z13™ SIMD Deep Dive

Dr. habil. Silvia M. Mueller
STSM HW Development / Arithmetical Engines
Member of the IBM Academy of Technology

Large Systems Update 2015
What is Different?

- **z196**: Transparent increase of capacity & performance
- **zEC12**: Transparent increase of capacity & performance
- **z13**: Transformational Enhancements for Arithmetic
Motivation: Performance

• System z is already the fastest at 5 GHz for the past couple generations
  • Frequency “saturated”
  • Newer technology gives us more gates on the chip but not higher frequency (power, cooling)

→ Performance Increase has to be achieved by other means

• Parallelism is the future
  • Simultaneous Multi-Threading – SMT
  • Single Instruction Multiple Data – SIMD

• Huge performance gains possible through parallelism

• Got to be smarter and figure out best ways to utilize parallelism
Outline

• Key features

• SIMD concept

• Acceleration through SIMD
  • Integer processing
  • String processing
  • Binary floating-point
  • Cobol floating-point
  • 128-bit floating-point

• Summary

RSA codes
Java / Cobol / PL1 string, XMLSS, Cognos
ILOG, SPSS, analytics, mobile codes
traditional codes (e.g., insurance)
emerging, big data analytics
z13 SIMD / Vector and Floating-Point Units (VFU)

- **Enhancements over zEC12**: ~5x area
- **Faster specialized engines → reduced latency**
  - DFX: decimal fixed-point (BCD) add / sub
  - Divide / square-root engines
- **SIMD unit + more registers**
  - Increased parallelism
  - E.g.: string engine
  - Higher throughput & shorter latency
- **Two parallel pipelines**
  - Twice the number of parallel engines
  - Capable to start 2 VFU instructions per cycle
  - Higher throughput
Why so Many Engines?

**zEC12 FPU**
- 1 floating-point pipeline
- Starting at most 1 op per cycle
- Long-running instructions block the whole pipeline (e.g.: divide 33-170 cycles)

**z13 VFU**
- Starting 2 ops per cycle
- Non-blocking pipelines, new instructions can be started while divides are ongoing (up to 2 binary + 2 decimal divides)
- Fast operations can overtake slower ones
- Higher throughput and shorter execution time
- SMT helps to increase available instruction level parallelism
z13 SIMD and Floating-Point Units

**Supported data types** – all enhanced on z13

- Binary Integers – unsigned and two’s complement
  - Decimal Integers – packed decimals (BCD)
- Floating-point – z Systems supports more types than any other platform

<table>
<thead>
<tr>
<th>Precision</th>
<th>Single precision (SP)</th>
<th>Double precision (DP)</th>
<th>Quad precision (QP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32b</td>
<td>32-bit</td>
<td>64-bit</td>
<td>128-bit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Radices</th>
<th>Business analytics</th>
<th>Cobol</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Binary (BFP)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Decimal (DFP)</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Hexadecimal (HFP)</td>
<td></td>
</tr>
</tbody>
</table>

**Character Strings**

- Character width: 8, 16, and 32 bit
- Null terminated (C) and length based (Java)
SIMD Concept
**SIMD – Single Instruction Multiple Data**

- **Old:** Single Instruction Single Data (64b)

  ![Old SIMD Diagram]

- **New:** Single instruction operates on multiple data in parallel

  ![New SIMD Diagram]
SIMD – Single Instruction Multiple Data

- Each register contains multiple data elements of a fixed size
  - Byte, Half-word, Word, Double-word, Quad-word
  - The collection of elements in a register is also called a **vector**
  - Field in the instruction word specifies data format type

- 128b wide vector

  ![Diagram of 128b wide vector]

  - 16x Byte
  - 8x Half-word
  - 4x Word
  - 2x Double-word
  - 1x Quad-word
SIMD Processing

• Benefits
  • Explicit data parallelism
    – Easier to exploit than instruction level parallelism
  • Significantly smaller code size
  • Improved execution efficiency

• Lessons learned from Power
  • Fast unaligned loads from memory
  • Fast transfer between vector & scalar code
  • High throughput, low latency interconnect within VFU
  • Efficient scatter and gather operations
SIMD Hardware Accelerator

Three distinct data types
- Integer: 16xB, 8xHW, 4xW, 2xDW, 1xQW
- String
- Binary Floating-Point: DP only (2xDP)

Implementation
- New SIMD FXU
- New string engine
- Enhancing BFU for SIMD

Exploitation
- String processing for Cognos, XMLSS in Cobol, PL1, Java, C/C++
- Analytics workloads like ILOG, SPSS & Mobile codes
More Execution Pipes Need More Data

- Quadrupled the Floating-Point registers FPRs to create the Vector Register file
  Overlay of register files: easier to mix scalar & vector code

- VRs contains floats, integers, and strings
  Fast exchange between different data types
Accelerating Integer Processing
Binary Integer Support

**zEC12** (grey) vs. **z13** (grey + purple)

- **GPR**: 16 x 64bit
- **VR**: 32 x 128bit

- **FX0A**
  - divide
- **FX0B**
  - divide
- **FX1A**
- **FX1B**

64b Integer
- Increased from 2x engines to 8x engines (4x scalar, 2x2 SIMD)

16b Integer
- Increased from 2x engines to 20x engines (4x scalar, 2x8 SIMD)

© 2015 IBM Corporation
Binary Integer SIMD Engine

- **Latency**: 3-5 cycles
- **Supported data width**
  - 8b, 16b, 32b, 64b, 128b
- **Instructions (basic)**
  - Add, sub, compare, mul
  - Shift, rotate, splat, permute
  - Logical: and, or, nor, xor
- **Complex instructions**
  - Galois multiply to accelerate CRC (cyclic redundancy coding) and check sum
  - **Check sum**: computes $5 \times 32b$ add modulo $2^{32}$ per cycle, per pipe
  - **CRC**: 64 bit CRC assists computes $AxB + CxD + E$ of 64 bit in carryless form

<table>
<thead>
<tr>
<th></th>
<th>8b</th>
<th>16b</th>
<th>32b</th>
<th>64b</th>
<th>128b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add, sub, compare</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Min, max</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>b</td>
</tr>
<tr>
<td>Galois field mul (&amp; add)</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Multiply (&amp; add)</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector sum across (adding 2, 4 elements)</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>
Integer Multiply & Divide

- **Multiply for RSA codes: 256b integer multiply**
  - With SIMD exploitation 7x faster than best scalar code on z13
    - 2x from hardware, SIMD integer multiply instructions
    - 3.5x from software, enhanced algorithm
  - New RSA code integrated in JAVA

- **Integer Divide**
  - New divide algorithm computing the quotient 3 bit per cycle
  - Two, stand-alone, non-blocking engines in FXU
  - Improvement over zEC12
    - Speedup 1.7x
    - Throughput 3.4x
  - Can be used for fast, divide based hash functions
String Support – Exceptionally Good Example

Case conversion: String.toUpperCase()

MIN: 2.5x  Max: 58.8x  Avg.: 37x

Performance information was determined in a controlled IBM environment. Results may vary.
SIMD String Support

- **String Engine** (4-cycle)
  - Element size: byte / halfword / word
  - Operation with / without end-of-string checking (C vs Java)

- **String Search Operations**
  - Find equal / not equal → element wise compare of two strings
  - Find any equal → find whether two strings have any character in common
  - Range compare → compares string against several ranges

- **Load / store for string processing**
  - For known length strings (Java) & null terminating character strings (C)
  - With arbitrary byte / halfword boundary

- **Language support:** Java, C/C++, Cobol, PL1
SIMD String Support

• **Big comparator array**
  - Supporting strings with 16x8b, 8x16b, 4x32b
  - Comparators dynamically re-arranged to match required width

• **Very high parallelism for small data types**

**Byte string**
- 16 x 16
- Compares 256

**Half word string**
- 8 x 8 = 64
- Compares

**Word string**
- 4 x 4 = 16
- Compares
String Support: Vector Find Any Element Equal

Vector register: String (here halfwords)

Byte index          0    2    4    6    8    10   12   14
‘z’       ‘N’     ‘e’     ‘x’    ‘t’    0    ‘#’    ‘#’

Compare Vector
  ‘a’
  ‘e’
  ‘i’
  ‘y’
  ‘x’
  ‘#’
  ‘*’
  ‘*’

OR-ing all range checks per element

Result type: Mask
  00   00   FF   FF   00   00   FF   FF

zEC12:
Depending on coding
Up to 8x6 HW compares
→ 48 instructions

z13:
Single instruction
High parallelism
SIMD String Support

- In the past, loading or storing to a string could result in exceptions

```
Hello World
```

- **New Load / Store instructions**
  - With Length
    - Java strings
      - Load / store specified number of bytes
    - C strings
      - Specifies block size (64B, 128B, 256B, 2K, 4K, …)
  - To Block Boundary

```
? ? ? ?
```

Location not accessible

Storage

New Loads

Normal Loads

Our Mousetrap
You don’t want to touch: could cause prgm intpt

Are exact and will not set off an alarm
Accelerating Binary Floating-Point
Improved Binary Floating-Point

• **Emerging Workload: Telematics**
  • Tightly connecting DB accesses with numerical calculations on geospatial points
  • Intensive use of binary floating-point and math functions (\(\tan\), \(\arctan\), \(\log\), \(\exp\), …)

• **Performance results**
  • Vector **math library** functions (MASS lib)
    - Average speedup of 8x
      - \(v\cos\) 8.3x
      - \(v\exp\) 8.7x
      - \(v\log\) 10.7x
    - New algorithms and exploiting new SIMD hardware
  
  • **ILOG CPLEX** example
    - Throughput improvement up to 2.9x

  (# cycles per vector element: z13 with MASS lib relative to zEC12 with XLC runtime lib)
Improved Binary Floating-Point Latencies

<table>
<thead>
<tr>
<th></th>
<th>z196 / zEC12</th>
<th>z13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add, sub, mul,</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>FMA, convert</td>
<td></td>
<td></td>
</tr>
<tr>
<td>compare</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>Divide 32b</td>
<td>33</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>27</td>
</tr>
<tr>
<td>Divide 64b</td>
<td>39</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>53</td>
<td>36</td>
</tr>
<tr>
<td>SQRT 32b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQRT 64b</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BFU blocked,
No other floating-point instruction can be started

- More than 2x speedup on some BFP codes
- Due to improved BFU design
  - Faster compare, divide, sqrt
  - Special engine for divide/sqrt allows issue under slow operations
  - Available even on old binaries
- Efficient execution of parallelism “provided” by SIMD, tuning, and larger register file
Loop Optimization to Increase Instruction Level Parallelism

For (i=1, i<1025, i++)
{  s += a[i] * b[i]; }

For (i=1, i<1025, i+=8)
{  s0 += a[i] * b[i];
    s1 += a[i+1] * b[i+1];
    s2 += a[i+2] * b[i+2];
    ...
    s7 += a[i+7] * b[i+7];
}  
s = s0 + s1 + s2 + ... + s7;

Hiding 8-cycle BFU latency
Requires 3*8=24 registers

zEC12 has only 16 FPRs
→ BFU cannot be fully utilized
BFU Code with Very High Register Pressure

### zEC12 / Scalar Code
- **Register file**: FPR 16 x 64b
- **BFU**: 1x, ~50% utilized
- **Code**: Highly tuned
  - Loop unrolling

### z13 / Scalar Code
- **Register file**: FPR 16 x 64b
  - 32 x 64b scalar
  - 2x registers
- **BFU**: 2x, ~50% utilized
- **Code**: Recompile & tuning
  - Access to 2x registers

### z13 / Vector Code
- **Register file**: VRF 32x128b
- **BFU**: 2x, ~100% utilized
- **Code**: tuning & recompile
  - MASS / ATLAS libs
  - Vectorizing JAVA

< 2x speedup
Cobol Floating-Point
Improved HEX Floating-Point Performance

- **Default floating-point type in Cobol**
  - 32b / 64b → 6 / 14 fraction digits

- **Improved latency and throughput**
  - Special engine for divide / sqrt
    - 1.7 – 2x faster than zEC12
    - Allows issue under divide / sqrt
  - 2 engines each: BFU & Divide

- **Transparent – no recompile needed**
  - No SIMD support & only „old“ register file

- **Traditional Workload: two German insurance companies**
  - Home grown COBOL code, 64b floating-point
  - Observation on zEC12: 50% of CPU cycles spent in Divide

### Table

<table>
<thead>
<tr>
<th>Hex FP</th>
<th>z196 / zEC12</th>
<th>z13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add, sub, compare, mul, mul-add, convert</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Divide 32b</td>
<td>39</td>
<td>19</td>
</tr>
<tr>
<td>Divide 64b</td>
<td>55</td>
<td>30</td>
</tr>
<tr>
<td>SQRT 32b</td>
<td>41</td>
<td>21</td>
</tr>
<tr>
<td>SQRT 64b</td>
<td>68</td>
<td>39</td>
</tr>
</tbody>
</table>

BFU blocked

20+% speedup expected on z13
128b Floating-Point
128b Floating-Point for Big Data Analytics

- With growing problem size, numerical sensitivities are magnified
  - Degrading stability of the algorithms and/or speed of convergence
  - Examples: ILOG, SPSS, risk assessment codes

- **Use case: ILOG**
  - Switch critical loops from 64b to 128b
  - 15-30% faster convergence
  - In production at big bank in Europe

- **z Systems is only platform w/ hardware support for 128b floating-point**

**z196 / zEC12**
- Blocking, high latencies (35-170 cycles)

**z13: novel HW solution**
- Add / multiply: 3-4x faster
  - 12x throughput
- Divide / Sqrt: 3x faster
  - 6x throughput
- FP 64b ops can be issued while 128b arithmetic is ongoing

⇒ Very attractive to enhance BFP 64b code with BFP 128b code elements
z13 Arithmetical Engines

- **Significant enhancements to for all data types**
  - Faster / specialized engines
  - SIMD support
  - Multiple parallel engines
  - Non-blocking, starting 2 VFU ops + 4 FXU ops per cycle
  - Quadrupled number of registers

- **Performance increase**
  - Increased instruction throughput
  - Reduced execution latency
  - Increased exploitable data parallelism

- **Speedup strongly depends on the application & coding style**
Thank you!

Dr. Silvia M. Mueller (smm@de.ibm.com)