^{32} + x^{31} + x^{30} + x^{29} + x^{28} + x^{27} + x^{26} + x^{25} + x^{24} + x^{23} + x^{22} + x^{21} + x^{20} + x^{19} + x^{18} + x^{17} + x^{16} + x^{15} + x^{14} + x^{13} + x^{12} + x^{11} + x^{10} + x^{9} + x^{8} + x^{7} + x^{6} + x^{5} + x^{4} + x^{3} + x^{2} + x + 1 \)

- \( LFSR2(x) = x^{129} + x^{119} + x^{62} + x^{61} + x^{60} + x^{59} + x^{58} + x^{57} + x^{56} + x^{55} + x^{54} + x^{53} + x^{52} + x^{51} + x^{50} + x^{49} + x^{48} + x^{47} + x^{46} + x^{45} + x^{44} + x^{43} + x^{42} + x^{41} + x^{40} + x^{39} + x^{38} + x^{37} + x^{36} + x^{35} + x^{34} + x^{33} + x^{32} + x^{31} + x^{30} + x^{29} + x^{28} + x^{27} + x^{26} + x^{25} + x^{24} + x^{23} + x^{22} + x^{21} + x^{20} + x^{19} + x^{18} + x^{17} + x^{16} + x^{15} + x^{14} + x^{13} + x^{12} + x^{11} + x^{10} + x^{9} + x^{8} + x^{7} + x^{6} + x^{5} + x^{4} + x^{3} + x^{2} + x + 1 \)
Results for Virtex XCV300E-8 (including computer interface)
- Period 7.482ns, 129 slices (4%), 4 BRAM (12%)
- High frequency clock can be > 400 MHz (133 MHz used)
- Ring oscillator freq > 800 MHz

Reported results are for no clock doubler but doubled version passes tests as well

Passes Crush, NIST and Diehard tests
## NIST and Diehard Tests

<table>
<thead>
<tr>
<th>Test</th>
<th>P-value</th>
<th>Pass Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>0.145326</td>
<td>0.9900</td>
</tr>
<tr>
<td>Block Frequency</td>
<td>0.657933</td>
<td>0.9700</td>
</tr>
<tr>
<td>Cusum-Forward</td>
<td>0.383827</td>
<td>1.0000</td>
</tr>
<tr>
<td>Cusum-Reverse</td>
<td>0.867692</td>
<td>1.0000</td>
</tr>
<tr>
<td>Runs</td>
<td>0.289667</td>
<td>0.9700</td>
</tr>
<tr>
<td>Long Run</td>
<td>0.759756</td>
<td>0.9900</td>
</tr>
<tr>
<td>Rank</td>
<td>0.514124</td>
<td>0.9900</td>
</tr>
<tr>
<td>FFT</td>
<td>0.779188</td>
<td>1.0000</td>
</tr>
<tr>
<td>Aperiodic Templates</td>
<td>0.657933</td>
<td>0.9600</td>
</tr>
<tr>
<td>Periodic Templates</td>
<td>0.289667</td>
<td>0.9900</td>
</tr>
<tr>
<td>Universal</td>
<td>0.162606</td>
<td>1.0000</td>
</tr>
<tr>
<td>Approximate Entropy</td>
<td>0.924076</td>
<td>0.9900</td>
</tr>
<tr>
<td>Random Excursions</td>
<td>0.637119</td>
<td>0.9565</td>
</tr>
<tr>
<td>Serial1</td>
<td>0.534146</td>
<td>1.0000</td>
</tr>
<tr>
<td>Serial2</td>
<td>0.262249</td>
<td>1.0000</td>
</tr>
<tr>
<td>Lempel Ziv</td>
<td>0.616305</td>
<td>0.9900</td>
</tr>
<tr>
<td>Linear Complexity</td>
<td>0.637119</td>
<td>1.0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test</th>
<th>P-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Birthday Spacings</td>
<td>0.310619</td>
</tr>
<tr>
<td>Overlapping 5-Permutation (chisqr 66.743792)</td>
<td>0.994677</td>
</tr>
<tr>
<td>Overlapping 5-Permutation (chisqr 107.948832)</td>
<td>0.253086</td>
</tr>
<tr>
<td>Binary Rank (31x31)</td>
<td>0.155</td>
</tr>
<tr>
<td>Binary Rank (32x32)</td>
<td>0.080</td>
</tr>
<tr>
<td>Binary Rank (6x8)</td>
<td>0.051318</td>
</tr>
<tr>
<td>Bitstream</td>
<td>0.008018</td>
</tr>
<tr>
<td>OPSO</td>
<td>0.996754</td>
</tr>
<tr>
<td>OQSO</td>
<td>0.011809</td>
</tr>
<tr>
<td>DNA</td>
<td>0.050285</td>
</tr>
<tr>
<td>Steam Count-the-1</td>
<td>0.066896</td>
</tr>
<tr>
<td>Byte Count-the-1</td>
<td>0.040476</td>
</tr>
<tr>
<td>parking Lot</td>
<td>0.921990</td>
</tr>
<tr>
<td>Min. Distance</td>
<td>0.496703</td>
</tr>
<tr>
<td>3D Spheres</td>
<td>0.016095</td>
</tr>
<tr>
<td>Squeeze</td>
<td>0.456598</td>
</tr>
<tr>
<td>Overlapping Sums</td>
<td>0.080856</td>
</tr>
<tr>
<td>Runs up</td>
<td>0.053444</td>
</tr>
<tr>
<td>Runs down</td>
<td>0.738119</td>
</tr>
<tr>
<td>Craps</td>
<td>0.985720</td>
</tr>
</tbody>
</table>
Conclusions

› New RNG introduced
  - Combines weak RNG with high speed stream cipher to produce a physical noise source
  - Clock doubler increases oscillator phase noise
  - Small area, high output rate, good statistical properties
  - ASG makes a very low cost implementation, other stream ciphers can be used
Overview

› A true random number generator
› Gaussian random numbers
› Monte Carlo arithmetic
Ziggurat Gaussian Random Number Generator

› Introduction
› Ziggurat Method
› Hardware Architecture
› Results & Performance
› Conclusions
Normally distributed (Gaussian) random number generators (GRNGs) are used extensively in simulations:
- Monte Carlo simulation of financial derivatives
- Design of channel codes

Need to be:
- Fast
- Area efficient
- Accurate, even in the tail regions of the distribution
## Speed Comparison of GRNGs

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>FP32</th>
<th>Relative Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ziggurat</td>
<td>37804</td>
<td>1.00</td>
</tr>
<tr>
<td>Wallace (q=4)</td>
<td>33561</td>
<td>0.89</td>
</tr>
<tr>
<td>Monty-Python</td>
<td>26014</td>
<td>0.69</td>
</tr>
<tr>
<td>PPND7</td>
<td>15239</td>
<td>0.40</td>
</tr>
<tr>
<td>Ahrens-Dieter</td>
<td>10047</td>
<td>0.27</td>
</tr>
<tr>
<td>Mixture-of-Triangles</td>
<td>9653</td>
<td>0.26</td>
</tr>
<tr>
<td>Polar</td>
<td>9039</td>
<td>0.24</td>
</tr>
<tr>
<td>GRAND</td>
<td>8672</td>
<td>0.23</td>
</tr>
<tr>
<td>Hastings-ICDF</td>
<td>8344</td>
<td>0.22</td>
</tr>
<tr>
<td>Ratio-of-uniforms (Leva)</td>
<td>7757</td>
<td>0.21</td>
</tr>
<tr>
<td>PPND16</td>
<td>7574</td>
<td>0.20</td>
</tr>
<tr>
<td>Marsaglia-Bray</td>
<td>7256</td>
<td>0.19</td>
</tr>
<tr>
<td>Box-Muller</td>
<td>7142</td>
<td>0.19</td>
</tr>
<tr>
<td>Ratio-of-uniforms (Kinderman)</td>
<td>6772</td>
<td>0.18</td>
</tr>
<tr>
<td>Central-Limit</td>
<td>2953</td>
<td>0.08</td>
</tr>
<tr>
<td>Central-Limit-Stretched</td>
<td>2732</td>
<td>0.07</td>
</tr>
</tbody>
</table>

Figures courtesy of David Thomas.
For any pdf $f(x)$, generate a random point uniformly in the area under the curve.

The x value of the random point has the desired distribution.

Gaussian normal distribution

- $f(x) = \exp[-x^2/2]$
1. Generate a random integer $j$, $i = j \& 255$, $x = jw_i$

2. If $x < x_i$ return $x$. /* rectangle */

3. If $i = 0$ return an $x$ from the tail. /* tail */

4. If $[f(x_{i-1}) - f(x_i)] \leq f(x) - f(x_i)$ return $x$ /* wedge */
› **Rectangle:**
  - if (\(|j| < k_i\)) return \(jw_i\).
  - Occurs 98.5% of the time, no elementary functions

› **Tail:**
  - do {
    \[
    x = -\log(\text{UNI})/r; \quad y = -\log(\text{UNI});
    \]
    } while(y+y<x*x);
  
  return (x > 0) ? (r+x) : -(r+x);

› **Wedge:**
  - If \([f(x_{i-1})-f(x_i)]U < f(x)-f(x_i)\) return \(x\) /* \(f(x) = \exp[-x^2/2]\) */
  - Don’t generate output when above pdf curve \((P_{\text{accept}} = 0.993 \text{ for } n=256)\)
Hardware Architecture

› Stage 1-3: Computation of Rectangular region

› Stage 4: Iterative operation unit for handling Wedge and Tail regions

› Stage 5: Output selection

› Others: Optional On-Chip test module
A pseudorandom uniform RNG with slightly superior randomness than a LFSR with modest hardware cost

- Period $\approx 2^{88}$

```c
unsigned long s1, s2, s3, b;
double tau88 ()
{
    /* Generates numbers between 0 and 1. */
    b = (((s1 << 13) ~ s1) >> 19);
    s1 = (((s1 & 4294967294) << 12) ~ b);
    b = (((s2 << 2) ~ s2) >> 25);
    s2 = (((s2 & 4294967288) << 4) ~ b);
    b = (((s3 << 3) ~ s3) >> 11);
    s3 = (((s3 & 4294967280) << 17) ~ b);
    return ((s1 ~ s2 ~ s3) * 2.3283064365e-10);
}
```
To achieve high throughput, tail/wedge evaluation overlapped with rectangular region computation.
- Entry into FIFO1 every cycle.
- For wedge/tail, additional entry in FIFO2.
- OU takes data from FIFO2 and generates output in FIFO3
- Computation of tail and wedge.
- Throughput need only be about 1.5% of stage 1-3.
- Multiplexer joins the two streams
Operation Unit Architecture

- Optimised for polynomial evaluation
  - Register file
  - Coefficient ROM
  - 36-bit ALU
    - two adders
    - an 18x18 multiplier
  - Control logic with finite state machine
Polynomial Evaluation for Elementary Functions

› General polynomial is used to calculate the Elementary and Natural Logarithm Functions

\[ P_n(x) = C_n x^n + C_{n-1} x^{n-1} + \ldots + C_1 x + C_0 = \sum_{i=0}^{n} C_i x^i \]

› Horner’s method used (better numerical stability)
Range Reduction: Exponential Function

\[ e^X = 2^k e^x = e^{k \ln 2} e^x = e^{x + k \ln 2} \]

where \[ k = \left\lfloor x \log_2 e + 0.5 \right\rfloor \]

and \[-0.5 \ln 2 \leq x \leq +0.5 \ln 2\]

▸ Max relative error is \(2^{-15}\)
\[ \ln X = \ln(2^k x) = \ln x + \ln 2^k = \ln x + k \ln 2 \]

where \( X = 2^k x \) and \( 0.5 \leq x < 1 \)

- Use a left shifter to find the first set bit in \( X \)
- Max relative error is \( 2^{-15} \)
## Implementation

**GRNG on XC2VP30-6 and XC3S200-4 FPGAs**

<table>
<thead>
<tr>
<th></th>
<th>XC2VP30-6</th>
<th>XC3S200-4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SLICEs</strong></td>
<td>880 out of 13,696 (6%)</td>
<td>908 out of 1,920 (47%)</td>
</tr>
<tr>
<td><strong>Block RAMs</strong></td>
<td>5 out of 136 (2%)</td>
<td>4 out of 12 (33%)</td>
</tr>
<tr>
<td><strong>MULT18X18s</strong></td>
<td>2 out of 136 (13%)</td>
<td>2 out of 12 (16%)</td>
</tr>
<tr>
<td><strong>DCMs</strong></td>
<td>1 out of 8 (12%)</td>
<td>1 out of 4 (25%)</td>
</tr>
<tr>
<td><strong>Period of “CLK”</strong></td>
<td>5.88ns (170MHz)</td>
<td>6.106ns (163.8MHz)</td>
</tr>
<tr>
<td><strong>Period of “CLK2”</strong></td>
<td>11.76ns (85MHz)</td>
<td>12.21ns (81.9MHz)</td>
</tr>
</tbody>
</table>
Comparison with Previous Techniques

Comparisons of different noise generators implemented on XC2V4000-6

<table>
<thead>
<tr>
<th></th>
<th>Ziggurat</th>
<th>Wallace</th>
<th>Box-Muller</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>168MHz</td>
<td>155MHz</td>
<td>133MHz</td>
</tr>
<tr>
<td>SLICEs</td>
<td>891</td>
<td>770</td>
<td>2514</td>
</tr>
<tr>
<td>Block RAMs</td>
<td>4</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>MULT18X18s</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Throughput(M/sec)</td>
<td>168</td>
<td>155</td>
<td>133</td>
</tr>
</tbody>
</table>
The throughput of the implementation,

\[
\left( f_{\text{clk}} \text{ is } 170\text{MHz and } P_{\text{accept}} = 0.993 \right)
\]

Max Throughput = \( f_{\text{clk}} \times P_{\text{accept}} \)

= 169 million samples / second
χ²_{511} test
- test based on 512 bins spaced uniformly over [-8,8].
- statistic is 422.539.
- for a 95% level of confidence range, the critical value is 565.

Gaussian to uniform transformation
- the resulting samples pass all the tests in the DIEHARD testsuite.
INITIALISATION

$n$ is the size of the $w$ and $k$ tables.

$i = 0..n-1$ and $r = x_{n-1}$.

$w_0 = 0.5^{32} v / f(r); \quad k_0 = \lfloor 2^{32} r f(r) / v \rfloor$.

$w_i = 0.5^{32} x_i; \quad k_i = \lfloor 2^{32} (x_{i-1} / x_i) \rfloor$.

$\{f_i\}$, where $f_i = e^{-i^2/2}$.

$U(0,1)$ is a uniform random number generator in $(0,1)$.

REPEAT

Generate a signed random 32-bit integer $j$.

Set index: $i \leftarrow j \& (2^n - 1)$. Set $x \leftarrow j w_i$.

IF $|j| < k_i$ THEN RETURN $x$. /* rectangle */

IF $i = 0$ THEN /* tail */

DO

Generate iid uniform $(0,1)$ variates $u_1, u_2$.

$x \leftarrow -\ln(u_1) / r, y \leftarrow -\ln(u_2)$.

WHILE $u_2 + u_2 < u_1^2$.

RETURN $x > 0 ? (r + x) : -(r + x)$.

IF $(f_i + (f_{i-1} - f_i) U(0,1)) < e^{-x^2/2}$ /* wedge */

RETURN $x$.

UNTIL FALSE.
Conclusions

› An hardware architecture for Gaussian random number with high throughput (168.8 million random numbers).

› Decoupling of fast rectangular region computation from slow wedge & tail computation.

› Pipelined datapath for high speed.

› Finite state machine with sequential elementary function evaluation for efficient hardware utilization.
Overview

› A true random number generator
› Gaussian random numbers
› Monte Carlo arithmetic
Monte Carlo Arithmetic

› Motivation
› Theory
› Processor Implementation
› Conclusion
Reconfigurable computing proposed by Estrin in the 1960's

Application acceleration possible but a lot of work

Research focused on speed and power

Trying to look at using soft processors with custom arithmetic units to assist verification of numeric software

- can use existing infrastructure
- changes to software are not required
- provide a utility that would be hard to achieve using other technologies
IEEE FP not associative

- Using 8 significant digits

  - (11111113+-11111111)+7.5111111=9.5111111
  - 11111113+(-11111111+7.5111111)=10.000000
IEEE rounding errors are biased

\[ rp(x) = \frac{622 - x \cdot (751 - x \cdot (324 - x \cdot (59 - 4 \cdot x)))}{112 - x \cdot (151 - x \cdot (72 - x \cdot (14 - x)))} \]

Plot \( rp(x) - rp(u) \) for \( x = (u + \epsilon), \ldots, (u + 300\epsilon) \), \( u = 1.60632, \epsilon = 2^{-22} \)

- single precision IEEE
(a-b) is inaccurate when $a \approx b$
- The major source of loss of significance

e.g. Quadratic formula to solve $7169x^2-8686x+2631=0$

Intel processor, OSX, gcc 4.2.1 (ANSI C),

IEEE double: $6.06243866321686e-01$ $6.05361657461268e-01$

IEEE single: $6.06197e-01$ $6.05408e-01$
Standard Deviation Example

• Find standard deviation of \((i + 10e5)\) for \(i = 1\) to \(N\)

/* 1 pass */
for (i = 1; i <= N; i++)
{
    sum += x(i);
    sumsq += x(i)*x(i);
}
xbar = sum / N;
sigmasq = (sumsq - sum * xbar) / (N - 1);
sigma = sqrt(sigmasq);

/* 2 pass */
for (i = 1; i <= N; i++)
    sum += x(i);
xbar = sum / N;
for (i = 1; i <= N; i++)
    t += (x(i) - xbar) * (x(i) - xbar);
sigma = sqrt(t / (N - 1));

• \text{FLOAT} \ 32.25296783447265625000
• \text{2FLOAT} \ 36.80579757690429687500
• \text{DOUBLE} \ 36.80579664491269653581

• 2 pass more accurate if std \ll mean as it doesn’t square x(i)
In 1982 VSE instituted a new index initialized to a value of 1000.000
- Updated after each transaction
- Approx two years later it had fallen to 520

Cause: updated value was truncated rather than rounded
- Rounded calculation gave a value of 1098.892
› Runtime error analysis extremely slow
  - Common way to analyse rounding error: use double precision and hope it works!

› Fixed point computation
  - Error is bounded in fixed interval
  - Static analysis works very well

› Floating-point computation
  - Error depends on magnitude
  - Static analysis requires range to be bounded
  - Difficult to obtain a tight bound statically
FPGA

- Custom floating-point units can be used to do runtime checking
- In this talk, will describe our work on implementing Monte Carlo arithmetic (D.S. Parker UCLA) for runtime validation of sensitivity to FP rounding errors
Monte Carlo Arithmetic

› Motivation
› Theory
› Processor Implementation
› Conclusion
› Floating – Point number:

\[ x = -1^n \cdot f \cdot 2^e \]

› Modeling rounding error with random variable (machine precision \( p \) (=23 for single precision)):

\[ x = -1^n \cdot f \cdot 2^e + 2^{e-p} \xi, \xi = U[-0.5, 0.5] \]
Define floating point operation $x \odot y$ in terms of real operation $x \cdot y$ (where $\cdot \in \{+,-,\times,\div\}$)

Randomization: if $t$ is the floating point fraction precision, $e$ is exponent of $x$ and $\xi = U[-0.5,0.5]$

$$\text{randomize}(x) = \begin{cases} x & \text{if exact within } t \text{ digits} \\ x + 2^{e+1-t} \xi & \text{otherwise} \end{cases}$$

Random rounding:

$$\text{random\_round}(x) = \text{round}(\text{randomize}(x))$$

Monte Carlo Arithmetic:

$$x \odot y = \text{round}(\text{randomize}(x) \cdot \text{randomize}(y))$$

Results different each time program is run $\Rightarrow$ Monte Carlo simulation
Standard error $\sigma/\sqrt{n}$ gives a measure of the error in the mean.

Notice convergence to the exact sum value 9.5111111.
Zero expected rounding error

\[
rp(x) = \frac{622 - x \cdot (751 - x \cdot (324 - x \cdot (59 - 4 \cdot x)))}{112 - x \cdot (151 - x \cdot (72 - x \cdot (14 - x)))}
\]

Plot \( rp(x) - rp(u) \) for \( x = (u + \epsilon), \ldots, (u + 300\epsilon) \), \( u = 1.60632, \epsilon = 2^{-22} \)

- single precision IEEE
- histogram for MCA

Figure: D.S. Parker UCLA
For large values most of the digits of the result will be different.
Monte Carlo Arithmetic

› Motivation
› Theory
› Processor Implementation
› Conclusion
Developed a source-to-source compiler based on CIL

Intercepts all floating-point calls and redirects to a MCA library
- Library can be software or hardware
- Software version implemented using arbitrary precision arithmetic and supports float and double

User need only modify code to tell MCA analysis what the outputs are
Comparison of Algorithms

- Linpack vs Gaussian elimination
- Tests show linpack achieves better accuracy for any virtual precision
MCA Co-Processor

- MCA Floating point unit implemented as co-processor
  - Simplify implementation of basic operations by using multiple floating point operations
  - Co-processor linked to 32-bit RISC processor through serial communications bus
  - Software access – can perform both IEEE:754 floating point operations and MCA operations and analyze results using simple software routines
  - High level synthesis and C-to-RTL tools used to speed up design process
MCA Addiition

- Addition performed in terms of the *inexact* function:

\[
inexact(x \pm y) = \text{round}(inexact(x) \pm inexact(y)) \\
= \text{round}((x + \xi_x) \pm (y + \xi_y)) \\
= \text{round}(x \pm y + \xi)
\]

- Define \( \xi \) as perturbation applied to the operation, can be implemented as a floating point number where:

\[
|\xi| = \beta^{e_x-t}(|\xi_x| \pm |\xi_y|\beta^{-(e_x-e_y)}) \\
\xi_x, \xi_y \in U[-\frac{1}{2}, \frac{1}{2})
\]

- \( \xi \) generated by first generating random fixed point value, applying required exponent and sign and normalizing the value using bitwise operations
MCA Co-Processor: Basic Operators

MCA operations are simplified:

- Perform single MCA operation using multiple IEEE-754 FP operations, as shown for addition.
- Perturbation value $\xi (X_i)$ is generated using RNGs and formatted as a FP value.
- Standard FP operation performed.
- Perturbation applied to result using addition/subtraction.
Multiplication operation is also performed in terms of the inexact function:

\[
\text{inexact}(xy) = \text{round}(\text{inexact}(x)\text{inexact}(y)) \\
= \text{round}((x + \xi_x)(y + \xi_y)) \\
= \text{round}(xy + x\xi_y + y\xi_x + \xi_x\xi_y)
\]

\(\xi\) value expanded as follows:

\[
\xi = \beta^{e_x + e_y - t}[m_x\xi_y + m_y\xi_x + \xi_x\xi_y \beta^{-t}]
\]

\(\xi_x, \xi_y \in U[-\frac{1}{2}, \frac{1}{2}]\)

Final term shifted right by 2t places so ignored.
Final multiplication operation:

\[ xy = \text{round}(xy + \xi) \]

\[ |\xi| = \beta^{e_x + e_y - t}[m_x|\xi_y| + m_y|\xi_x|] \]

Operation also implemented using multiple FP operations.
\[
x \div y = \text{round}\left(\frac{\text{inexact}(x)}{\text{inexact}(y)}\right)
\]

\[
= \text{round}\left(\frac{x + \xi_x}{y + \xi_y}\right)
\]

› Division cannot be simplified to give a combined $\xi$

› Also means need higher precision
MCA Co-Processor: System Overview

- Co-Processor uses IEEE-754 core to perform FP operations.
- Perturbation values generated using RNGs, then passed to FP core as inputs.
- Configuration register used to store precision value, t.
- Precision of MCA operations can be modified at any time to perform Variable precision MCA.
Co-Processor linked to Xilinx μBlaze Processor through Fast Simplex Link (FSL).

Co-Processor and bus interface implemented using AutoESL (C-to-RTL).
Co-Processor Results

- Implemented with Xilinx XC5VLX110T at 62.5MHz using ISE version 13.2

- Testing platforms:
  - Test group 1 (HW-FP): Test routines run for standard IEEE-754 floating point operations as control, no MCA applied.
  - Test group 2 (SW-MCA): Test routines run with MCA implemented in software routines only.
  - Test group 3 (HW-MCA): Test routines run with MCA performed using variable precision MCA co-processor.
Approx 65% more area and 150% slower (non-pipelined)

Further optimizations include improved communications and further investigations into pipelined implementation.

<table>
<thead>
<tr>
<th></th>
<th>Slice Registers</th>
<th>Slice LUT</th>
<th>LUT-FF Pairs</th>
<th>Slices</th>
<th>Max Freq.</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE:754 FPU</td>
<td>762</td>
<td>1127</td>
<td>1336</td>
<td>408</td>
<td>89 MHz</td>
<td>9.89</td>
</tr>
<tr>
<td>MCA FPU</td>
<td>1479</td>
<td>3313</td>
<td>2281</td>
<td>670</td>
<td>89 Mhz</td>
<td>3.86</td>
</tr>
<tr>
<td>MCA Overhead</td>
<td>94%</td>
<td>193%</td>
<td>70%</td>
<td>64%</td>
<td>0%</td>
<td>2.5X</td>
</tr>
</tbody>
</table>
Statistical testing:

- Basic operators compared against SW implementation developed by Parker.

- Statistical results show correct implementation of basic operators and mean, standard deviation and standard error of results correlate with Parker’s implementation.

Detection of catastrophic can be performed using co-processor

Analysis of relative standard deviation used to differentiate between stable algorithms and algorithms susceptible to round-off error.
Knuth Test $x=11111113.0$, $y=11111111.0$ $z=7.5111111$

\[ u = [(x + y) + z] - [x + (y + z)] \]
Testing shows round-off error from MCA Co-Processor is random and un-correlated:
Statistical Results (cont)

**Significant Figures v. Precision**

**Relative Standard Deviation v. Precision**
Results of performance testing (FSL):

Performance Results

<table>
<thead>
<tr>
<th>Test Type</th>
<th>IEEE</th>
<th>MCA - Hardware</th>
<th>MCA - Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>MFLOPS</td>
<td>1.5</td>
<td>0.5</td>
<td>0.0</td>
</tr>
<tr>
<td>Cosine</td>
<td>1.0</td>
<td>0.75</td>
<td>0.5</td>
</tr>
<tr>
<td>Knuth</td>
<td>2.0</td>
<td>1.5</td>
<td>0.25</td>
</tr>
<tr>
<td>Kahan</td>
<td>0.5</td>
<td>0.25</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Monte Carlo Arithmetic

› Motivation
› Theory
› Processor Implementation
› Conclusion
Conclusion

› MCA can be used to understand rounding errors
› Implementations of MCA in software slow
› Presented floating-point unit which can perform either MCA or IEEE754 single precision floating-point operations
  - Estimate rounding error with minimal impact on performance
  - Quantitatively understand sensitivity of programs to rounding
Future Work

› Performance improvements
  - AXI4 and ARM
  - Pipelining of inputs

› Develop a family of accelerated processors for different runtime validation tasks
  - Sensitivity to rounding
  - Range analysis
  - Precision estimation

› Hopefully this work will
  - Allow existing programs to be easily validated
  - Enable aggressive optimisation of floating point wordlengths in reconfigurable computing systems
References


