19840: IBM z Systems Microprocessor Optimization Primer

C. Kevin Shum, IBM Distinguished Engineer
Member, IBM Academy of Technology
Email: cshum@us.ibm.com
linkedin.com/in/ckevinshum twitter.com/C_Kevin_Shum
Learning Objectives

• For anyone who is new to the z System ecosystem and is working on optimization of software code performance
• To understand a little bit about the processor subsystems of IBM z Systems, from z196 to z13
• To learn some interesting parts of the z/Architecture
• To gain high level insights with potential methods to optimize code performance
• For the very detailed documentation where this presentation is based on, see:
  http://ibm.co/1qeGrpc
z/Architecture: Basics

- 64 bit architecture, supported by latest z Systems
  - CISC (vs. RISC) with highly capable instructions
  - Big-Endian (vs. Little-Endian)
- Instructions include
  - Typical load/store/Reg-Reg/Reg-Storage instructions, with logical/arithmetic operations
  - Indirect/relative branches, conditional branches, subroutine calls
  - Storage to Storage operations
  - Hexadecimal, IEEE Binary/Decimal, floating point operations
z/Architecture: Advanced

- Atomic Operations
  - Compare and Swap, Load and Add
  - Simple “immediate”-storage operations, e.g. AND/OR
- Hardware Transactional Memory (since zEC12)
  - Atomic operations across multiple instructions
- Run-time Instrumentation (since zEC12)
  - Hardware assisted profiling of software behavior
- Simultaneous Multithreading (since z13)
  - 2 software threads running in parallel inside a processor core
- SIMD: Single Instruction Multi-Data (since z13)

```
LHI    R2, 1
LOOP   LT  R1, lock
       JNZ LOOP
       CS  R1, R2, lock
       JNE LOOP

*Pseudo-code for contested locks
```
z/Architecture: Uni Storage Consistency

• General rules (important for full software compatibility):
  – Program must behave as if executed serially
  – Each instruction can use all results of previous instructions

• Operand accesses must be observed to be done in program order
  – Store / fetch conflicts recognized by real* address
  – Most operands processed left to right (except *Fixed-point decimal* operands)
  – Storage-storage (SS) instructions are in a byte-by-byte observability

• Instruction pre-fetches may be observed
  – Instructions executed must reflect prior stores
  – Serialization can add further restrictions (more in 2 pages)

*Real address - Result of dynamic address translation (DAT) or the logical address when DAT is off, subject to prefixing
z/Apartment: MP Storage Consistency

- Must be able to define consistent ordering of accesses
  - “as seen by this and other processors"
  - Some instruction operations are allowed to have ambiguous
- Operand fetches and stores must appear to occur in proper order
  - All processors must obey uniprocessor rules
  - For OOO pipeline, program observation will still appear in proper order
- Operand accesses must be DW-consistent
  - No "score-boarding" should be observed
  - e.g. across a “LM” (load multiple of 32bit GRs)
- Instruction fetches are generally allowed in any sequence
z/Architecture: Serialization

- z/Architecture defines a set of situations in which additional restrictions are placed on the storage access sequence.
- Defined as “A serialization operation consists in completing all conceptually previous storage accesses and related reference-bit and change-bit settings by the CPU, as observed by other CPUs and by the channel subsystem, before the conceptually subsequent storage accesses and related reference-bit and change-bit settings occur.”
- Defined for specific points in instruction stream:
  - Usually "before and after" specific opcodes.
  - Includes Instruction fetches as well as operand accesses.
  - Exception: Instruction fetch for the serializing instruction itself.
z/Architecture: Implementation

• Most instructions are implemented directly in hardware
• Some “multi-operation” instructions are expanded into multiple micro-operations
• The more complicated instructions or operations are implemented through “millicode”
  – A special firmware layer
  – A vertical microcode, optimized for the corresponding processor
  – Executes like a “hidden” subroutine
  – Uses most z/Architecture instructions, and “specials” only for millicode
  – Operates an attached co-processor where appropriate
Processor: History Highlights

- **Z196**
  - 5.2 GHz, 1st generation out-of-order pipeline
  - High Word Architecture, Conditional load/store
  - More non-destructive instructions

- **zEC12**
  - 5.5 GHz, 1st generation level 2 branch predictor, split level 2 caches
  - Hardware Transactional Memory
  - Run-Time Instrumentation

- **Z13**
  - 5.0 GHz, 2x wide pipeline
  - Simultaneous Multithreading Support (2 threads)
  - Single-Instruction-Multiple-Data (strings, fixed/floating point)
Processor: Cache Handling

- L1 and L2 are private Caches, Store-Through
  - L3 and L4 are shared, and store-in
- Cache line sizes are typically 256 bytes
  - Extract CPU Attribute Instruction for cache related information
- Self-Modifying Code supported, but can take a lot of cycles
  - Avoid false sharing of code and write-able data in same cache line
- Core pipeline processing of “stores” uses a store-allocate policy
  - Wait until the exclusive ownership is obtained in L1
- Cache lines in lower level are owned (or tracked) by upper level
  - Cache States: “exclusive”, “read-only”, “unowned”
Global Fabric Interface

384MB
Shared eDRAM L4

- 48MB Shared eDRAM L3
- 6 L3s
- 36 L1 / L2s

1st Level:
- 64 KB I-L1 + 96 KB D-L1
- 6-way Set Associative I-L1
- 256B cache line size

2nd Level:
- 1 MB L1+* Inclusive of D-L1
- 1 MB L2 Inclusive of L1, L1+* 8-way* Set Associative
- 256B cache line size

3rd Level:
- 48 MB Inclusive of L1, L1+, L2
- 12-way Set Associative
- 256B cache line size

4th Level:
- 384 MB
- Inclusive of L1, L1+, L2, L3
- 24-way Set Associative
- 256B cache line size

* The L1+/L2 design in zEC12 is too complicated for this document. One can treat it as two L2s, each 1MB and 8-way set associative.

Global Fabric Interface

480MB
Shared eDRAM L4

- 64MB Shared eDRAM L3
- 3 L3s
- 24 L1 / L2s

1st Level:
- 96 KB I-L1 + 128 KB D-L1
- 8-way Set Associative I-L1
- 256B cache line size

2nd Level:
- 2 MB I-L2 Inclusive of I-L1
- 2 MB D-L2 Inclusive of D-L1
- 8-way Set Associative
- 256B cache line size

3rd Level:
- 64 MB Inclusive of L1, L2
- 16-way Set Associative
- 256B cache line size

4th Level:
- 480 MB cache +
- 224 MB Non-data Inclusive Coherency (NIC) Directory
- Inclusive of L1, L2, L3
- 30-way Set Associative NIC
- 256B cache line size
Processor: Instruction Processing

• Branch Prediction asynchronously predicts sequences of branches
  – Directions and targets
• Instructions are fetched from instruction cache in respect to predictions
• Instructions are decoded and then dispatched in-order into issue queue
  – Instructions are managed in groups of up to 3 instructions/micro-ops
• Ready instructions are then issued, potentially out-of-order (OOO), to the execution units
• Instructions finish and are then completed in-order to have their results architecturally committed
  – Outstanding OOO resources are now released for future uses
Instruction Flow

1. (z13) new execution units to accelerate business analytics workloads
2. (z13) additional execution units for higher core throughput
3. (z13) additional instruction flow for higher core throughput
4. (zEC12) new instruction flow and execution units for relative branches
5. (z13) new execution units to accelerate business analytics workloads
Processor load/store handling

- Load/Store unit handles data operand accesses with controls to L1 and L2 data caches
- L1 cache can return 2 up to 8-byte data w/ any alignment
- Vector load/store requires 2 cycles to load the full 16 byte of data
- Includes a simple loop-based stride HW prefetcher to prefetch up to 2 cache lines
- Supports software Prefetch Data instructions (with queues & limitations)
- Can forward pending storage updates into the pipeline for future reads
  - If not bypassable, a pipeline reject happens to the “load”
  - If a “load” executes before the “store”, a pipeline flush occurs
  - Multiple predictive mechanisms built-in to enable data forwarding
Instruction Highlights: Conditionals

- Some conditional branches might be very hard to predict
- Conditional load/store instructions allow execution being predicated on the condition code
- Slower than traditional load/store; but can be used to avoid costly branch mis-prediction flushes

Old Code

<table>
<thead>
<tr>
<th>CR</th>
<th>R1, R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRNE</td>
<td>skip</td>
</tr>
<tr>
<td>L</td>
<td>R4, (addressX)</td>
</tr>
<tr>
<td>skip</td>
<td>AR R4, R3</td>
</tr>
<tr>
<td>..</td>
<td></td>
</tr>
</tbody>
</table>

New Code

<table>
<thead>
<tr>
<th>CR</th>
<th>R1, R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>R4, (addressX), b'0111'</td>
</tr>
<tr>
<td>AR</td>
<td>R4, R3</td>
</tr>
<tr>
<td>..</td>
<td></td>
</tr>
</tbody>
</table>

*Pseudo-code for illustration only*
Instruction Highlights: Branch Preloads

- Provided since zEC12
- Potentially overcomes the limitation of hardware branch history tables
  - The processor may now predict the presence of branches without having seen them before, or if their history was displaced
  - The directives are not intended for overriding or modifying an existing hardware history entry's target address to a different value
- The more (taken) branches in-between, and the further away in sequential memory address, the more likely a preload will succeed
- The relative form of preload instruction, BPRP (Branch Prediction Relative Preload), is better than BPP (Branch Prediction Preload)
Instruction Highlights: Prefetch Data

- Provided since z10
- Potentially overcomes the limitation of hardware’s cache limitation by prefetching a cache line of data operand
  - Directed cache line will be loaded into private data L1/L2 (etc.)
  - Can also indicate the “state” of cache line to acquire as (upon a miss)
    - Read-only (will never store); vs. Exclusive (will store)
    - Also provides way to “release” a specific cache line
  - Proactive demote/unown of exclusive lines can allow another processor to acquire exclusivity faster
- Both prefetching and “un-fetching” can be difficult to use
Instruction Highlights: Access Intent

- Provided since zEC12
- Potentially provides some hints to the cache system
  - Overall intention of the next immediate instruction’s operand accesses
  - Adjusts/Steers related cache handling
- Can indicate the “state” of cache line to acquire as (upon a “load” miss)
  - Read-only (will never store); or Exclusive (will store later)
- Can indicate reusability of cache line
  - Overrides typical hardware “least recently used” (LRU) eviction scheme
  - Leaves cache lines in compartments that will be evicted sooner
  - Potentially good for “streaming” accesses
Recommendations: General

- General guidance only
  - Exceptions cases exist, but normally rare with compiled code
- Processors are designed to handle wide range of applications
  - Cache intensive (big and low latency caches)
  - Compute intensive (out of order, lots of execution bandwidth)
- Pipeline changes are expected from one design to next
  - Aim to improve performance of “old-existing” code
  - Newly compiled code can take advantage of new instructions and get even more
  - Slight performance variations expected if code is “super-tuned” to one design
- Other references:
  - SHARE presentations from Dan Greiner, Peter Relson, etc.
Recommendations: Branch Related

- Align frequently called functions to storage boundary (16b, 32b, etc.)
- Frequent code path should be the not-taken path (fall-through)
- Unroll loops to at least 12* instructions to enable efficient predictions
- Pairing up calls and returns
- Inlining subroutines if they are small and called often
- Use relative branches instead of indirect branches
- Use “combo-branches”
  - Branch on count, branch on index; Compare and branch/jump
- Use conditionals if certain branches are hard-to-predict

* z13 prediction latency to cover 2 cycles of dispatched instructions
Recommendations: Instruction Selections

• Use simple register-storage instructions, instead of load+register-register
  – A(dd), instead of L(oad), A(dd)R(egister)
• Use instructions that do not set condition code
• Use simple yet powerful instructions, like rotate-then-*-selected bits
  – Register is rotated, operations with another register on selected bits
• Use compare-and-trap, for null-pointer checks
• Did you know?
  – 3-way address generation is handled without additional delays for operand accesses (vs. 2-way)
  – Register clearing instructions were special cased to be “really” fast
Recommendations: Code Arrangement

• Instructions are managed in group of 3
  – Branches end a group in z196, then end if predicted taken or second in a group after zEC12
  – Arrange code to maximize effective group size with simple instructions

• Dependent results are bypassable within the same “side” of FXU
  – Consecutive dependent fixed-point code => same group => same side

• Avoid mixed-mode floating point operations
  – Long<>Short, Binary<>Hex

• Try using software hint instructions
  – Needs to be very early to be effective, use run-time profile if possible
Recommendations: Instruction Cache

• Avoid sharing cache lines between code and write-able operand data
  – The further away they are from each other, the less likely they collide
  – Target operand of an “Execute” is handled as code (not data)
  – Watch out in older assembler code where local (static) save areas and macro expansions with in-line storage parameters can be mixed into code area

• Instruction Cache Optimization
  – Minimize cache lines used, by packing frequently used modules
  – Don’t overdo software hints, apply only where it makes sense
  – Balance resulting loop size after unrolling and inlining
  – Branch Preload also touches target cache line
Recommendations: Data Cache

- **Data Cache Optimization**
  - Don’t mix multiple distinct shared writeable data in the same cache line
  - Avoid using storage as a running variable (that can be updated very frequently)
  - L1 access pipeline has been 4 cycles
    - From a “load” issue to the dependent instruction issue is 4 cycles
    - Schedule non-dependent instructions in-between
  - Avoid complicated bypassing need from pending stores to subsequent “loads”
  - Try using NIAI - Next Instruction Access Intent and PFD/PFDRL - Prefetch Data (Relative) where appropriate; but watch out for overkill
Conclusion and Remarks

- Learned a little bit about z/Architecture
  - Including some not-so-obvious instructions
- Learned a little bit about the processor subsystem design
  - Including some details on cache management and instruction pipeline
  - Learned a little bit about how to optimize for the latest design
  - Including some “not-to-do” guidelines
- Since this is a “primer to a primer”, not all materials are discussed
  - Refer to the web document, which is easily a 5-hours presentation
  - Needs your feedback on what to add/change/update for the future
    http://ibm.co/1qeGrpc
Visit the SHARE Booth (#303)

**Educate**
- Learn more about **hot topics** from peers in the **Tech Talk Corner**
- Take part in the **Hack-a-Thon** with IBM & Rocket

**Network**
- **Engage** with fellow attendees
- Connect with the **zNextGen®** community
- Join the **#SHAREatl** social media conversation

**Influence**
- Discover ways to **get involved** with SHARE
- Meet **SHARE Volunteers**

*Complete your session evaluations online at SHARE.org/Evaluation*

Except where otherwise noted, this work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs 3.0 license.
http://creativecommons.org/licenses/by-nc-nd/3.0/
Thank You for Attending!
Please remember to complete your evaluation of this session in the SHARE mobile app.

19840: IBM z Systems Processor Optimization Primer