The IBM z13 SIMD Accelerators for Integer, String, and Floating-Point
z Systems - Processor Roadmap

**z10**
- 2/2008
- Workload Consolidation and Integration Engine for CPU Intensive Workloads
- Decimal FP
- Infiniband
- 64-CP Image
- Large Pages
- Shared Memory

**z196**
- 9/2010
- Top Tier Single Thread Performance, System Capacity
- Accelerator Integration
- Out of Order Execution
- Water Cooling
- PCIe I/O Fabric
- RAIM
- Enhanced Energy Management

**zEC12**
- 8/2012
- Leadership Single Thread, Enhanced Throughput
- Improved out-of-order
- Transactional Memory
- Dynamic Optimization
- 2 GB page support
- Step Function in System Capacity

**z13**
- 1/2015
- Leadership System Capacity and Performance
- Modularity & Scalability
- Dynamic SMT
- Supports two instruction threads
- SIMD
- PCIe attached accelerators (XML)
- Business Analytics Optimized

Leadership Single Thread, Enhanced Throughput
Improved out-of-order
Transactional Memory
Dynamic Optimization
2 GB page support
Step Function in System Capacity

© 2015 IBM Corporation
z13 Continues the CMOS Mainframe Heritage Begun in 1994

* MIPS Tables are NOT adequate for making comparisons of z Systems processors. Additional capacity planning required
** Number of PU cores for customer use

© 2015 IBM Corporation
z13 Core changes

- 1Q 2015 GA
- 8 cores per die
- 24 CP chips in system
- 192 cores x 2 threads => 384 threads
- Grow LSPR 8way by 11%, up to 1.4x perf with SMT2

- ST -> 2 way SMT
- Double the instruction fetch, decode, dispatch bandwidth
- Frequency hit (5.2 GHz z196 -> 5.6 GHz zEC12 -> 5.0 GHz z13)
- Double the number of execution units
  - 2 FXU writers, 2 FXU cc,
  - 2 BFU, 2 DFU
    - Quad precision is faster, Divide is faster
  - 2 SIMD units
  - 2 DFX – accelerate SS decimal
z13 Instr / Execution Dataflow

- Instr decode/ crack / dispatch / map
- IssQ side0
- Branch Q
- IssQ side1
- GR 0
- LSU pipe 0
- FXU* 0a
- FXU 0b
- VFU0
- Vector0 / FPR0 register
- 128b string/int SIMD0
- BFU0
- DFU0
- VFU1
- Vector1 / FPR1 register
- 128b string/int SIMD1
- BFU1
- DFU1
- GR 1
- LSU pipe 1
- FXU 1a
- FXU 1b
- D$

*FXa pipes execute reg writers and support b2b execution to itself
FXb pipes execute non-reg writers and non-relative branches (needs 3w AGEN)

Additional instruction flow for higher core throughput
Additional execution units for higher core throughput
New arch registers / execution units to accelerate business analytics workloads
Performance

- Technology is not getting any faster
- System z is already the fastest at 5 GHz for the past couple generations
- 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
SIMD – Single Instruction Multiple Data

Old road

New super highway
SIMD – Single Instruction Multiple Data

OLD

One 64 bit operation

VERSUS many operations up to 128 bit
such as Sixteen 8 bit operations

NEW
Different size data types – all 128 bit wide

- 16 x 8, 8 x 16, 4 x 32, 2 x 64 or 1 x 128 vs 1x64b
Datatypes

1.65438A890167 \times 2^{(-33)}

-561

1.534,678,432 \times 10^{(-32)}

"Hello World!"

"To be, or not to be,"

© 2015 IBM Corporation
Data Types Supported

- Integers – unsigned and two’s complement
- Floating-point – System z supports more types than any other platform
  - 32 (Single Precision – SP),
  - 64 (Double Precision – DP), and
  - 128 bit (Quad Precision – QP)
  - Radices of 2 (Binary Floating-Point – BFP),
  - 10 (Decimal – DFP),
  - 16 (Hexadecimal – HFP)
- Character Strings
  - Characters 8, 16, and 32 bit
  - Null terminated (C) and Length based (Java)
Z13 SIMD accelerates:

- Integer
- Binary Floating-Point Double Precision – BFP DP
  - Also indirectly quad precision
- Strings
SIMD Integer

- z13 has 2 new SIMD execution pipelines each 128 bits wide
- 16 x 8 bit / 8 x 16 bit / 4 x 32 bit / 2 x 64 bit / 1 x 128 bit
- Add, Subtract, Compare
- Logical operations
- Shifts
- Min / Max / Absolute Value
- Select
- CRC / Checksum
SIMD Integer - types

- Byte, Halfword, Word, Doubleword, Quadword in 128 bit register

128b wide vector:
16xB, 8xHW, 4xW,
2xDW, 1xQW
zEC12 (grey) integer vs. z13 (grey + purple)

Increased from 2 x 64 bit pipelines
To 8 x 64 bit pipelines but even more
Parallelism with smaller integer datatypes
Integer Dataflows from zEC12 to z13

Integer Dataflow Roads are 4 times as wide supporting up to 18 times the number of elements.

OLD 64 bit one lane road

OLD 64 bit one lane road

New 64 bit one lane road

New 64 bit one lane road

New 128 bit SIMD highway

New 128 bit SIMD highway
Integer – Error Coding Acceleration

- Includes 64 bit Cyclic Redundancy Coding (CRC) assists to perform
  - AxB + CxD + E of 64 bit in carryless form.

- Also includes a Checksum accelerator to add 5 x 32 bit operands modulo 2**32 to accumulate a new 128 bit operand to a running sum every cycle.
More Execution Pipelines Need More Data

- Quadrupled the FPRs to create the Vector Register file
- VRs contains floats, integers, and strings

Think of this as a parking lot for rental cars
Or a truck depot.
Speeding Up Hash Functions / Integer Divide

- **Z196**
  - Multiplicative integer divide algorithm
  - Inside BFU, 1x, blocking

- **zEC12**
  - SRT based, 1st generation (k=2)
  - Speedup over Z196: 1.4 – 3.6x

- **z13**
  - SRT based, 2nd generation (k=3)
  - Stand-alone engines in FXU, 2x
  - Non-blocking
  - Speedup over zEC12: 1.7x
  - Throughput: 3.4x

Ex.: 234876 ÷ 2 = 78292
234876 ÷ 7893 = 29
Z13 Floating Point throughput is twice as wide

Z13 has 2 BFU pipelines

SIMD / Vector architecture makes it easy to clump data
To fully utilize 2 BFUs with 1 thread.
Note we double pump a vector to the same BFU pipeline
Improved Binary Floating-Point (DP, SP) Latencies

<table>
<thead>
<tr>
<th></th>
<th>z196 / zEC12</th>
<th>z13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add, sub, mul, FMA, convert</td>
<td>8</td>
<td>8</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>Divide 64b</td>
<td>40</td>
<td>27</td>
</tr>
<tr>
<td>SQRT 32b</td>
<td>39</td>
<td>21</td>
</tr>
<tr>
<td>SQRT 64b</td>
<td>53</td>
<td>36</td>
</tr>
</tbody>
</table>
Z13 allows multiple floating point units to execute at same time and does not stall on long operations like divide and square root.
z13 does not stall on divide, each floating-point pipeline now consists of 5 sub-pipelines.

zEC12 has 1 pipeline
Run-on instructions
Block pipeline

z13 has non-blocking pipelines,
only blocks on specific resource /sub pipeline, such as divider
Floating-point Quad precision is much faster

- All other platforms use software, z is only platform to use hardware
- Moved quad precision to wide pipeline
- Now over 3x faster with 2x the bandwidth for a total of 6x performance over zEC12
- Also non-blocking

Multiple passes
BFP Quad Precision latency in cycles

<table>
<thead>
<tr>
<th>Operation</th>
<th>zEC12</th>
<th>z13</th>
</tr>
</thead>
<tbody>
<tr>
<td>add, sub</td>
<td>35</td>
<td>11</td>
</tr>
<tr>
<td>multiply</td>
<td>55 - 97</td>
<td>23</td>
</tr>
<tr>
<td>divide</td>
<td>~165</td>
<td>49</td>
</tr>
<tr>
<td>sqrt</td>
<td>~170</td>
<td>66</td>
</tr>
</tbody>
</table>

1 engine

2 engines

Each 3 X faster
String Acceleration

- Character types supported are Byte, Half Word, and Word

- Loads and Stores for both
  - known length strings (Java strings), and
  - those with a null terminating character (C strings)

- String Search Acceleration
  - Find Equal
  - Find Not Equal
  - Range Compare
In the past, loading or storing to a string could result in exceptions

NEW INSTRUCTIONS:
Load with Length
Java strings

Load to Block Boundary
C strings

Are exact and will not set off an alarm.
String Acceleration with Powerful New Instructions

16 characters of the string can be examined every cycle.
Search text can be 16 characters or 8 pairs of characters for ranges.

Vector Find Element Equal – “Barak”
Returns a Pointer

Vector String Range Compare
Use it to Find IsNotAlphaNumericORHyphen
(GE ‘0’ LE ‘9’) or
(GE ‘a’ LE ‘z’) or
(GE ‘A’ LE ‘Z’) or
(EQ ‘-’ EQ ‘-’);

Returns a pointer

<table>
<thead>
<tr>
<th>Name</th>
<th>Telephone Number 1</th>
<th>Telephone Number 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>123-45-6789</td>
<td>845-555-1212</td>
</tr>
<tr>
<td>Barak Hussein Obama II</td>
<td>987-65-4321</td>
<td>212-555-1212</td>
</tr>
<tr>
<td>Mary H Lamb</td>
<td>555-55-5555</td>
<td>555-555-5555</td>
</tr>
<tr>
<td>Humpty Dumpty</td>
<td>666-66-6666</td>
<td>666-666-6666</td>
</tr>
<tr>
<td>Kings Men</td>
<td>777-77-7777</td>
<td>777-777-7777</td>
</tr>
<tr>
<td>Emergency</td>
<td>911-91-1911</td>
<td>???-???-??????</td>
</tr>
</tbody>
</table>
MATRIX MULTIPLICATION
Matrix multiply - DGEMM

- A product matrix element $p(i,j)$ is equal to the sum of a row $i$ x column $j$

To create independent operations we do the opposite,
We multiply a column by a row and form independent product elements

FPRs are allocated to accumulated product elements (in green)
An iteration consists of accumulating one new product to a set of independent Product elements.
DGEMM 4x4

- Load 8 FPRs
- 16 FMAs
  - With 2 BFU pipes
  - Could separate by 8 cycles
  - Need 24 arch FPRs (use Vector Scalar)

- # Regs = NxM + N + M, could do 4x5 with 32 regs but not power of 2
- Other algorithms could separate the adds to not delay multiplies

Form 16 entries At a time

4 rows

x

4 columns

=
DGEMM 4x4 Vector-Scalar

No stall cycles

16 independent FMAs
DGEMM Conclusions

- So 2x4 algorithm about optimal for 16 regs and only 1 BFU needed.
- To use bandwidth of 2 BFUs with 8 cycle latency then 4x4 needed. This would approximately double the throughput.
  - Can form 16 entries in same time to form 8.
- Use of vectorized 4x8 causes no slow down to code now, and in the future may allow for up to 2x performance.
Range compare for isAlphaNumeric() in one instruction

<table>
<thead>
<tr>
<th></th>
<th>0 'a'</th>
<th>GE</th>
<th>1 'z'</th>
<th>LE</th>
<th>2 'A'</th>
<th>GE</th>
<th>3 'Z'</th>
<th>LE</th>
<th>4 '0'</th>
<th>GE</th>
<th>5 '9'</th>
<th>LE</th>
<th>Cntl 15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1 1 1</td>
<td>1 1</td>
<td>1 1 1</td>
<td>1 1</td>
<td>1 1 1</td>
<td>1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0 1 A b c 0 5 6 7 d z Z

F T T F F F T T F ...  
T T T T T T T T T ...  
T T T F F F F T T T ...  
T F F T T T F F F ...  
T T T T T T T T T ...  
F F F T T T F F F ...  

AND

AND

AND

AND

AND

1 1 1 1 1 1 1 1 1 1

© 2015 IBM Corporation
DGEMM 2x4

This is the actual sequence used:

- Load 0,2,9,E,C,B
- \(A = A + 2xB\)
- \(8 = 8 + Bx0\)
- \(7 = 7 + 2x9\)
- \(5 = 5 + 2xE\)
- \(3 = 3 + 2xC\)
- \(1 = 1 + 0x9\)
- \(6 = 6 + Ex0\)
- \(4 = 4 + Cx0\)

- 6 Loads followed by 8 FMAs

Each iteration move

Form 8 entries At a time
2x4 scroll pipes

- First iteration
  - 6 Loads come back in different cycles
  - So FMAs start up randomly

- Further iterations
  - Loads done ahead of time
  - Critical path is FMA to FMA for accumulation
SCROLL PIPELINES
D-decode, m-mapping, I-issue, F-fpu execution, p-putaway, r-checkpointed
DGEMM 2x4

8 FMAs

FPU pipeline latency critical
And how many multiplies can you
Stick between dependent accumulation

If you could stick 16 FMAs
Between dependent ones
You’d get 2 FMAs per cycle