IBM z Systems

IBM Z / LinuxONE System Processor Optimization Primer

v2 (Jan 2018) – updates in blue

C. Kevin Shum

Distinguished Engineer
IBM Z Systems Microprocessor Development
Member of IBM Academy of Technology

Trademarks

The following are trademarks of the International Business Machines Corporation in the United States, other countries, or both.

Not all common law marks used by IBM are listed on this page. Failure of a mark to appear does not mean that IBM does not use the mark nor does it mean that the product is not actively marketed or is not significant within its relevant market.

Those trademarks followed by ® are registered trademarks of IBM in the United States; all others are trademarks or common law marks of IBM in the United States.

For a more complete list of IBM Trademarks, see www.ibm.com/legal/copytrade.shtml:

*BladeCenter®, CICS®, DataPower®, DB2®, e business(logo)®, ESCON, eServer, FICON®, IBM®, IBM (logo)®, IMS, MVS, OS/390®, POWER6®, POWER6+, POWER7®, Power Architecture®, PowerVM®, PureFlex, PureSystems, S/390®, ServerProven®, Sysplex Timer®, System p®, System p5, System x®, z Systems®, System z9®, System z10®, WebSphere®, X-Architecture®, z13™, z13s ™, **z14** ™, z Systems™, z9®, z10, z/Architecture®, z/OS®, z/VM®, z/VSE®, zEnterprise®, zSeries®, IBM Z ®

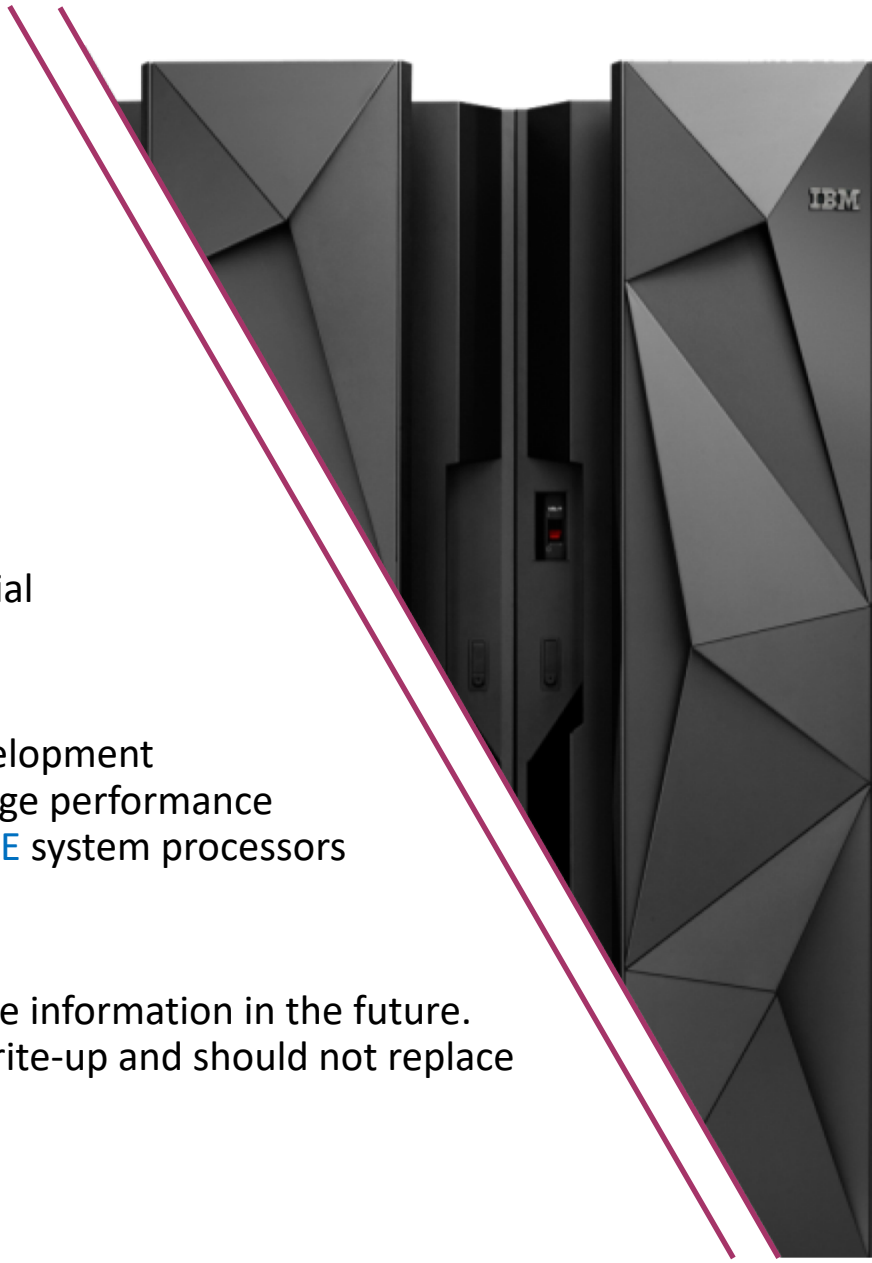
The following are trademarks or registered trademarks of other companies.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.
Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.
Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.
Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.
Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.
UNIX is a registered trademark of The Open Group in the United States and other countries.
Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
ITIL is a registered trademark, and a registered community trademark of the Office of Government Commerce, and is registered in the U.S. Patent and Trademark Office.
IT Infrastructure Library is a registered trademark of the Central Computer and Telecommunications Agency, which is now part of the Office of Government Commerce.

*** All other products may be trademarks or registered trademarks of their respective companies.**

Notes:
Performance is in Internal Throughput Rate (ITR) ratio based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput improvements equivalent to the performance ratios stated here.
IBM hardware products are manufactured Sync new parts, or new and serviceable used parts. Regardless, our warranty terms apply.
All customer examples cited or described in this presentation are presented as illustrations of the manner in which some customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics will vary depending on individual customer configurations and conditions.
This publication was produced in the United States. IBM may not offer the products, services or features discussed in this document in other countries, and the information may be subject to change without notice. Consult your local IBM business contact for information on the product or services available in your area.
All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.
Information about non-IBM products is obtained Sync the manufacturers of those products or their published announcements. IBM has not tested those products and cannot confirm the performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.
Prices subject to change without notice. Contact your IBM representative or Business Partner for the most current pricing in your geography.

Documentation Objectives



Provides an overview of the processor subsystems of IBM Z / LinuxONE systems, with focus on the core microarchitectures from z196 to z14

Version 2 (v2) includes readability changes, minor corrections, additional information, and z14 design updates

Gives high-level insights with information and potential methods to optimize for code performance

Fosters a deep technical exchange with non-IBM development teams within the open source community to encourage performance optimization that is tailored towards IBM Z / LinuxONE system processors

"tell us what you need to know"

If needed, this document might be updated with more information in the future. However, it is not intended to be a comprehensive write-up and should not replace any formal architecture documents

z/Architecture and Implementation

- z/Architecture[1] is a 64-bit architecture that is supported by IBM Z / LinuxONE microprocessors
  - A Complex Instruction Set Computer (CISC) architecture, including highly capable (and thus complex) instructions
  - Big-Endian (BE) architecture (vs. Little-Endian) where bytes of a multi-byte operand data element are stored with the most significant byte (MSB) at the lower storage address
- z/Architecture grows compatibly upon each generation, and includes many innovative features
  - Typical load/store/register-register/register-storage instructions, including logical and arithmetic functions
  - Branch instructions supporting absolute and relative offsets, and subroutine linkages
  - Storage-storage instructions, e.g. “MOVE characters (MVC)” (for copying characters), including decimal arithmetic
  - Hexadecimal, binary, and decimal (both IEEE 754-2008 standard) floating-point operations
  - Vector (SIMD) operations (from z13 on), including fixed-point, floating-point, and character string operations; decimal operations added from z14* on
  - Atomic operations including COMPARE AND SWAP, LOAD AND ADD, and OR (immediate) instructions
  - Hardware transactional memory, through the Transactional Execution Facility (since zEC12), including the definition of a constrained transaction that can be retried by the hardware
  - Two-way Simultaneously Multi-Threading (SMT-2) support (since z13)
- Highly complex instructions are implemented through a special firmware layer – millicode[2]
  - Millicode is a form of vertical microcode that is pre-optimized for each processor generation
  - An instruction that is implemented in millicode is executed by the hardware similar to a built-in subroutine call that transparently returns back to the program when the millicode routine ends
  - A millicode instruction routine consists a subset of the existing instructions in the z/Architecture, with access to its own pool of internal registers in addition to program registers and specialized hardware instructions
  - Some complex routines might involve operating along with a private co-processor or special hardware that is only accessible by millicode

*Further z14 updates in page 10

Microprocessor CPU State

- Under z/Architecture, the architected states of a software thread running on a microprocessor core, referred to as **CPU**, involves the following highlighted components*
  - Program Status Word (PSW)
    - Instruction Address (aka Program Counter, including where the next instruction address is for execution)
    - Condition Code (2 bits, set depending on results of some previously executed instructions)
    - Addressing Mode (2 bits, indicating 24-bit, 31-bit or 64-bit addresses)
    - DAT Mode (when 1, indicates that implicit dynamic addressing translation is needed to access storage)
    - Address Space Control (controls translation modes: Access Register specified, Primary, Secondary, or Home)
    - Key (4-bit access key that is used to access storage when key-controlled protection applies)
    - Wait state (waiting, no instructions are processed)
    - Problem State (when 1, indicates problem state, not supervisor state; privileged instructions cannot be executed and semi-privileged instructions can be executed only if certain authority tests are met)
    - Masks (control various kinds of interrupt enablement)
  - Registers
    - Access Registers (ARs): 16 total, 32 bits each, used mainly during access register based translation mode
    - General Registers (GRs): aka general-purpose registers, 16 total, 64 bits each, with high and low 32-bit-word independent operations for address arithmetic, general arithmetic, and logical operations
    - Floating-Point Registers (FPRs): 16 total, used by all floating-point instructions regardless of formats; a register can contain either a short (32-bit) or a long (64-bit) floating-point operand; while a pair can be used for extended (128-bit) operands
    - Vector Registers (VRs): available since z13, 32 total, 128 bits each, when present, FPRs overlay the VRs
    - Floating-Point-Control Register: 32-bit, contains mask, flag and rounding mode bits, and a data exception code
    - Control Registers (CRs): 16 total, bit positions in the registers are assigned to particular facilities in the system

*For more information, see z/Architecture Principles of Operations

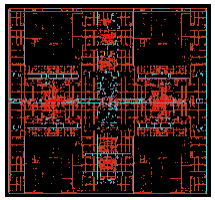
Highlights of the Recent Microprocessor Cores

- The z10 processor[3,4] started the recent ultra-high frequency pipeline design in Z processors
- Z196[5,6] introduces the first generation out of order pipeline design
  - Runs at 5.2 GHz on the EC class machines
  - Introduces high-word architecture with operations on upper 32 bits of general registers (GRs)
  - Adds more nondestructive arithmetic instructions
  - Adds conditional load and store instructions, for reducing potential branch wrong penalties
- zEC12[7] improves upon the first generation out of order design
  - Runs at 5.5 GHz on the EC class machines
  - Introduces level-2 branch prediction structure[8]
  - Introduces a set of split level-2 (L2) caches, providing low-latency large capacity instruction and operand data caching per processor core
  - Integrates tightly L2 data cache lookup into level-1 (L1) data cache design, further improves L2 data cache access latency
  - Supports Hardware Transactional Memory[9] (Execution) and Run-Time Instrumentation facilities
- z13[10] improves further on top of the zEC12 design
  - Runs at a slightly lower maximum frequency of 5 GHz; with a much wider pipeline (2x) to handle more instructions per cycle for a net increase in overall instruction execution rate
  - Integrates L2 instruction cache lookup into L1 instruction cache design to improve L2 instruction cache access latency
  - Supports simultaneous multi-threading (SMT) for 2 threads
  - Introduces Single-Instruction-Multiple-Data (SIMD) instructions for vector operations[1]
- Z14[14] improves further on top of the z13 design with updates highlighted in page 10

Figure 1: z Processor Historic Roadmap

65 nm

z10
2/2008



Workload Consolidation and Integration Engine for CPU Intensive Workloads

Decimal FP

InfiniBand

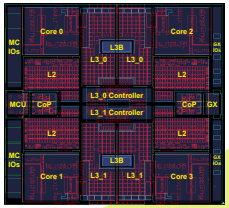
64-CP Image

Large Pages

Shared Memory

45 nm

z196
9/2010



Top Tier Single Thread Performance, System Capacity

Accelerator Integration

Out of Order Execution

Water Cooling

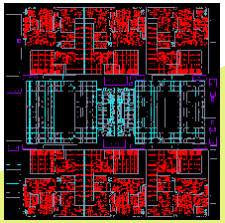
PCIe I/O Fabric

RAIM

Enhanced Energy Management

32 nm

zEC12
9/2012



Leadership Single Thread, Enhanced Throughput

Improved out-of-order

Transactional Memory

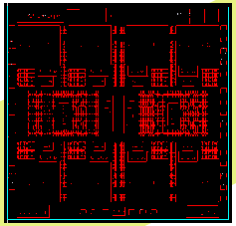
Dynamic Optimization

2 GB page support

Step Function in System Capacity

22 nm

z13
3/2015



Leadership System Capacity and Performance

Modularity & Scalability

Dynamic SMT

Supports two instruction threads

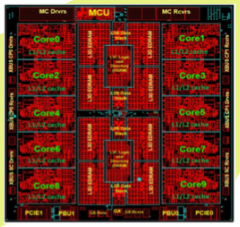
SIMD

PCIe attached accelerators

Business Analytics Optimized

14 nm

z14
9/2017



Pervasive encryption

Low latency I/O for acceleration of transaction processing for DB2 on z/OS

Pause-less garbage collection for enterprise scale JAVA applications

New SIMD instructions

Optimized pipeline and enhanced SMT

Virtual Flash Memory

System Cache Structure

- A IBM Z system consists of multiple computing nodes that are connected through the global fabric interface. Each system node includes a number of processor (CP) chips (6 in z196, 6 in zEC12 and 3 in z13*)
  - In z10, z196, and zEC12, the system consists of up to four nodes, with each node fully interconnected to every other node through the level-4 (L4) caches
  - In z13*, the system consists of up to eight nodes, packaged as one pair of nodes per drawer
    - The nodes on each drawer are connected to each other through the L4 caches
    - Each node is connected to the corresponding node on each other drawer through the L4 caches
    - The three CP chips in each node are connected to each other through the shared on-chip level-3 (L3) caches
- Each processor (CP) chip includes a number of processor cores
  - There are 4 cores in a z196 CP chip, 6 in zEC12, and 8 in z13
  - Each core includes both local level-1 (L1) instruction and operand data caches, and a local level-2 (L2) cache
  - Since zEC12, a pair of L2 caches supports instruction and operand data separately
  - Each L2 cache is connected to the on-chip (shared) L3 cache
- Caches are managed “inclusively” such that contents in lower-level caches are contained (or tracked) in the higher-level caches
  - In z13*, the L4 maintains a non-data inclusive coherency (NIC) directory to keep track of cache-line states in the L3 without having to save a copy of the actual cache-line data.
  - Cache lines are managed in different states (simplistic view):
    - “exclusive” (at most 1 core can own the line to store or update at any time)
    - “shared” or "read-only" (can be read by 1 or more cores at any time)
    - “unowned” (where no core currently owns the cache line)
  - When a cache line is shared and a processor wants to store (update) one of the elements, a cache coherency delay is required to invalidate all existing read-only lines in other caches so this processor can be the exclusive owner
  - Similarly, this exclusive line will need to be invalidated before another processor can read or write to it

*z14 related updates in page 10

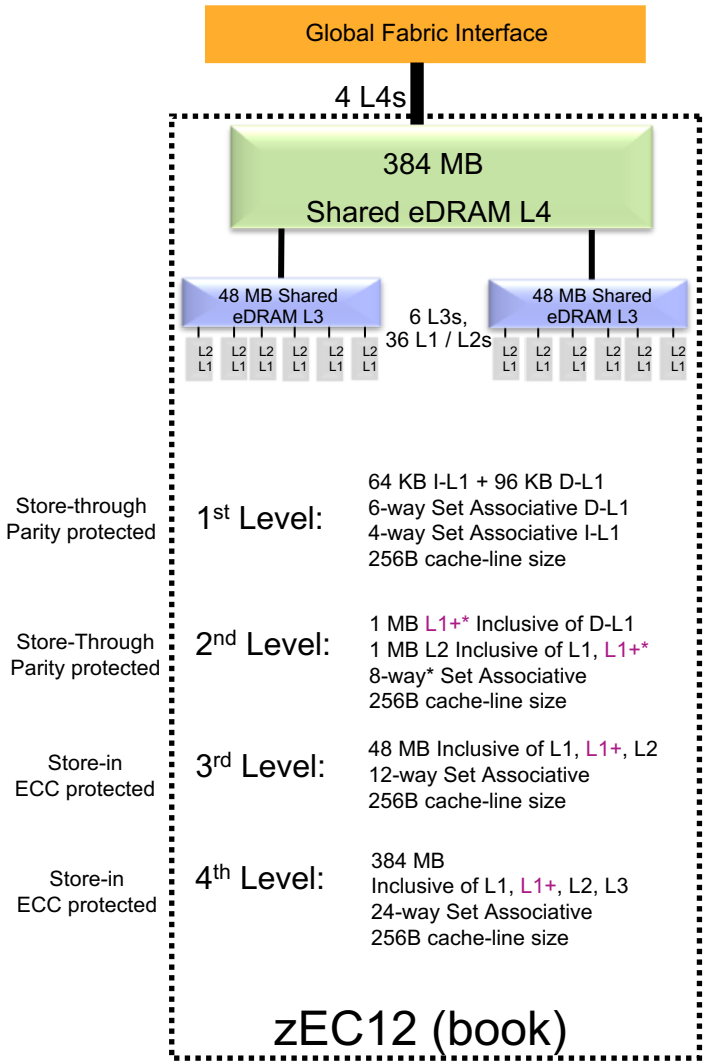
Near-Core Cache Operations

- The L1 and L2 (private) caches are store-through, i.e., each storage update is forwarded immediately to the shared L3 cache after the instruction performing the update completes
  - For reference, L3 and L4 (shared) caches are store-in, i.e., storage updates are kept in the cache until the cache entry is replaced by a new cache line or evicted to move to another L3 or L4 cache
- The cache-line size (for all caches) being managed across the cache subsystem is currently 256 bytes
  - Although the cache-line size remains stable across recent machines, it should not be relied upon
  - However, it is unlikely that the cache-line size will grow beyond 256 bytes
  - EXTRACT CPU ATTRIBUTE instruction should be used to obtain information about the cache subsystem, e.g. cache sizes and cache-line sizes for each cache level
- The z/Architecture and the processor design supports self-modifying code
  - However, supporting self-modifying code can be costly due to movement of cache lines between the instruction and operand data caches (L1 and L2). More details are provided at page 56 titled “Optimization – Data Placement”
  - Due to out of order and deep pipelining, self-modifying code becomes even more expensive to use and is not advised
  - Even if there is no intention to update the program code, false sharing of program code and writeable operand data in the same cache line will suffer similar penalties
- The L1 implements a “store-allocate” design where it must obtain the exclusive ownership before it can store into a cache line
  - The storing instruction will stall in the pipeline until the correct cache state is obtained
  - It is important not to share writeable operand data elements in the same cache line for independent multiprocessor operations
- The associativity of a cache (as specified in subsequent pages) reflects how many compartments are available for a particular cache line to be stored in
  - For an 8-way associative cache, a cache line (based on its line address) can be saved in one of 8 compartments

High-level updates about z14

- z14[14] improves upon the z13 SMT design with focus on special functions
  - Runs at a faster maximum frequency of 5.2 GHz; with a similar pipeline to z13
  - Maintains the tight integration of each L2 cache lookup with the corresponding L1 cache
  - Integrates the level-1 Translation-Lookaside-Buffer (TLB1) function into the L1 directory for both instruction and data cache accesses
  - Operates TLB2 lookup in parallel with L2 directory lookup pipeline to drastically reduce TLB miss penalties
  - Further improves handling of simultaneous multi-threading (SMT) for 2 threads, focusing on maximizing execution overlap within the pipeline, and parallelizing TLB and cache misses
  - Adds amazing performance and functionality in the Co-Processors (COP) for compression and cryptography
- z14 introduces many notable z/Architecture features, including:
  - Guarded Storage Facility to enable pause-less garbage collection for Java
  - New compression modes to improve compression ratio and to provide order-preserving compression
  - New encryption modes including SHA3, AES-GCM
  - True (hardware) Random Number Generation support
  - New SIMD instructions, e.g. Binary-Coded Decimal (BCD) arithmetic, single & quad precision floating-point, long-multiply
- z14 system structure and cache topology
  - The processor subsystem consists of up to 4 nodes, with 1 node per drawer
    - Each node is connected to each other node through the L4 caches
    - Each node consists of 2 clusters, with 3 CP chips per cluster
  - Each processor (CP) chip includes 1 L3 cache, which is shared by 10 processor cores through the L2/L1 caches similar to z13 design

Figure 2: Cache Hierarchy and sizes (zEC12 and z13)



* 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

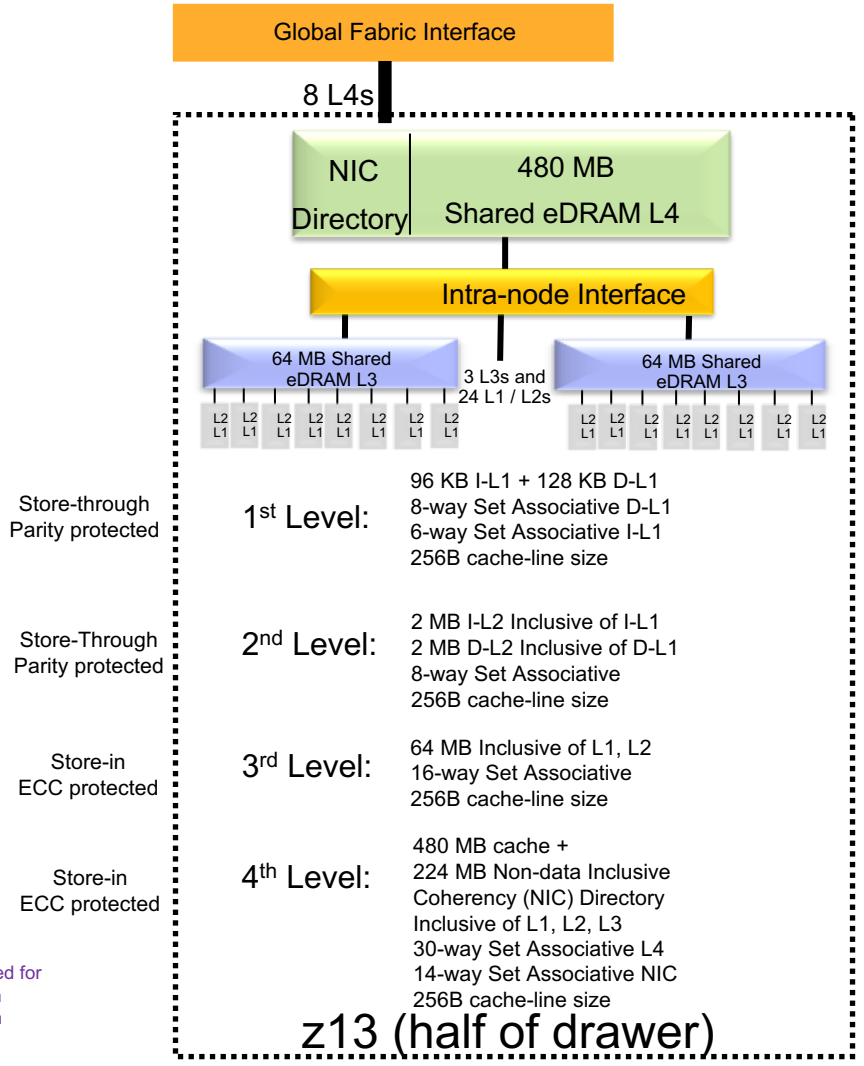
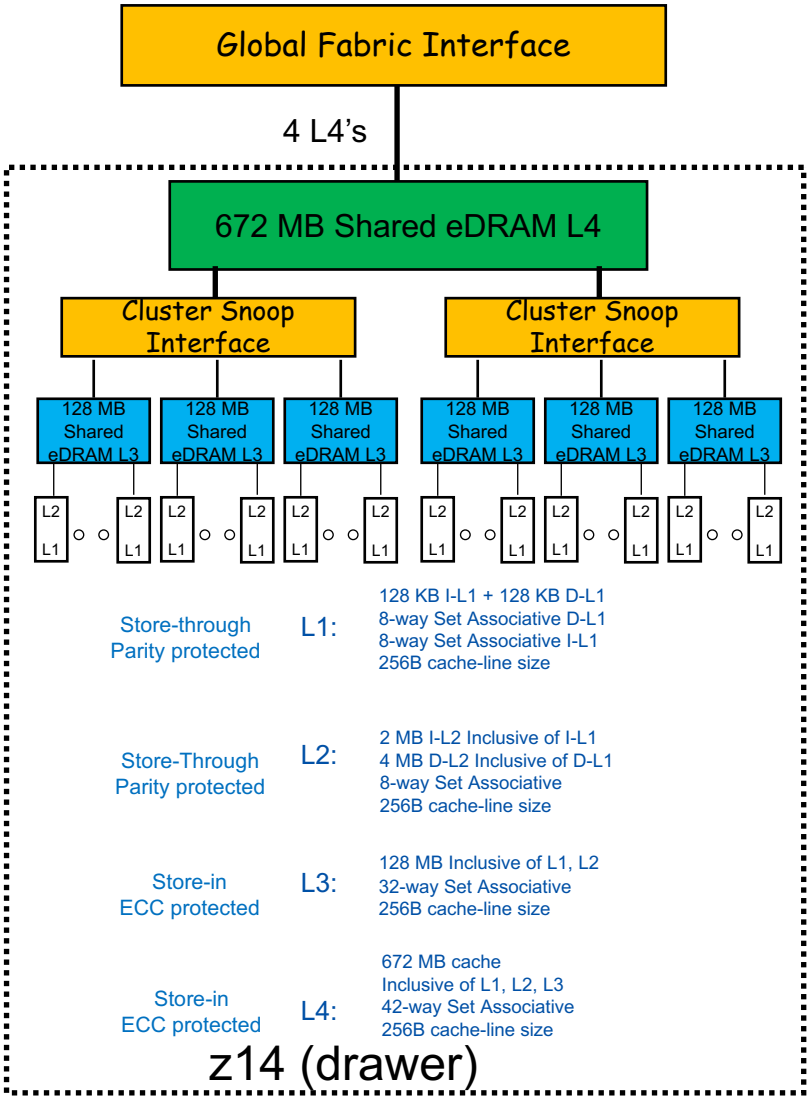


Figure 2a: Cache Hierarchy and sizes (z14)



High-Level understanding of the microprocessor core

- The z microprocessor cores can be simplified into a number of functional units (which are further described in some published papers):
  - Branch prediction unit
    - Two-level structure of branch histories; advanced design predicts both targets and directions
  - Instruction caching and fetching unit
    - Based on branch prediction information, delivers instructions in a seamless fashion
  - Instruction decoding and issuing unit
    - Decodes instructions in groups; issues micro-operations out-of-order to the execution units
  - Fixed-Point Execution unit
    - Executes most of the fixed-point operations, and (since z13) fixed-point divides
  - Vector & Floating-Point Unit
    - Handles floating-point arithmetic operations, complicated fixed-point operations, and (since z13) vector operations
  - Load/Store (or Operand Data-caching) unit
    - Accesses operand data for both fetch (load) or store (update) operations
  - Co-processor unit
    - Supports data compression, cryptographic functions, UTF translations (since zEC12); operates through millicode routine
  - Second-Level Translation and Cache unit
    - Maintains the private second-level translation-lookaside-buffer (TLB2) and cache (L2)
- We will give a high-level overview of the microprocessor design features
  - For more information, see articles that are listed in the reference section near the end

Branch Prediction Unit

- Branch prediction in z processors is performed 'asynchronously' to instruction processing
  - The branch prediction logic can find/locate/predict future occurrences of branch-type instructions (including calls and returns) and their corresponding directions (taken or not-taken) and targets (where to go next) on its own, without requiring / waiting for the downstream pipeline to actually decode / detect a branch instruction
  - The branch prediction logic tries its best to predict the program path much further ahead than where the instruction fetching unit is currently delivering instructions (and should be way ahead of where the execution engines are executing)
- The branch prediction logic employs many advanced algorithms and structures for predicting branching behaviors in program code, as seen in Figure 3, including
  - First-level branch target buffer (BTB1) and branch (direction) history table (BHT1)
  - Second-level target and history buffers (BTB2 and BHT2) (introduced in zEC12) with a pre-buffer (BTBP) used as a transient buffer to filter out unnecessary histories
    - Note: BHT2 is used separately in z196/zEC12 and is fully integrated into BTB2 for z13/z14
  - Accelerators for improving prediction throughput (ACC) by “predicting the prediction” (since zEC12) so it can make a prediction every other cycle (for a limited subset of branches)
  - Pattern-based direction and target predictors (PHT and CTB) to predict how the program will progress based on branch history pattern that represents “how the program gets here”, e.g. for predicting an ending of a branch-on-count loop, or a subroutine return that has multiple callers
  - In z14, both a neural-network-based perceptron engine for enhanced PHT direction prediction and a simple call-return stack for target prediction are introduced for additional accuracy
    - Only branch targets of > 512 bytes away will be considered as a potential call-return pair
- The branch prediction logic communicates its prediction results to the instruction fetching logic through an overflow queue (BPOQ); such that it can always search ahead of where instructions are being fetched

Figure 3: Branch Prediction Structure

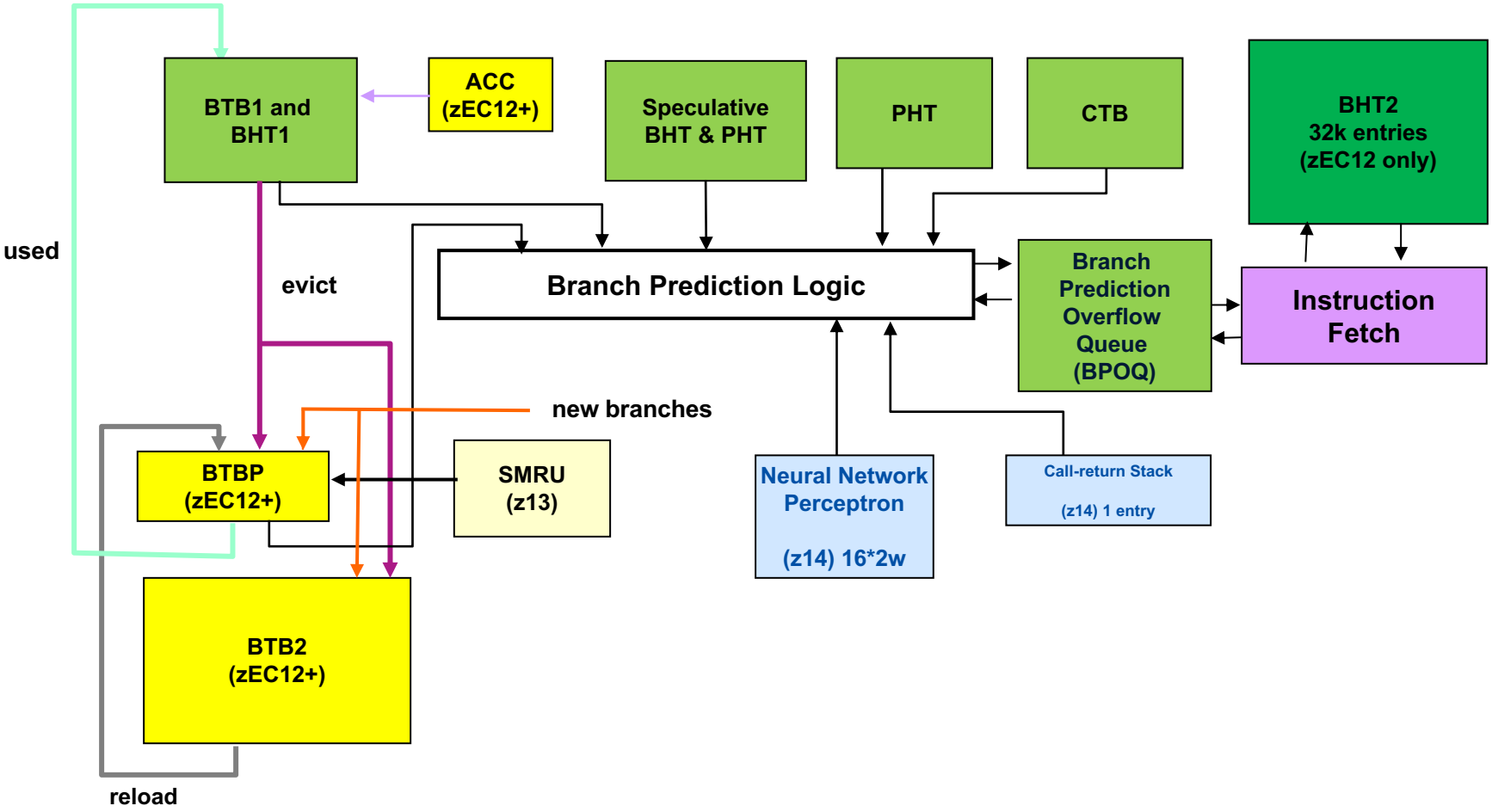


Table 1: Branch Prediction Resources

Label	Structure Name	Description	z196	zEC12	z13	z14
			Rows x Sets (where applicable)			
BTBP	Branch Target Pre-buffer	0.5th level branch instruction address and target predictor look-up in parallel to BTB1, upon usage, transfer to BTB1	NA	128 x 6	128 x 6	128 x 6
BTB1	L1 Branch Target Buffer	1st level branch instruction address and target predictor	2048 x 4	1024 x 4	1024 x 6	2048 x 4
BHT1	L1 Branch History Table	1st level direction predictor (2-bit) : weakly, strongly taken, or not-taken	2048 x 4	1024 x 4	1024 x 6	2048 x 4
BTB2	L2 Branch Target Buffer	2nd level branch instruction address and target history buffer	NA	4096 x 6	16K x 6	32K x 4
BHT2	L2 Branch History Buffer	2nd level direction 1-bit predictor for branches not predicted ahead of time	32 K	32 K	NA	NA
ACC	Column Predictor (z13+) / Fast Re-indexing Table (zEC12)	Accelerate BTB1 throughput in finding the “next” branch	NA	64	1024 (search-based)	1024 (stream-based)
SBHT/ PHT	Speculative BHT & PHT	Speculative direction prediction with transient updates at (out-of-order) resolution time prior to actual completion	3 + 2	3 + 2	8 + 8	8 + 8
PHT	Pattern History Table	Pattern-based tagged direction prediction	4096	4096	1024 x 6	2048 x 4
CTB	Changing Target Buffer	Pattern-based target prediction predicts branches with multiple targets, typically subroutine returns and branch tables	2048	2048	2048	2048
SMRU	Super MRU table (z13+)	Protect certain branches from normal LRU out to make the BTBP more effective	NA	NA	128	128

Instruction Delivery

- Since z/Architecture instructions are of variable lengths (2, 4 or 6 bytes), an instruction can start at any halfword (integral 2-byte) granularity
- Instruction fetching logic fetches “chunks” of storage-aligned instruction data from the instruction cache, starting at a disruption point, e.g. after a taken branch (including subroutine calls and returns) or after a pipeline flush
  - Up to 2 16-byte chunks for z196 and zEC12 and up to 4 8-byte chunks for z13 and z14
- These “chunks” of instruction data are then written into an instruction buffer (as a “clump”) where instructions are extracted (or parsed) into individual z-instructions in program order
- The instruction decode logic then figures out high-level characteristics of the instructions and which/how the execution engines will handle them
  - Is it a storage access? A fixed-point instruction? Which execution units will be involved?
  - Is it a branch-type instruction? If yes, did the branch prediction logic predict that? If not, notify the branch prediction logic (to restart its search) and then proceed based on predefined static prediction rules (e.g. branch-on-conditions are default to be not-taken, while branch-on-counts are defaulted to be taken)
  - Is it going to be implemented in millicode? If yes, did the branch prediction logic predict that? If not, reset the front-end to start at the corresponding millicode routine entry instruction
  - For a complex instruction, does it needs to be “cracked” or “expanded” into simpler internal instructions, called micro-operations (µops)? For example, a LOAD MULTIPLE instruction will be expanded into multiple “load” µops that fetch from storage and write individual general registers (GRs)
- Instructions (and µops) are then bundled to form an instruction group (for pipeline management efficiency), and dispatched (written) into the instruction issue queue

Instruction Cracking or Expansion

- There are multiple reasons for instruction cracking or expansion
- Always (due to inherent multiple operations needed), e.g.

```
– BRANCH ON COUNT (BCTR)     ------------> add register with immediate value of -1
–                                |                              |------>  scratch condition code
–                                |                                                 |
–                                |-----> branch evaluation <-----------------------|
```

- Length based (multiple operations based on length), e.g.

```
– 8-byte MOVE characters (MVC)  ------------------>  load into scratch register
–                                  |------------>  store from scratch register

– 16-byte LOAD MULTIPLE (LM) ---------->  load into register 1
–                              |------>  load into register 2(displacement adjusted at dispatch)
–                              |------>  load into register 3(displacement adjusted at dispatch)
–                              |------>  load into register 4(displacement adjusted at dispatch)
```

- Although the processor pipeline may be “RISC-like”, typical **register-storage** instructions, e.g. “ADD” in example below, are handled efficiently in the design with a feature called “dual issue”, and should be used whenever appropriate
  - ADD: Register1 <= Register1 + memory((Base register) + (Index register) + Displacement)

```
– Register-storage ADD (A) ---------------> load from storage into target register
–                              |                   .. Some cache access cycles later
–                              |-----------> add R1 with target register
```

  - The instruction is **not** considered to be cracked because it is tracked as 1 instruction by using 1 issue queue entry (and 1 global completion table entry), though it is issued to both the LSU and a non-LSU execution unit - hence it's 'dual issued'

Instruction Grouping

- Instructions (and µops) are dispatched (or written) in-order into the out-of-order issue queue as a group. They are then tracked in the global completion table (GCT) until every instruction in the group finishes its processing. When all instructions in a group finish processing, the group is completed and retired
- As instructions (and µops) are grouped, they are subject to various grouping rules, which prevent certain instructions (and µops) from being grouped with others
- During a dispatch cycle, z196 and zEC12 support one group of up to 3 instructions, while z13 and z14 allow two groups of up to 3 instructions
- Some basic rules of grouping
  - Simple instructions, including most “register-register” and “register-storage” type instructions, can be grouped
  - Branch instructions, if second in the group, or if predicted taken, will be the last instruction in the group
    - Best group size if taken branches are the third in a group
  - µops that are expanded from the same instruction will usually be grouped
    - But not with other instructions (or µops) in z196, zEC12
    - If expanded into only 2 µops, can be grouped with one other simple instruction after (in z13, z14)
  - Storage-storage instructions are usually grouped alone, except for the µops that they may be expanded into
  - Other instructions that are alone in a group:
    - Register-pair writers, e.g. DIVIDE (D, DR, DL, DLR), MULTIPLY (M, MR)
    - Non-branch condition code readers, e.g. ADD LOGICAL WITH CARRY (ALC*), SUBTRACT LOGICAL WITH BORROW (SLB*)
    - Explicit floating-point control register readers or writers
    - Instructions with multiple storage operands
    - EXECUTE or EXECUTE RELATIVE instruction or its target
  - Since z13, max group size will be 2 if any µop has more than 3 register sources (including Access Register usage in AR mode)

Instruction Dispatching

- As instructions are dispatched, the source and target architected registers are renamed into a virtual pool of physical registers and are tracked accordingly
  - The amount of rename tracking resources (how many in-flight mappings can be tracked) and physical registers available are key factors of the effectiveness of an out-of-order design
  - In z196 and zEC12, the mapping tracker (the mapper) consists of 1 bucket of 48 mappings
    - GRs: 1 mapping per each 32-bit register write, 1 mapping per each full 64-bit register write
    - FPRs: 1 mapping per each 32-bit register write, 1 mapping per each full 64-bit register write
    - ARs: 1 mapping per each 32-bit register write
  - Since z13, the mapping tracker consists of 2 buckets of 64 mappings each = 128 total mappings
    - GRs: 1 mapping per each 32-bit register write, the GR #'s LSB decides which bucket to use; a 64-bit register write will require 2 mappings, one from each bucket
    - FPRs: 1 mapping per each write, the FPR #'s 2nd LSB decides which bucket to use
    - ARs: 1 mapping per each write, the AR #'s LSB decides which bucket to use
  - Since z13, multiple writes to the same register in the same group does not require separate trackers
- Instructions in a group are dispatched into one of the two issue queues (side 0 and side 1).
  - The total size of issue queue directly relates to the overall out-of-order window and thus affects performance
  - In z196 and EC12, only one instruction group can be written into one of the two queue sides in any cycle; in an alternating fashion
  - Since z13, two groups can be written in any cycle with one group into each side; with the older group on side 0
- The issue queue includes a dedicated "virtual branch queue" since zEC12, 1 per side, that handles relative branch instructions whose targets are less than 64 Kilobytes away
  - These branches will alternate to the different sides of the virtual branch queue independently of the other instructions in the group

Instruction Issue and Execution

- After instructions are dispatched into the issue queues, the issue queues will issue the oldest (and ready) instruction from each issue port to the corresponding execution engine
- Each issue-queue side is connected to a number of specific processing engines, **using z14 as an example in Fig. 4**,
  - There are 5 issue ports (per side; 10 total per core); each to a different engine, including
    - A relative branch unit (**RBU**) handles relative branches
    - A GR writing fixed-point unit (**FXUa**) handles most of the fixed-point arithmetic and logical operations, and also includes a multiply engine and a divide engine (both being non-blocking)
    - A non-GR writing fixed-point unit (**FXUb**) handles other fixed-point operations that do not write any GR results
    - A load/store unit (**LSU**) port, with accesses to the operand data-cache, handles memory accesses
    - A vector & floating-point unit (**VFU**) handles complicated operations
  - Inside each of the **VFU**, there are multiple engines that execute different functions in parallel to each other
    - **BFU** that handles both hexadecimal and binary (IEEE standard) floating-point arithmetic operations, and vector floating-point operations
    - **DFU** that handles decimal (IEEE standard) and quad-precision floating-point arithmetic operations; and since z14, BCD vector convert, multiply, and divide operations
    - **SIMD** that further consists of multiple subunits: PM engine that performs vector permute functions; XS engine that performs fixed-point arithmetic and logical functions; XM engine that performs several multiply functions and ST engine that performs string-related functions
    - **DFX** that handles decimal (BCD) fixed-point arithmetic operations, and since z14, simple BCD vector operations
    - **FPD** that handles divide and square root operations for both binary and hexadecimal floating-point arithmetic
  - Typical pipeline delays through each of the execution engines are shown in Fig. 5
- Differences vs. z13, zEC12, and z196 are shown as colored boxes in Fig. 4

Figure 4: z14 high-level instruction & execution flow

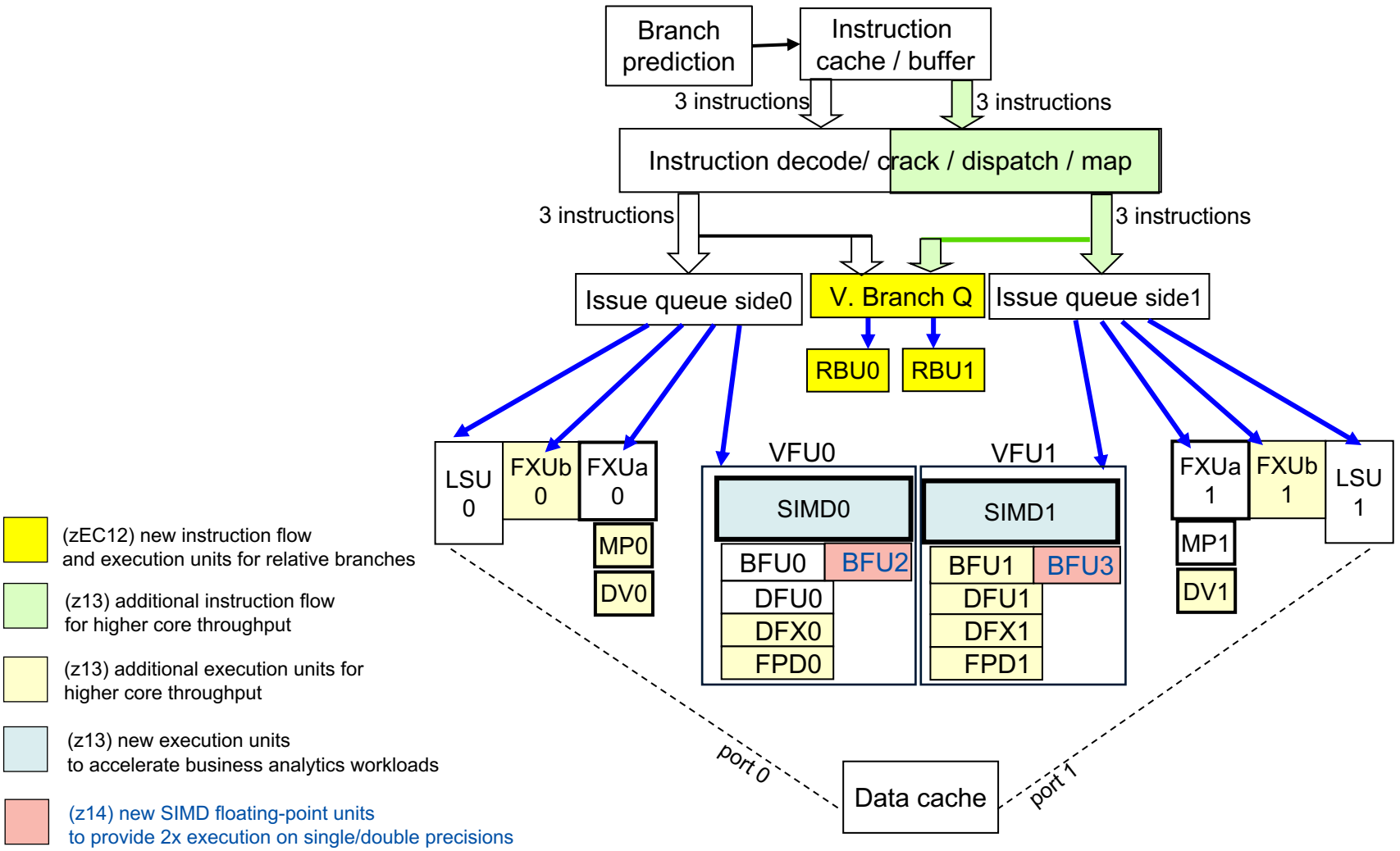


Figure 5: z14 Execution Engine Pipelines

Only 1 of 2 issue sides shown

- **Typical pipeline depths and bypass capabilities shown**
- **Some instructions may take longer to execute or bypass results**
- **Access registers not shown**

ACC – GR access
WB – GR write back

V-ACC – FPR/VR access
VWB – FPR/VR write back

CC – condition code calculation

BYP – data bypass network cycle

FPD, DFU – functions, e.g. divide, square-root, may take multiple passes through the pipeline

G2F – GR to VR/FPR moves
F2G – VR/FPR to GR moves

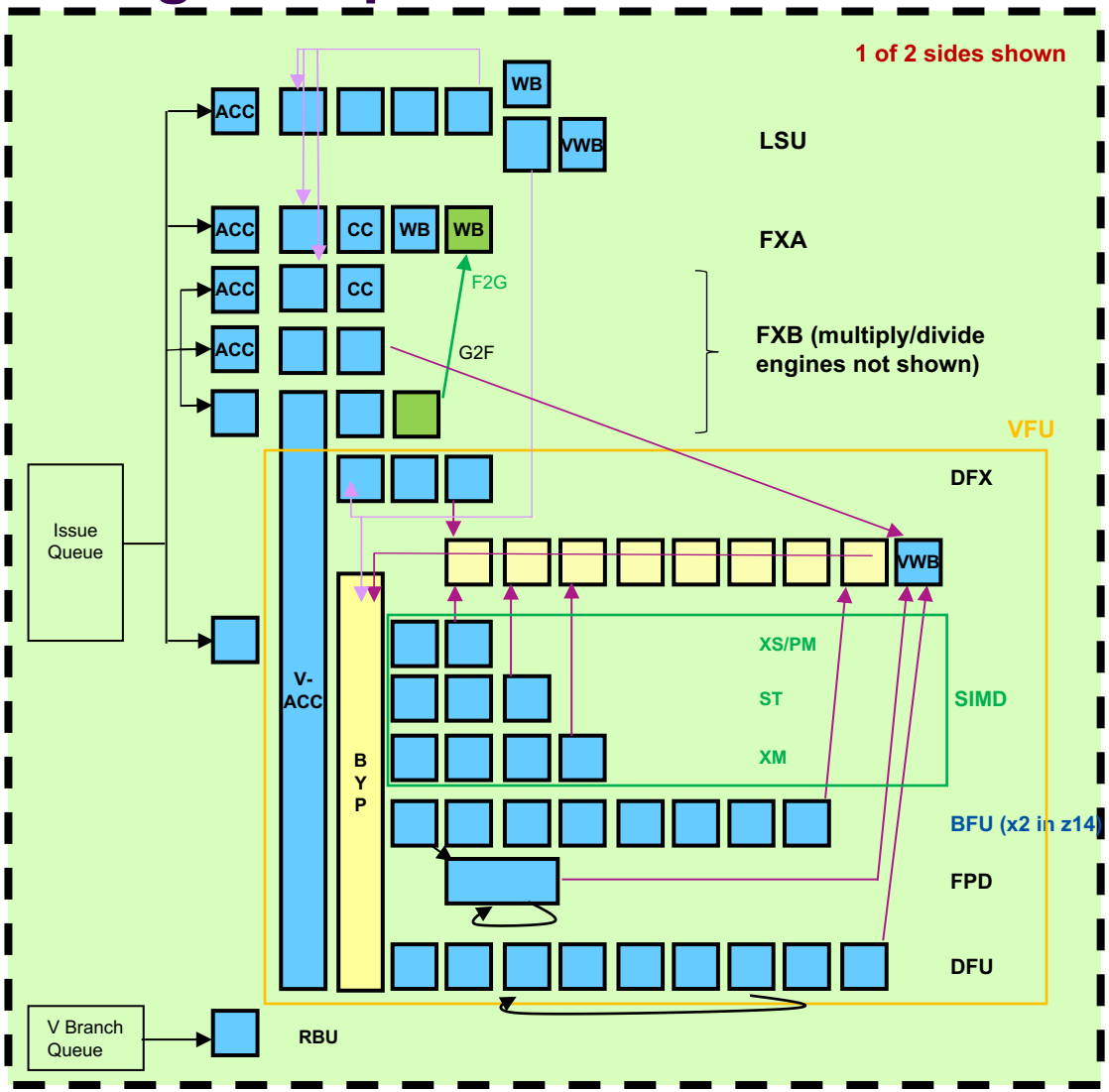


Table 2: Out of order resources

	Z196	zEC12	z13	z14
GR	80 (16 permanently reserved for millicode)	80 (up to 16 reserved for millicode) + 16 immediate value entries	120 (up to 16 reserved for each thread while in millicode) + 8 immediate value entries	Same as z13
FPR / VR(z13)	48 FPRs	64 FPRs	127 FPRs / VRs (up to 8 reserved for each thread while in millicode) + a zero-value entry	Same as z13
AR (access register)	56 (16 permanently reserved for millicode)	56 (16 permanently reserved for millicode)	96 (up to 8 reserved for each thread while in millicode)	Same as z13
Issue Queue	20 x 2 sides	20 x 2 sides + 12 x 2 sides of Branch Queue	30 x 2 sides + 14 x 2 sides of Branch queue	30 x 2 sides + 16 x 2 sides of Branch queue
Global Completion Table	24 x 3 instructions (complete up to 3 instructions/cycle)	30 x 3 instructions (complete up to 3 instructions/cycle)	24 x 2 x 3 instructions (complete up to 6 instructions / cycle)	Same as z13
Unified Mapping Trackers	48	48	64 + 64	Same as z13

The load/store unit

- The load/store unit (LSU) handles the operand data accesses with its L1 data-cache and the tightly coupled L2 data-cache
- The L1 data cache is 2-ported and each port can support an operand access of data elements of up to 8 bytes a cycle
  - There is no performance penalty on alignment except for when the element crosses a cache line boundary
  - Vector elements of more than 8 bytes are accessed in two successive cycles
- In addition to the prefetching of cache misses as part of the natural behavior of the out-of-order pipeline
  - LSU supports software prefetching through PREFETCH DATA type instructions
  - LSU also includes a stride-prefetching engine that prefetches +1, +2 cache-line strides, when a consistent stride is detected between cache miss address patterns at the **same** instruction address across loop iterations
- To minimize pipeline bubbles typically caused by “store-load” dependencies through storage, LSU provides a sophisticated bypass network to bypass pending storage updates that are not yet available in the L1 cache into dependent loads as if the operand data was in L1 (subject to certain limitations). But in general,
  - Data should be bypass-able even if bytes are required from different storing instructions for a load request
  - Data should be bypass-able if the store data is ready a small number of cycles before the dependent load request
  - Multiple mechanisms are used to predict dependencies (based on prior pipeline processing history) between load and store instructions, and will stall load instructions just long enough to enable “perfectly” timed bypasses
  - If a store operation is performed after its dependent load (due to out-of-order operations), a flush occurs
  - If a store operation is performed before its dependent load and the data is not bypass-able (due to timing or hardware limitations), the load is rejected and retried

On-chip Core Co-Processor

- On-chip core co-processors (COPs) are available to enable hardware acceleration of data compression, cryptography, and, on zEC12 and after, Unicode conversions
  - Each COP is private to each core since zEC12, but is shared by two cores in z10 and z196
- The co-processor handles COMPRESSION CALL (CMPSC) instruction that compresses data and cryptographic functions (under the CPACF facility, next page) that support latest NIST standards
  - In addition, Unicode UTF8<>UTF16 conversions are supported in zEC12; and since z13, all Unicode conversions (UTF 8<>16<>32) are supported
- Co-processors are driven through commands of millicode, as it emulates the corresponding complex z instruction
  - Millicode interprets the instruction, tests storage areas, and sets up the co-processor
  - Millicode fetches the source operand
  - Millicode writes source operand data into the co-processor to be processed
  - Millicode sets up result storage areas for co-processor to use - often it is the actual target areas
  - Coprocessor works on the instruction with the provided source data and generates output data
    - For CMPSC, the coprocessor will also fetch dictionary tables accordingly
  - Co-processor writes into the pre-set result storage areas
    - In some cases, millicode will transfer the Co-processor results to the target areas
  - Millicode analyzes status information from the co-processor and repeats work if needed
  - Millicode ends when the instruction (or a unit-of-operation) is completed
- In SMT mode (since z13), the co-processor handles one thread at a time. If the second thread requires the COP, it waits until the first thread finishes an appropriate unit-of-operation or the whole instruction

CPACF - CP Assist for Cryptographic Functions

- Also known as the Message-Security Assist (MSA) instructions
- Runs synchronously as part of the program on the processor
- Provides a set of symmetric cryptographic and hash functions for:
  - Data privacy and confidentiality
  - Data integrity
  - Random Number generation
  - Message Authentication
- Enhances the encryption/decryption performance of clear-key operations for
  - SSL/TLS transactions
  - Virtual Private Network (VPN)-encrypted data transfers
  - Data storing applications

Supported Algorithms	Clear Key	Protected Key
DES, T-DES	Y	Y
AES128	Y	Y
AES192	Y	Y
AES256	Y	Y
AES-GCM(z14)	Y	Y
GHASH	Y	N/A
SHA-1	Y	N/A
SHA-256	Y	N/A
SHA-384	Y	N/A
SHA-512	Y	N/A
SHA-3 224 (z14)	Y	N/A
SHA-3 256 (z14)	Y	N/A
SHA-3 384 (z14)	Y	N/A
SHA-3 512 (z14)	Y	N/A
PRNG	Y	N/A
DRNG	Y	N/A
TRNG (z14)	Y	N/A

Instructions of Interest

- We will discuss some of the instructions in z/Architecture and their handling that might be of general interest:
  - Simple instructions, including descriptions of some interesting ones
  - Special Storage-to-Storage instructions
  - MOVE LONG instructions
  - High Word instructions
  - Conditional instructions
  - EXECUTE instructions
  - BRANCH PREDICTION PRELOAD instructions
  - DATA PREFETCH instructions
  - NEXT INSTRUCTION ACCESS INTENT instruction
  - Atomic and locking instructions
- And a few architecture features:
  - Hardware Transactional Execution
  - Guarded Storage Handling
  - Vector (SIMD) instructions
  - BCD Vector Instructions
- And some storage usage model highlights

Simple Instructions

- Simple instructions
  - Fixed-point results are bypassed without delay into the next dependent fixed-point instruction if the instructions are in the **same side** of the issue queue; otherwise, there will be at least a one-cycle delay
  - An instruction with a storage access operand will need to wait for 4 cycles if the operand is a hit in L1 data cache
  - An operand written by a store instruction to a storage address followed by a load instruction of the same address will require at least 2 to 4 cycles to be bypassed as L1 cache hit data
  - Floating-point instructions are generally pipelined, but can be of different latencies. The design forwards dependent data as soon as it is available
  - Non-floating-point vector (SIMD) instructions (since z13) have shorter latencies than floating-point ones
    - SIMD results are also bypassed when available
- Destructive and non-destructive instructions
  - Many z/Architecture instructions specify just two operands, with one operand doubling as both a source and a target
    - These instructions are shorter (in length) and occupy less space in storage
    - If a register-based operand that will be overwritten is still required after an instruction execution, software must first make a copy of the register before the overwriting instruction
  - Many non-destructive instructions were introduced since z196, such that the register copy operations can be avoided
- Load and Store Reversed instructions
  - To facilitate conversion between big-endian (BE) and little-endian (LE) formats, a few instructions are provided to reverse the byte ordering of a data element to/from memory
  - Both load and store operations are supported
  - 2, 4, and 8-byte operands are supported
  - MOVE INVERSE (MVCIN) is also available for more than 8-bytes storage to storage data swap
    - Millicode implements this instruction by doing a byte-by-byte copy

Special Storage-to-Storage Instructions

- z/Architecture includes a set of storage-storage instructions in which the data size is specified in the instruction as the length field
  - Mostly defined to be left-to-right and byte-at-a-time operations
  - Special hardware is being used to speed up certain common cases
- MOVE Characters (MVC)
  - If <= 16 bytes, it is cracked into separate load and store µops
  - If > 16 bytes, it is handled by sequencing logic inside the LSU
  - If the destination address is 1 byte higher than the source address (and they overlap), it is special cased into hardware as a 1-byte storage-padding function (with faster handling)
  - If the destination address is 8 bytes higher than the source address (and they overlap), it is special cased into hardware as an 8-byte storage-padding function (with faster handling)
  - For other kinds of address overlap, it will be forced into millicode to be handled a byte at a time
  - Since z10, special case detection is done at decode time, not detected during address generation, and thus requires the instructions to have B1=B2

– COMPARE LOGICAL Characters (CLC)
  - If <= 8 bytes, it is cracked into separate load and compare µops
  - If > 8 bytes, it is handled by the sequencing logic inside the LSU

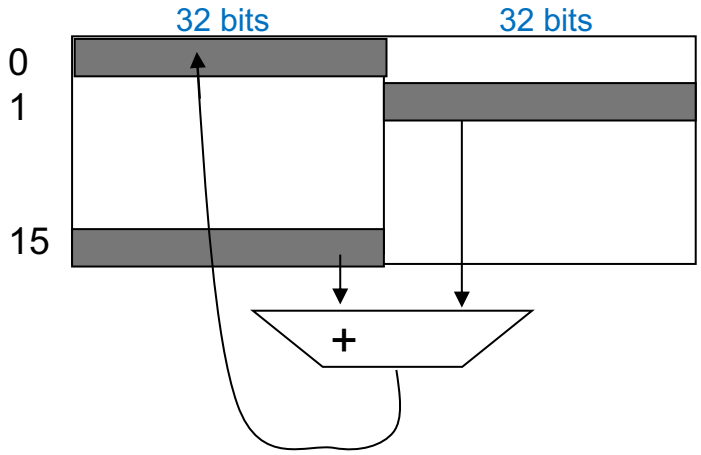
- EXCLUSIVE OR Characters (XC)
  - If <= 8 bytes, it is cracked into separate load and “or-and-store” µops
  - If > 8 bytes and if base register values and displacement values are equal, i.e. an exact overlap on addresses, it is special cased into hardware as a storage clearing function (with faster handling)
    - Since z10, this special case detection is done at decode time, not detected during address generation, and thus requires the instructions to have B1=B2 and D1=D2
  - If > 8 bytes and no exact overlap on addresses is detected, it is handled by sequencing logic inside the LSU
  - For other kinds of address overlap, it will be forced into millicode to be handled a byte at a time
  - AND Characters (NC) and OR Characters (OC) instructions are implemented similarly, without the special clearing function

MOVE LONG Instructions (MVCL*)

- MOVE LONG instructions can copy a large amount of data from one storage location to another
- A special architected functional variant can also be used to pad storage
- These instructions are implemented in millicode
- A special engine is built per CP chip for aligned copying or padding functions at a page granularity
  - The page-aligned copying or padding is done “near memory”, instead of through caches, if
    - Not executed inside a transaction
    - Padding character specified is neither X’B1’ nor X’B8’
    - A preceding NIAI instruction does not indicate that the storage data will be used subsequently
    - The operands must not have an access exception
    - Length >= 4K bytes
    - For moves: source and destination addresses are both 4K-byte aligned
    - For padding: destination address is 4K-byte aligned
  - Otherwise, the move process will operate through the caches (L1, L2…)
  - Note that the evaluation is revised every unit-of-op
    - For padding, even if starting address is not aligned, millicode pads in cache to the first 4K-byte boundary, then uses “near memory” pad engine for the next aligned 4K-byte pages until the remaining length is less than 4K bytes. After that, padding is done in cache again
- Near-Memory engine usage is best when the amount of data involved is large and the target memory is not to be immediately consumed in subsequent processes
  - Since the special engine is shared within a CP chip, contention among processors is possible
  - Such contention is handled transparently by millicode and additional delay may be observed

High Word Instructions

- Provided since z196
  - Intended to provide register-constraint relief for compilers
- High words of GRs are made independently accessible from the low words of GRs
- Software can use up to 32 word GRs, 16 doubleword based GRs, or combination of word and doubleword GRs
- For register dependencies, including address-generation interlocks, the high-word GRs are treated separately from the low-word GRs
- Various types of operations are supported
  - Add, subtract, compare, rotate, load, store, branch-on-count



Conditional Instructions

- In many applications (for instance, sorting algorithms), conditional-branch outcomes are highly data dependent and thus highly unpredictable
  - A mispredicted branch can result in a pipeline flush, and may incur many cycles of branch correction penalty
- A limited set of conditional load/store instructions are provided (z196+) where the execution is predicated on the condition code
  - Highly **unpredictable** branches can be replaced with conditional instructions
- In the example, the old code shows a COMPARE register instruction (CR) followed by a BRANCH ON CONDITION instruction (BRNE for BC), and a LOAD instruction (L) that may or may not be executed depending on the outcome of the branch
- The new code sequence replaces the branch and load instructions with a LOAD ON CONDITION (LOC) instruction
  - It is cracked into a load from storage and a conditional select µop
  - The conditional select µop uses the condition code to select between the original register value and the new value from storage
  - This sequence now avoids potential branch wrong flushes

NOTE: Access exceptions may be reported whether the storage content is effectively accessed or not

```
Old Code
        CR   R1, R3
        BRNE skip
        L    R4, (addressX)
skip    AR   R4, R3
        ..



New Code
        CR   R1, R3
        LOC  R4,(addressX), b'0111'
        AR   R4, R3
        ..


*Pseudo-code for illustration only
```

EXECUTE Instructions

- "Execute" instruction is commonly used* with targets being storage-related instructions (e.g. MVC, CLC mentioned before) where the length field (specifying the number of bytes) can be substituted with the contents of a general register (GR) without actually modifying the instruction in memory (and without explicit branch to or from the 'target' instruction)
- "Execute" is handled by the processor like a branch, by
  - Jumping to the target of the execute instruction as a branch target, and fetching it
  - Decoding and executing the target instruction (modifying as needed)
  - Immediately returning back to the subsequent instruction after the "execute" (except when the target is a taken branch itself)
- This "implied" branch handling is supported by the branch prediction logic to reduce the overall processing delay
- Certain pipeline delay is required between the reading of the GR and the "modification" of the target instruction
  - The delay is reduced since z13 for a selected group of instructions: MVC, CLC, and TRANSLATE AND TEST (TRT)
- When the operand length is mostly random during run-time, the alternative of using a branch table is not preferred due to its potential inaccuracy in branch prediction

```
Example where MVC's length depends
on compare of R1 and R3:

        LHI   R4, x'1'
        LHI   R5, x'2'
        CR    R1, R3
        LOCR  R4, R5, b'1000'
        EX    R4, move
        ..

move    MVC   0(length, R13), 0(R14)

*Pseudo-code for illustration only
```

*other **tricky** EXECUTE usages are not discussed here; e.g. in modifying register ranges, lengths of operand 1 or operand 2, and branch masks

BRANCH PREDICTION PRELOAD Instructions

- BRANCH PREDICTION PRELOAD (BPP) and BRANCH PREDICTION RELATIVE PRELOAD (BPRP) instructions introduced with zEC12 specify the location of a future to-be-taken branch and the target address of that branch
- By providing such directives to the hardware's branch prediction logic, the limitation of the hardware branch table's capacity may be overcome
  - The processor may now predict the presence of branches without having seen them before or if their history was displaced
  - The directives will not override or modify an existing hardware history entry's target address
- As described earlier, the branch prediction logic should always search ahead 'asynchronously' of where in the program instructions are currently being decoded and executed
  - Just like requesting a stop on a bus, the request needs to be activated BEFORE the bus passes the desired stop; to be effective, the preload instruction needs to be executed **before** the prediction logic may search pass the branch address to be effective
  - The preload instructions are thus best used when the program's run-time behavior involves a lot of somewhat cold modules; such that (taken) branches are likely not being predicted and the instructions are likely not in the cache; such that the preload instructions can have a good chance of being executed AHEAD of the search logic
  - The actual usage of the preload instruction is therefore most effective when in conjunction with profile-directed feedback (PDF), or in a JIT environment where the run-time characteristic can be extracted and analyzed
- The more (taken) branches in-between and the further away in sequential memory address, the more likely a preload will succeed
  - At a minimum, the target branch should be more than 1 (taken) branches and 256 sequential bytes away
- The relative form of preload instruction, BPRP, should be used if possible as it can be activated earlier in the pipeline, providing a better chance of being effective
- The preload mechanism may also perform an instruction cache touch (and thus a potential prefetch) on the branch target
  - Do not use for purely instruction cache prefetches, as that will pollute the branch prediction history structure

PREFETCH DATA Instructions

- Starting with z10, PREFETCH DATA (PFD) and PREFETCH DATA RELATIVE LONG (PFDRL) instructions were introduced to enable program code a way to manipulate the local data cache
  - It is architecturally a no-op
- The provided prefetch function allows code to potentially acquire a cache line (into L1) in a correct cache state (for read-only or for write) ahead of the actual load/store instructions that will access the data
  - Note: prefetching a cache line that is contested among multiple processors is usually a bad idea
- These prefetch instructions not only allow operand data prefetching, they also provide a way to release a local cache line's ownership (also known as untouch)
  - The untouch function is to allow software code to proactively release (or invalidate) its ownership (from the processor that it is running on) of a specified cache line; as such it can be used when done using a shared data structure
  - Such that, when a different processor accesses this same cache line some time later, the shared cache (L3/L4) will not need to spend time in removing the line from this "last-owning" processor before granting ownership to the "newly-requesting" processor
- These directives should be used carefully, and some experimentation may be required to yield desired performance effect
  - Prefetch function can be redundant with given hardware capabilities
    - The out-of-order pipeline inherently performs "baseline" prefetching
    - The stride-prefetch engine also prefetches cache lines based on fetching patterns and miss history
    - The L4 cache does limited prefetching from memory based on certain miss criteria
    - Prefetch can hurt if the cache line is contested with other processors
  - Untouch function can be tricky to use
    - If it is a highly contested cache line, demote operation might hurt (by adding more related operations to the system)
    - If the cache line is cold, it might not matter
    - In general, the demote variant (code 6) is preferred to the full untouch variant (code 7) since it usually incurs less overhead; as it can be used when done updating a shared data structure
- NOTE: EXTRACT CPU ATTRIBUTE (ECAG) instruction should be used, instead of hardcoding any cache-related attributes, to minimize the chance of observing adverse effects on different hardware models

NEXT INSTRUCTION ACCESS INTENT (NIAI) Instruction

- The NIAI instruction was introduced in zEC12 for program code to provide some hints to the cache system on the intention of the next immediate instruction's operand accesses, so the hardware can adjust its related handling
  - The instruction behaves like a "prefix" instruction but architecturally it is a separate (no-op) instruction
  - Hints will also be passed into instructions that are implemented in millicode
- The cache subsystem provides heuristic to maintain cache ownership among multiple processors
  - Upon a cache miss from a "current" processor core for a "fetch-only" (non-storing) instruction, the cache subsystem may return an exclusive state if the cache line was previously updated by a "previous" processor
  - This design anticipates that this "current" processor will likely follow suit of the "previous" processor and store to the cache line after this fetch-only miss, saving coherency delays (otherwise seen when changing from a shared state to an exclusive state)
  - In the case where the heuristic is not working perfectly, e.g. when there are multiple "readers" on a cache line, the NIAI instruction (code 1 - "write") can be used by a "writer" process to indicate subsequent store intention upon an initial fetch
- The NIAI instruction can also be used to indicate "truly read-only" usage of a cache line.
  - Given the "reader and writer" processes described above, a NIAI (code 2 – "read") can be used to specify the read-only intention of the consumer (or reader) process's accesses to a cache line; thus preventing the line from potentially migrated to the reading processor as exclusive (write) ownership
  - The hint can now help reduce the coherency penalty on the next round when the producer (or writer) process is writing into the cache line again
- Cache lines are usually managed from most recently used (MRU) to least recently used (LRU) in the cache, so lines that have not been used recently are evicted first when new cache lines are installed
  - This scheme generally works well, but is suboptimal in cases where the process is operating on streaming data where data is only accessed once and then becomes uninteresting
  - In these streaming cases, it is desirable to label such data as LRU so that it's not retained at the expense of other data that will be used again
  - The NIAI instruction (code 3 – "use once") can be used to indicate streaming data accesses such that the local cache will keep those data in compartments that will be evicted sooner

Atomic and Locking Instructions

- z/Architecture provides a set of instructions that can be used for atomic operations
  - e.g. TEST AND SET (TS), COMPARE AND SWAP (CS)
  - They check a value in storage (fetch) and then conditionally update the storage value (store) such that the fetch and the store are observed to be "atomic", meaning to an outside observer the two actions appear to have occurred simultaneously
- The Interlocked-Access Facility instructions were added on z196
  - Load and "arithmetic" instructions for unconditional updates of storage values
    - (Old) storage location value loaded into GR
    - Arithmetic or logical operation (*add, and, xor, or*) result overwrites value at storage location
    - Best for unconditionally updating global information, like a counter or a flag
  - Interlocked storage updates with an immediate operand are also supported
    - Supported operations include *add*, *and*, *xor* and *or*
  - LOAD PAIR DISJOINT (LPD, LPDG)
    - Load from two different storage locations into GR N, N+1
    - Condition code indicates whether the fetches were atomic
- Hint: For software locks, if the lock is likely concurrently used by multiple processors (i.e. often contested), the following sequence should be considered
  - It is more desirable to test the lock value before using atomic instruction (e.g. CS) to set the lock

```
        LHI   R2, 1          ; value to set lock
LOOP    LT    R1, lock       ; load from memory and test value; always test first
        BCR   14,0           ; serialization to guarantee getting new value
        JNZ   LOOP           ; repeat if non-zero
        CS    R1, R2, lock   ; set lock if lock was empty
        JNE   LOOP           ; retry if lock became set
*Pseudo-code for illustration only; see additional discussion in page 62
```

Hardware Transactional Memory

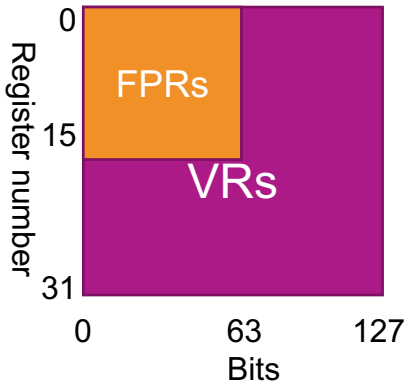
- Beginning in zEC12, z/Architecture supports hardware transactional (memory) execution through the ***T***ransaction e***X***ecution facility, occasionally referred to as TX
  - A group of instructions can be observed to be performed with atomicity, or not done at all (aborted)
  - Non-transactional stores are allowed within a transaction
  - A form of constrained transaction (transaction with restrictions) is also supported where the hardware will automatically retry an aborted/failed transaction until the transaction is successful
  - Optional detail debug data can be provided
- Transaction usage is not advisable if the contention of used storage is already high
  - Likely end up wasting CPU cycles if the transaction keeps aborting due to real-time cross-CPUs memory access contentions
  - Aborts are expensive (>200 cycles) and worse if abort debug information is requested
- Hint: compute complex results outside of a transaction, then use transaction with only a small number of instructions to check data, and then store the results away
- Access (fetch) footprint* is limited by L2 cache associativity and size
  - Up to 1 Mbyte in zEC12, 2 Mbyte in z13, 4 Mbyte in z14
- Update (store) footprint* is limited by L2 cache associativity and size of an internal store buffer
  - The buffer design can support up to 64 blocks of 128-byte (storage-aligned) data changed within a transaction
  - The L1 data cache is updated as each store instruction completes within a transaction, but L2 cache update from the buffer is deferred until transaction completes
- Note: Access footprint may be counted for fetches done through mispredicted branches. Footprint limitations are shared by the 2 threads when SMT2 is enabled such that effective footprint may be smaller than when one thread is running

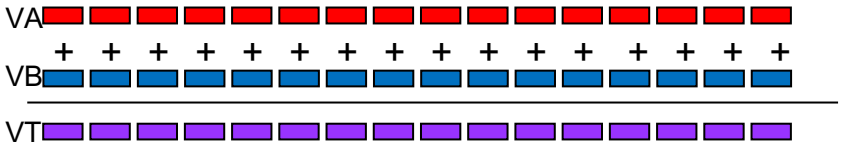
Guarded Storage Handling

- Beginning in z14, z/Architecture provides the guarded-storage facility for programming languages to more efficiently implement storage-reclamation techniques commonly referred to as garbage collection (GC)
  - Allows application/user threads to continue running concurrently during phase of GC compaction
  - Accomplished by providing hardware-assisted read barriers for guarded storage (GS) involved in a compaction event
- Prior to compaction, software defines a "range of guarded-storage segments" by specifying the GS starting/ending address and segment size to cover the entire region containing all guarded segments. A bit mask vector is set to define which segments are actually protected.
  - Done through the LGSC (LOAD GUARDED STORAGE CONTROLS) and STGSC (STORE GUARDED STORAGE CONTROLS) instructions
- When guarded storage is set up and enabled
  - When the second operand of the LGG (LOAD GUARDED) or LLGFSG (LOAD LOGICAL AND SHIFT GUARDED) instruction designates a GS segment as specified by software, a guarded-storage event is recognized, and control is passed to a guarded-storage event handler
  - Otherwise, the respective instructions perform their defined load operations
  - All other instructions that access a segment of guarded storage are unaffected by the facility
    - Only the new LGG and LLGFSG instructions can potentially generate a guarded-storage event
- It is expected that the problem-state guarded-storage event handler would
  - First save program GRs, move offending data to a non-guarded area and fix up any pointer addresses if necessary
  - Then after restoring program GRs, it will branch back to the interrupted program address previously saved during the GS event and continue normal program operations
- NOTE: Some refer to this feature as 'pause-less garbage collection', where 'pause-less' should be understood to mean 'less-pausing' and *not* 'pause-free'

Single-Instruction-Multiple-Data (SIMD)

- SIMD instructions, sometimes also referred to as vector instructions, were introduced in z13
  - See Silvia Mueller's z13 SIMD presentation* or Eric Schwarz's Journal article[11]
  - More instructions are added with z14
- To support these instructions, new vector registers (VRs) are architected
  - 32 x 128-bit architected registers are defined per thread
  - FPRs overlay VRs as follows:
    - FPRs 0-15 == Bits 0:63 of SIMD registers 0-15
    - Update to FPR <x> alters **entire** SIMD register <x>
    - Whenever an instruction writes to a floating-point register, bits 64-127 of the corresponding vector register are unpredictable
- Each SIMD instruction provides fixed-sized vectors ranging from one to sixteen elements
  - Some instructions operate only on a subset of elements
- The use of vector compares and vector select operations can help avoid unpredictable branch penalties similar to the simple conditional instructions described earlier





*http::://www-03.ibm.com/systems/dk/resources/T1_D1_S4_z13_SIMD_LSU_2015.pdf

Table 3: Types of SIMD instructions

Integer

16 x 8b, 8 x 16b, 4 x 32b, 2 x 64b, 1 x 128b

- 8b to 128b add, subtract
- 128b add/subtract with carry
- 8b to 64b minimum, maximum, average, absolute, compare
- 8b to 16b multiply, multiply/add
  32b x 32b multiply/add
- Logical operations, shifts
- Carry-less multiply (8b to 64b), Checksum (32b)
- Memory accesses efficient with 8-Byte alignment; minor penalties for byte alignment
- Gather / Scatter by Step; Permute; Replicate
- Pack/Unpack
- (z14) Pop-count (8b to 64b)
- (z14) Large integer multiplication for matrix operations that are used in cryptography

String

- Find 8b, 16b, 32b, equal or not equal with zero character end
- Range compare
- Find any equal
- Isolate String
- Load to block boundary - load/store with length (to avoid access exceptions)

Decimal (z14)

- Register-based versions of storage-based arithmetic/convert instructions
- Multiply-shift, shift-divide

Floating-point

32 x 2 x 64b

- Binary Floating-Point operations with double precision
- 2 BFUs with an effective increase in architected registers
- All IEEE trapping exceptions reported through VXC, and will not trigger interrupts
- Compare/Min/Max (per language usage) added in z14
- Two more precision binary floating-point operations are added in z14

32 x 4 x 32b
32 x 1 x 128b

BCD Vector Register-based Instructions

- BCD stands for binary-coded decimal
- In z/Architecture, decimal numbers may be represented in formats that are of variable length.
  - Each byte of a format consists of a pair of 4-bit codes; the 4-bit codes include decimal digit codes, sign codes, and a zone code
- New architecture is introduced in z14 to support BCD operations through vector registers, reducing memory accesses:
  - Load, store with length right-aligned
  - BCD arithmetic
- Special features that are targeted to COBOL (vs. existing BCD instructions)
  - Static & dynamic length specification
  - Elaborate signed / unsigned control
  - Condition code setting is optional
  - New variants of multiply and divide
- NOTE: Available with COBOL V6.2 and ARCH(12) compilation switch; NOTE: if ARCH() isn't specified the default is ARCH(7) for z9!

	OLD	NEW (since z14)
Function	Classic, storage-based	VR-based
Add	AP	VAP
Subtract	SP	VSP
Zero & add	ZAP	VPSOP
Compare	CP	VCP
Test	TP	VTP
Shift & round	SRP	VSRP
Multiply/multiply-shift	MP	VMP, VMSP
Divide/remainder/shift-divide	DP	VDP, VRP, VSDP
BCD → Binary	CVB*	VCVB*
Binary → BCD	CVD*	VCVD*
Load immediate		VLIP
Load rightmost		VLRL(R)
Store rightmost		VSTRL(R)

Uniprocessor Storage Consistency

- Uniprocessor view of 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 storage accesses must be observed to be done in program order
    - Store / fetch conflicts are recognized by **real*** address
    - Most operands are processed left to right
    - Fixed-point decimal operands are processed right to left
    - Storage-storage (SS) instructions are observed to operate in a byte-by-byte fashion
  - Instruction pre-fetches may be observed
    - Must still detect store updates / instruction fetch conflicts; where detection is on **logical*** address only
    - Instructions executed must reflect prior stores
    - Serialization can add further restrictions (see details in next page)

*Logical address
  - What program specifies
  - May be virtual or real, depending on DAT (dynamic address translation) mode specified in the program status word (PSW)
    - Unless explicitly overridden by the instruction itself (see detail instruction definitions)

*Real address
  - Result of dynamic address translation process or the logical address when DAT mode is off in PSW
  - Subject to prefixing

Multiprocessor 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 results (See the section “Storage-Operand Consistency” in the z/Architecture Principles of Operation for details)
- Operand fetches and stores must appear to occur in proper order
- All processors must obey uniprocessor rules
  - Although the processor is designed to do things out-of-order, the observed results must be consistent
  - The processor has states and checking in place, such that when the out-of-order accesses might be observed to be inconsistent, the pipeline will flush and retry the operations; possibly in a “safer” (slower) mode
- Operand accesses must be DW-consistent
  - No "score-boarding" should be observed
  - e.g., DW consistency is maintained for LOAD MULTIPLE (LM) when the loads are expanded into individual GR writing operations
- Instruction fetches are generally allowed in any sequence

```
CPU1                CPU2

Store  R1, AA       Store   R1, BB
Load   R2, AA       Load    R2, BB
Load   R3, BB       Load    R3, AA

As an example, if both final Load instructions get “old” (pre-store) values : Violation!
```

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
- See additional discussion in page 62 “frequently asked questions (2)”

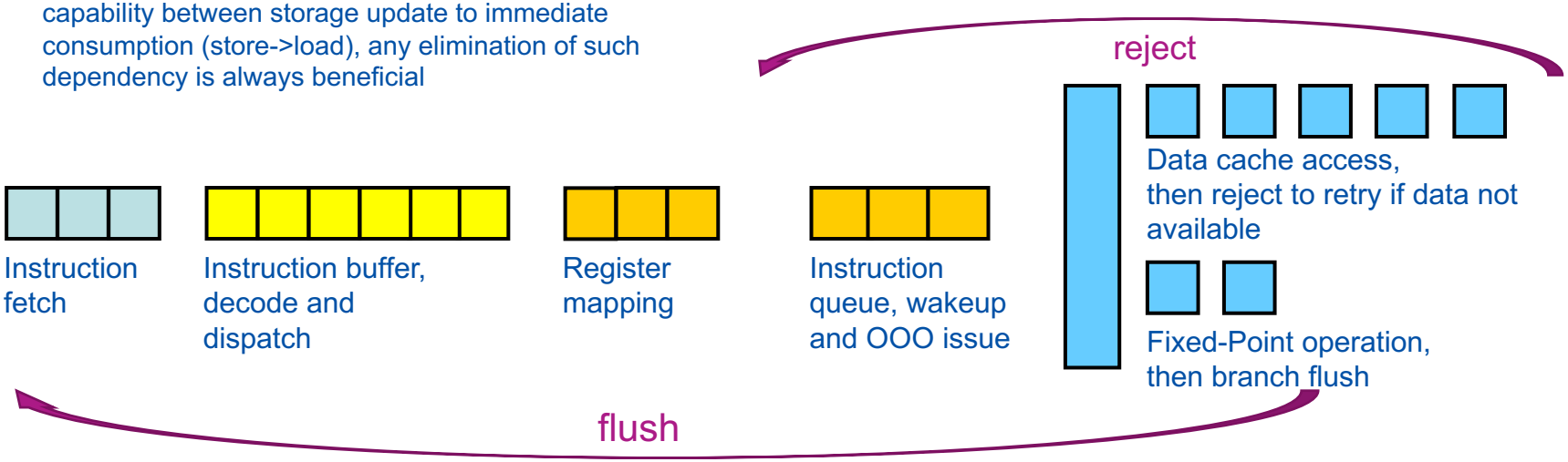
```
CPU 1                    CPU 2

MVI   A, X'00'           G   CLI   A, X'00'
BCR   14, 0                  BNE G

The BCR 14, 0 instruction executed by CPU 1 is a serializing instruction that forces the store
by CPU 1 at location A to become visible to other CPUs. However, per the architecture, CPU 2 may
loop indefinitely, or until the next interruption on CPU 2, because CPU 2 may already have
fetched from location A for every execution of the CLI instruction. The architecture requires
that a serializing instruction be placed in the CPU-2 loop to ensure that CPU 2 will again fetch
and refresh the data value from location A.
See additional discussion in page 62 “frequently asked questions (2)”
```

General Optimization Guidelines

- See references in page 66 and 67 for additional information on topics you might be interested
- CPU Measurement Facilities (CPUMF) are provided with user-accessible hardware instrumentation data to understand performance characteristics
  - Documentation and education materials can be found on online with IBM's supporting websites
  - Extracted data is presently made available in z/OS through SMF 113 records and in z/VM through MONWRITE data
- Some general recommendations will be provided next, including some that have been mentioned in previous pages
  - All descriptions provided are of general guidance only
  - It will not be practical to describe all intricate design details within the systems in this document
  - There may be counter-examples (usually rare occurrences) that will observe hardware behavior differently than described, or not adhere to optimization recommendations provided
  - Detail instruction-by-instruction classifications and timings will not be provided in this document
- Z processors are designed for processing both cache-intensive and CPU-centric workloads and are optimized to handle code that was hand-written many years ago or was generated from the latest compilers, with that code running in applications, middleware or operating systems
  - However, code that was hand-written or generated using older versions of compilers many years ago could benefit from a scrub or refresh as there are very likely new instructions and optimizations that would further improve code performance
  - General rules that help produce good performance code for modern processor microarchitectures usually apply to z processors too
  - Microprocessor pipeline, branch prediction algorithm, cache subsystem structure, and their characteristics will likely change from generation to generation to obtain better general performance improvements and bigger system capacity
  - Code sequence can be tuned to get more performance by optimizing to a new processor pipeline, or by using new instructions or new architectures
  - Performance variations should be expected on highly optimized code that is tuned to a specific processor generation vs. another generation

Optimization - General Pipeline

- Recent z microprocessor cores have fairly deep instruction pipelines
  - Driven by high-frequency design (up to 5+ GHz since zEC12)
  - Latest z13/z14 pipeline: 20+ cycles from instruction fetch to instruction finish
- Pipeline hazards can be expensive
  - Branch wrong flush (for either direction or target) – 20+ cycles
  - Cache reject when cache data is not available or cache resources are not available – 12+ cycles
- Code optimization technique can help, for example (more guidelines provided in subsequent pages):
  1. Arrange frequent code in “fall through” paths
     Although the z processors improve upon branch prediction design every generation to get better accuracy and larger coverage, allowing the frequent code in a ”straight-line” sequence is always beneficial
  2. Pass values via registers rather than storage
     Although the z processors improve data bypassing capability between storage update to immediate consumption (store->load), any elimination of such dependency is always beneficial



Optimization - Branch Handling (1)

- Align frequently called functions to start at storage boundaries for efficient instruction fetching
  - at least at quadword (16-byte) boundary, but potentially even better if at octword (32-byte) or cache-line boundaries
- Rearrange code path around conditional branches such that the not-taken path (i.e. fall-through path) is the most frequent execution path
- Although the branch predictor attempts to predict every cycle, keeping loops to be at least 12 instructions will allow branch prediction to catch up
  - If more instructions can be used, branch prediction will be able to stay ahead of instruction fetching
- Although z processors before z14 do not include a call-return predictor, z14 included a simple call-return predictor such that pairing up calls and returns, i.e. no nesting of calls, may facilitate the design to work more effectively
- Consider in-lining subroutines if they are small and used often
- Unroll loops to parallelize dependency chains to maximize the advantage of parallel and out-of-order processing
- Use relative branches instead of non-relative (indirect branches) when possible
- There is usually an advantage to use a branch-on-count or a branch-on-index type instruction versus doing the operations as individual instructions, due to
  - Smaller instruction footprint and less hardware overhead
  - Branch-on-count type and branch-on-index-low-or-equal type instructions are predicted taken whenever the branch prediction logic is not able to predict its direction ahead of time
- Similarly, load-and-test or compare-and-branch type instructions will perform better than a pair of individual instructions
- Avoid hard-to-predict branches by using conditional instructions
  - Conditional instruction is usually slower than a correctly predicted branch + load/store instruction; thus "hard-to-predict" is an important criteria

Optimization - Branch Handling (2)

- The main branch prediction structure (BTB1) is managed on a **32-byte block** basis per set, the total number of active (and taken) branches that can be saved is therefore limited by the number of sets in the set-associative design.
  - For example, in z14, when the BTB1 is designed as a 4-way set associative structure, it can keep only 4 (taken) branches in history for prediction
    - SMT-2 operations can cause additional conflict misses vs. single-threaded operations
  - To account for potential SMT-2 conflicts, for a branch table that includes multiple frequently taken branches, try to limit the number of branches to 2 per 32-byte region by
    - Rearranging the branch table with intermixed infrequent entries
    - Padding the branch table entries with no-op instructions

Optimization - Instruction Selection (1)

- Register-storage format instruction is often more efficient than a 2-instruction sequence of “load” + “register-register” operations
- Use instruction variants that do not set condition code if available (and when the resulting condition code is not required)
- Use instructions of shorter instruction lengths if possible
- An instruction accessing storage by using a Base + Index + Displacement form (3-way) of address generation incurs no additional penalty vs. a Base + Displacement form (2-way) or a register-based form
  - Similarly, Base + Index + Displacement form for branch target calculation incurs no additional delays vs. a register form; e.g. BC vs. BCR
  - Precompute storage address only if you can use it for branch prediction preloading or operand data prefetching
  - However, “Load Address” type instructions will take an extra cycle through the FXU when both base and index registers are not using GR#0
- Understand rotate-then-*-selected-bits instructions, and see whether they can be used
  - The second-operand register is rotated left by a specified amount; then one of four operations (*and, xor, or, insert*) is performed using selected bits of the rotated value and the first-operand register
- Use compare-and-trap instructions where practical, in particular, for null-pointer checking
- Take advantage of the additional high-word GRs instead of performing register spill-and-fill through storage
  - Since z13, VRs might also be used
- Regular register clearing instructions are fast-pathed in the pipeline, and their results do not use any physical register renames (since zEC12)
  - SUBTRACT or EXCLUSIVE OR register (SR/SGR, XR/XGR of the same register); which sets CC=0
  - LOAD HALFWORD IMMEDIATE (LHI, LGHI of immediate value 0..7), which leaves CC unchanged
  - LOAD ADDRESS (LA) where Base, Index, and Displacements are all zero’s
  - And, since z13, LOAD ZERO {long}, {extended} (LZDR, LZER)

Optimization - Instruction Selection (2)

- Use the long-displacement variants, with a 20-bit signed displacement field, that provide a positive or negative displacement of up to 512K bytes if necessary
- A set of instructions (ends with RELATIVE or RELATIVE LONG) is provided to operate on data elements where the address of the memory operand is based on an offset of the program counter rather than an explicitly defined address location. The offset is defined by an immediate field of the instruction that is added (as a sign extended, halfword-aligned address) to the value of the program counter
  - Load, store, and various kinds of compares are provided
  - Such accesses are treated as data accesses (except for EXECUTE RELATIVE LONG); these data elements should not be placed anywhere near the same cache lines as the program instructions to avoid potential cache conflicts
- For operations on large amounts of memory, e.g. copying or padding storage, consider using instructions that can handle long operand lengths, e.g. MOVE characters (MVC), instead of doing individual loads or stores
- Complex instructions, e.g. COMPRESSION CALL (CMPSC), convert-UTF-UTF instructions, and cryptographic instructions, with help of the per-core co-processor, are usually faster than software routines especially for large datasets
- For serialization, a BCR 14, 0 (supported since z196) is better than BCR 15, 0 (which also requires checkpoint synchronization needed for software checkpoints that might incur additional delays)
- For storing clock value, use STOCK CLOCK EXTENDED (STCKE); if uniqueness is not required, use STORE CLOCK FAST (STCKF)
  - Design of z14 further optimizes towards STCKE/STCKF, STCK usages might observe relative slow-down
- Use simple “interlocked-access” instructions, e.g. LOAD AND ADD (LAA), OR/AND/XOR immediate (OI, NI, XI), instead of conditional loops using compare-and-swap type instructions, for any **unconditional** atomic updates
  - OI, NI, XI (and their long displacement variants, OIY, NIY, XIY) were used in examples that did not interlock in earlier architecture; these instructions are now interlocking since z196

Optimization - Instruction Selection (3)

- Control instructions that change the PSW masks/modes/spaces will introduce some forms of in-order operations within the pipeline and their usage should be limited
- EXTRACT and STORE FLOATING POINT CONTROLS instructions will also introduce some forms of in-order operations and should be avoided if possible
- Translation/Key changing/purging control instructions may involve some forms of serialization and should be limited
  - For SET STORAGE KEY EXTENDED, usage of the non-quiescing (NQ) variant is encouraged

Optimization - Instruction Scheduling (1)

- Optimizing instruction grouping efficiency might yield better performance
  - Arrange code such that 3 instructions that can be grouped together to optimize dispatch bandwidth
  - Instruction clump formation (instruction storage alignment) affects how instructions are fetched from the instruction cache, and may affect grouping effectiveness
  - Branch instruction ends a group in z196; but since zEC12, it ends only if it is predicted taken or if second in the group

- Execution results can be bypassed without any additional latency to a dependent instruction if the sourcing and receiving instructions are on the FXUs (FXUa, but not FXUb in z13/z14) of the same side of the issue queue
  - This bypass can be arranged by placing the related instructions consecutively, and thus usually in the same group (and the same side)

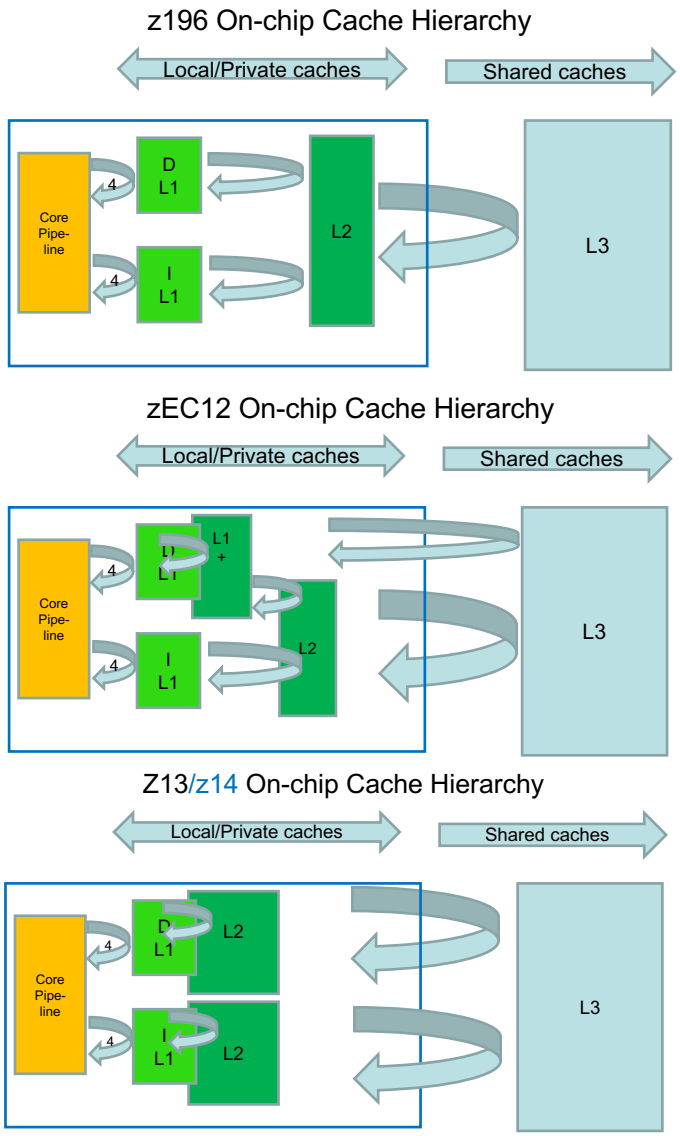
- Floating-point (FP) operations
  - Mixed mode FP (e.g. short->long, long->short, hex->bin, bin->hex) operations should be avoided; results are typically not bypassed, and can cost pipeline rejects or flushes
  - Since z13, the simpler mapper tracker design used for VRs (and FPRs) can lead to false dependencies in single precision FP operations; where possible, double precision FP operations should be used
  - Since z13, execution functions are evenly distributed (symmetric) among the 2 sides of the issue queue, scheduling that enables parallel processing among the 2 different sides can potentially achieve better performance
    - For reference, in z196 and zEC12, floating-point unit and fixed-point multiply engines are only provided on one side of the issue queue
    - Since z13, FP results bypassing capability are symmetric among FP operations from the two issue queue sides
    - The pipeline design can be very consistent in terms of instruction grouping when processing instructions in a loop, it is best to have long running or non-pipelined instructions, like floating-point SQUARE-ROOT/DIVIDE, distributed evenly between side0 and side1 of the dispatch groups by estimating the group boundaries such that they don't end up in the same side

Optimization - Instruction Scheduling (2)

- Software directives like branch prediction preload and prefetch data instructions are designed to help hardware optimize performance
  - Hardware is implemented to optimize performance (based on patterns) by predicting the intended program behavior. Hints about behavior from the programmer can help.
  - These directives potentially modify behavior of heuristic / history-based hardware mechanisms
    - The net performance improvement might vary between hardware implementations and the hints can potentially be ignored
  - Use responsibly
    - As usage might have adverse effects by increasing overall code size, these directives are best used by applying insights based on run-time profiles such that “blind” insertions can be avoided
    - Experimentation is highly advised
  - These directives should be placed as far back from actual branches or storage accesses as possible to be effective

Optimization - Local Caches

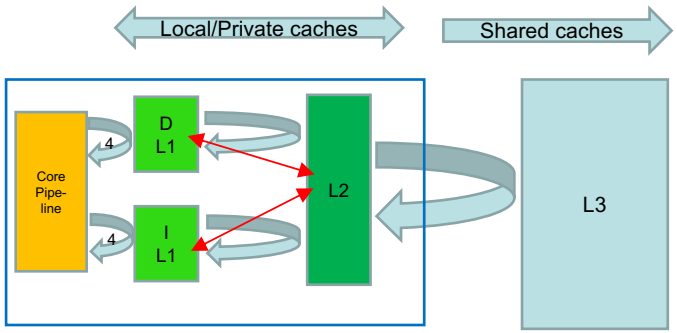
- Many traditional and current workloads running on IBM Z systems are data-centric, where a lot of time is spent in accessing storage and moving data around
  - Understanding the cache topology and optimizing data placement can be very important
- The picture on the right depicts the evolution of the per-core private L2 cache structure. Each generation improves the overall L2 cache size and latency (since z196):
  - By progressing from a unified (z196) to a hybrid (zEC12) to a fully split (z13/z14) design
  - The z13/z14's fully split L2s (vs. unified) for instructions (I-L2) and operands (D-L2) help keep data closer to the corresponding L1s
  - Contrast to the traditional serial lookup design, the integration of L2 directory lookup pipeline into L1 helps reduce access latency for L2 hit and L2 miss since the L2 status is now known much earlier
    - Integrated lookup approach is implemented in zEC12 for operands, then in z13/z14 for both operands and instructions
- The z processors' design allows large and fast L2 caches
  - L2 sizes are comparable to other competitors' L3s (MBs) by leveraging eDRAM technology in z13/z14
  - L2 access latency is also extremely aggressive, at around 10 cycles
- The cache-line size is 256 bytes throughout the cache hierarchy
  - For L1, L2, L3, and L4 (not shown here)
  - Safe value to use for separation / alignment



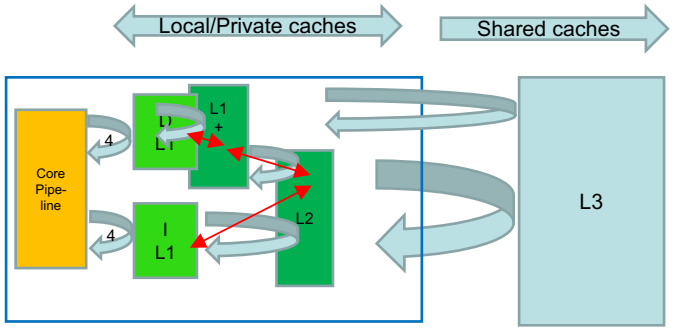
Optimization - Data Placement

- The on-chip L3 is shared by all cores on the CP chip
  - In z13/z14, it is also the common/sharing point for I-L2 and D-L2
- Split L1 caches design was (re-)introduced in z900 (2000)
  - Designs were optimized for well-behaved code
    - Increased cost of I and D cache contention if they are sharing contents in the same cache line in a conflicting way
  - With the split L2 cache design since z13, resolution of any conflict is moved to L3
- What happens when Instructions & Operands share same cache line in a split L2 design (since z13)?
  - OK (maybe inefficient) if operands usage is also read-only /shared since instructions are read-only by design
  - PROBLEM (performance wise) if stores are expected to those operand locations, including those stores on potentially predictive code paths
    - Extra cache misses will be encountered, driving long delays
- In general, the split L2 design is not a problem for
  - Re-entrant code or dynamic run-time code
  - Any LE-based compiler generated code
- However, there are problematic examples where conflicts can happen (and should be avoided):
  - True self-modifying code
  - Classic save area
  - Local save of return address
  - In-line macro parameters
  - Local working area right after code

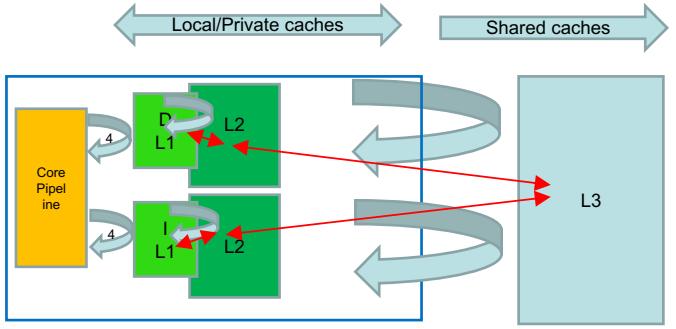
z196 On-chip Cache Hierarchy



zEC12 On-chip Cache Hierarchy



z13 On-chip Cache Hierarchy



Optimization - Instruction Cache

- As described in previous page, instructions (executable code) and operand data (working storage or stack storage) in the same cache lines, which can be costly due to moving cache lines between the separated (split) local caches (instruction/data L1/L2), should be avoided
  - Since both instruction and operand accesses can be predictive in nature, if the instruction and operand storage locations **can be located further apart**, the possibility of leading to unintended cache transfer delays can be reduced
  - The target operand of an EXECUTE-type instruction is treated as an instruction fetch (not data operand) and should be located as part of the instruction cache lines
  - Self-modifying code (or store-into-instruction-stream) is supported in hardware functionally, but in general, the sequence can become costly due to out-of-order pipelining and movement of cache lines
  - Pay attention to local (static) save areas and macro expansions with in-line storage parameters, especially in manually-assembled code, to avoid unintended sharing

- Instruction Cache optimization
  - Minimize the number of cache lines needed through the most frequent execution path
    - Separating out frequently and infrequently used code to different storage areas can improve both cache and translation-lookaside-buffer (TLB) efficiency
  - Software hints, e.g. prefetch data and branch prediction preload instructions, should not be added blindly
    - Unnecessary hints may increase instruction cache footprint and instruction processing delay
    - Branch prediction preload instruction also does instruction cache touch (as a way of prefetching)
  - Unrolling and in-lining should be done to improve potential processing parallelism, but should be targeted with a reasonable resulting loop size; i.e. aim for maximum processing with minimal loop size

Optimization – Shared Data

- Shared data structures among software threads / processes are common
  - Sharing is not necessarily bad and can be very useful to leverage the strongly consistent z/Architecture
- However, when updates are coming from multiple cores, the cache lines will bounce around among caches
  - Depending on locations of cores, added access latencies can be troublesome
- In the case of **false sharing**
  - Where independent structures / elements are in same cache line
  - Potential performance problems can be avoided by separating independent structures into different cache lines
- In the case of **true sharing**
  - Where real-time sharing of data structure is required among multiple software threads / processes
  - Often involved with the usage of atomic updates or software locks
  - When such operations are involved at a higher n-Way (concurrent software threads), the more frequent and parallel accesses that are performed at any given time, the more likely the cache lines involved are contested in real time
  - These cases can lead to “hot-cache-line” situations, where care and optimization as described here will be needed to help minimize cache coherency delays of cache lines movement latencies
  - A typical coding example was described on page 38 titled “Atomic and Locking Instructions”
- A recommendation is to consider splitting single locks into multiple locks when possible in highly-contested MP situations

Cache hit locations	Latencies (no queuing)	Intervention Overhead (if a core owes exclusive)
L1	4	NA
L2	~10	NA
L3 (on-chip)	35+ (z13) 45+ (z14)	40+
L3 (on-node)	200+ (z13) 220+ (z14)	20+ (z13) 35+ (z14)
L3 (off-drawer, Far column)	725+ (z13) 690+ (z14)	20+ (z13) 35+ (z14)

Cache best-case latencies (z13/z14)

Optimization – Data Cache (for operands)

- As explained in prior page, don't mix multiple distinct shared writeable data in the same cache line to avoid potential tug-of-war among multiple processors
  - Avoid putting multiple shared (and contested) locks in the same cache line
- Avoid using any storage element as a running variable that will get fetched and updated many times in close proximity
  - Consider using a general register (GR) instead
  - Similarly, avoid spill and fill through storage within a short number of instructions
- NIAI may be used to provide hints to the hardware about intentions of storage accesses to avoid delays from potential cache state changes
- Data Prefetch instructions with both prefetch and untouch functions are provided
  - For cache lines that are contested among many processors, it might not be desirable to prefetch the cache line ahead of time since it may add unnecessarily data movement in the system causing extra delays
- L1 cache access pipeline (from issue of data fetch to issue of dependent instruction) is currently 4 cycles
  - scheduling non-dependent operations in-between storage accesses and subsequent usages allows maximum parallel processing
- The hardware has mechanisms to detect store-to-load dependencies and to provide substantial bypass capabilities which improve with every generation, minimizing storage access dependencies can yield better performance
  - In general, simple store and load instructions are handled well while more complicated instructions or address overlaps may observe more pipeline rejects
  - In z13/z14's design, a generic temporary store data buffer is used to bypass pending store data to a dependent load. However, the following cases are not bypassable:
    - XC of > 8 bytes with perfect overlap (often used to clear storage with zeros)
    - MVC of > 16 bytes, even for the special case of 1-byte destructive overlap (often used to pad storage with a character)
    - Due to out-of-order handling of stores, multiple close-by store updates to the same doubleword location with dependent fetches in-between (sometimes observed due to loops)

Frequently Asked Questions (1)

- Question:
  - What should be used to move or clear large blocks of data?
- Answer:
- There are several ways to move or clear a large block of storage provided in the z/Architecture
  1. One MVCL instruction
  2. Loops of MVCs to move data
  3. Loops of MVC <Len>,<Addr>+1,<Addr> or XC <Len>,<Addr>,<Addr> to pad/clear an area
- As discussed on page 31 titled “MOVE LONG instructions”,
  - MVCL is implemented through millicode routines
  - Millicode is a firmware layer in the form of vertical microcode
    - Incurs some overhead in startup, boundary/exception checking, and ending
  - MVCL function implemented using loops of MVCs or XCs
  - Millicode has access to special near-memory engines that can do page-aligned move and page-aligned padding
    - Can be faster than dragging cache lines through the cache hierarchy
    - However, the destination data will NOT be in the local cache
- As such, the answer is “it depends” as there is no one answer to all situations. There are many factors to consider
  - Will the target be needed in local cache soon?
    - Then moving/padding “locally” will be better by using MVCs or XCs
  - Is the source in local cache?
    - Then moving/padding “locally” may be better by using MVCs, or XCs
  - How much data is being processed?
    - The more data you are required to process, the more you may benefit from using MVCL due to special hardware engines used by millicode
  - Experimentation is, therefore, highly advised

Frequently Asked Questions (2)

- Question:
  - The "Atomic and locking instructions" chart recommends the sequence:

```
LOOP   LT  R1, lock ; load from memory and test value; always test first
       BCR  14,0    ; serialization to guarantee getting new value
       JNZ LOOP     ; repeat if non-zero
```

  and the later "Serialization" chart shows a sequence

```
CPU 2
G   CLI  A, X'00'
    BNE  G
```

  with the comment: A serializing instruction must be in the CPU-2 loop to ensure that CPU 2 will again fetch and refresh data value from location A.
  We have code that does not perform the serializing instruction and it works just fine. Can you explain how this works, and whether the serialization instruction is really needed?

- Answer:
- From an architecture point of view, the serialization is needed to guarantee updated values will be used on both cases
- On the other hand, the serialization is not "needed" because all recent designs will always reacquire the cache line eventually. Such behavior is due to our hardware cache protocol and the presence of both architectural and micro-architectural interruptions. So, even without the serializing instruction, the two examples will not loop indefinitely
- However, future hardware implementations may not work the same way. Although the IBM designs ensure that an update to a location will eventually be seen by all processors without the proactive serialization, relying on the design behavior can be a potential performance issue depending on when the hardware refreshes itself
- On all recent processors, the serialization instruction is normally executed as a fast single-cycle instruction, so there is negligible performance impact to including it in the code. “Proactive” serialization should be added whenever necessary.

Frequently Asked Questions (3)

- Question:
  - I have stack pointer update code that I'm considering changing from L/AR/ST to LAA, but I see in POPs that LAA does a specific-operand-serialization function, presumably to support block-concurrent interlocked-update. However, since my stack cache lines are guaranteed to be unique per CP (unique 8K-byte blocks per CP with no overlap) I really don't need the interlocked-update. Will the LAA unnecessarily cause cache messaging or the cache line to be pushed out to memory and thus make the single instruction slower than the L/AR/ST trio?
- Answer:
- Among all recent implementations, the enforcement of serialization (or atomic update) is very minimal
  - The handling for special-operand-serialization is no different than normal operand access ordering/coherency and should not lead to any additional overhead that can be observable
- The only time serialization / atomic overhead can be observed is when the involved cache line has real-time contention among multiple processes (on multiple processors)
  - With high contention, the pipeline may run in a slow mode to adhere to the serialization/atomic requirement.
    - Since LAA is one storage-access instruction vs. L/AR/ST having 2 storage-access instructions, LAA will likely be faster
  - Without contention, LAA is expected to be a bit faster because it is 1 instruction (vs. 3) and the pipeline will handle it fairly seamlessly.
- Therefore, LAA is preferred to sequence of 3 instructions
- Note:
  - In general, the z processors are built to optimize single **“storage + arithmetic/logic”** operations through a “dual issue” design, such that instructions like “A”, “S”, “OI”, “NI” are handled very efficiently
    - These single instructions should be used over multiple simple instructions

Frequently Asked Questions (4)

- Question:
  - What instruction is best to use in setting registers to zeros?
- Answer:
- For most cases (in real life code), the differences between SR/XR vs. LHI or SGR/XGR vs. LGHI will not be noticeable
- Obtaining the ultimate performance depends on neighboring code, specifically on 1) code address alignment, 2) condition code (CC) dependencies and usage, and 3) register dependencies and usage
- To expand on the rule of thumb:
  1. If one doesn't want to think too hard, use L(G)HI. L(G)HI is generally good as one does not need to worry about register or condition code dependencies. L(G)HI doesn't write the CC, which is a potential plus in terms of processor resource usage. The only downside is its instruction length being a little longer. If someone would want to optimize further, X(G)R or S(G)R can be used instead to relief instruction fetching limitations which leads to #2
  2. If someone wants to think a little more, then the recommendation is to optimize for instruction path-length unless there is a dependency conflict. Thus,
     - If one is doing a 32/64-bit GR clear and doesn't care about the CC or would prefer the CC to be cleared, use X(G)R or S(G)R
     - If you're doing a 32/64-bit GR clear and cannot destroy the CC, use L(G)HI
     - If the GR to be cleared was used near-by (similar to CC concern), use L(G)HI to avoid register dependencies
  3. For extreme optimization, one will have to understand the processor pipeline a lot more in terms of instruction grouping, register and condition code buffer size, instruction alignment, etc.
     - All these considerations can get very complicated and convoluted. So, it is best to leave to the compiler
- Note that for writing zeros to FPRs, LZXR or LZDR can be used, while for VRs, VGBM w/ I2=0 can be used

Frequently Asked Questions (5)

- Question:
  - For operand data access, what hardware prefetching is available and how can you best use the PREFETCH DATA instruction as a form of software prefetching?
- Answer:
- Since z196, the z processor pipeline is out of order. As each generation improves upon the out of order window, the pipeline inherently overlaps storage accesses that are independent from each other; depending upon instruction fetching/dispatching, ”prefetching” is performed whenever out of order processing can ”get to it”
- Since z10, the z processor implements a hardware-based stride prefetch engine and its capability and accuracy has been improving on each subsequent system
  - It captures history of operand access addresses corresponding to a set of instruction address (PC) offsets
  - If it detects a striding pattern (+/- 1 to 2047) at a certain PC offset and determines the pattern has happened more than 2 times, it will start launching the next access in the form of a prefetch by predicting the next several striding accesses
  - For example, it will detect X, X+Y, X+2*Y at instruction address A (usually in a loop), and then launch hardware data prefetches at X+3*Y and X+4*Y; if it detects with more confidence, then it can prefetch up to X+5*Y (i.e. 3 strides ahead)
  - When the stride detected is small (less than a cache-line size), the HW prefetch engine can decide to be aggressive and prefetch the next cache line
- Since the HW prefetch engine is built to handle stride-based prefetching within loops, it is advisable not to insert additional line-based PFD instructions into the code since it might become redundant and potentially incur unnecessary overhead
- Software prefetching using line-based PFD instructions is best done far ahead of the actual load
  - An instruction distance of at least half of the out of order window, i.e. > 40 instructions for z13/z14, to see potential benefits

References

1. “z/Architecture: Principles of operation,” Int. Bus. Mach. (IBM) Corp., Armonk, NY, USA, Order No. SA22-7832-10, Feb. 2015. [Online].
2. M. Farrell et al, “Millicode in an IBM zSeries processor,” IBM J. Res. & Dev., vol. 48, no. 3/4, pp. 425–434, 2004.
3. C.F. Webb, ‘‘IBM z10: The Next-Generation Mainframe Microprocessor,’’ IEEE Micro, vol. 28, no. 2, 2008, pp. 19-29.
4. C Shum, “Design and microarchitecture of the IBM System z10 microprocessor,” IBM J. Res.& Dev., vol. 53, no. 1, 2009, pp. 1.1-1.12.
5. Brian W. Curran et al, “The zEnterprise 196 System and Microprocessor,” IEEE Micro, vol. 31, no. 2, 2011, pp. 26-40.
6. F. Busaba et al, “IBM zEnterprise 196 microprocessor and cache subsystem,” IBM J. Res.& Dev., vol. 56, no. 1/2, pp. 1:1–1:12, Jan./Feb. 2012.
7. K. Shum et al, “IBM zEC12: The third-generation high-frequency mainframe microprocessor,” IEEE Micro, vol. 33, no. 2, pp 38–47, Mar./Apr. 2013.
8. Bonanno et al, “Two Level Bulk Preload Branch Prediction”, HPCA, 2013
9. C. Jacobi et al, “Transactional Memory Architecture and Implementation for IBM System z,” IEEE/ACM Symposium on Microarchitecture (MICRO), 2012.
10. B. Curran et al, “The IBM z13 multithreaded microprocessor,” IBM J. Res. & Dev., vol. 59, no. 4/5, pp. 1:1–1:13, 2015.
11. E. M. Schwarz et al, “The SIMD accelerator for business analytics on the IBM z13,” IBM J. Res. & Dev., vol. 59, no. 4/5, pp. 2:1–2:16, 2015.
12. B. Prasky et al, “Software can Provide Information Directly to the System z Microprocessor”, IBM Systems Magazine, May 2014
13. C. Walters et al, "The IBM z13 processor cache subsystem", IBM J. Res. & Dev., vol. 50, no. 4/5, pp. 3:1-3:14, 2015

References (Cont'd)

14. C. Jacobi et al, "The IBM z14 microprocessor chip set", IEEE Hot Chips 29, 2017
15. John R. Ehrman's book on IBM Z system assembly programming:
    http://idcp.marist.edu/enterprisesystemseducation/assemblerlanguageresources-1.html
16. Dan Greiner's presentations of z/Architecture features at SHARE conferences:
    See share.org

THANK YOU

Suggestions, questions, comments:

cshum@us.ibm.com

https://www.linkedin.com/in/ckevinshum