IBM z13 Microprocessor and System

Tim Slegel
IBM Distinguished Engineer
IBM z Systems Processor Development
Poughkeepsie, NY

IBM z13 - Outline
• A Brief History of IBM Mainframes
• z/Architecture Instruction Set
• z13 Chip Technology
• Processor
  • Pipeline
    • Front end – I-fetch and branch prediction
    • Out-of-Order – register renaming
    • Execution units
    • Load/Store unit
    • CoP – Cryptographic Co-processor
  • SMT – Simultaneous Multi-Throwing
  • Transaction Memory (Execution)
  • Processor RAS and Sparing
• Caches and interconnects
• Memory
• System packaging and cooling
A Brief History of IBM Mainframe Systems

- IBM S/360 announced April 7, 1964
  - 50+ years of systems designed to run the same application software
- Evolved through S/370, S370-XA, S/390, ESA/390
- z/Architecture announced in 2000
  - 64-bit addressing and registers
- Technology
  - Before 1997, mainframe systems used bipolar technology => small scale integration and power-hungry
  - 1997 – transitioned to CMOS
  - Last couple generations have suffered from the end of Moore’s Law and decreasing rate of technology performance improvements
- Operating Systems:
  - z/OS – flagship OS used by numerous corporations worldwide
  - zLinux – Linux for the mainframe. Extremely popular with clients.
  - z/VM (and now KVM) – primarily second-level virtualization platform for zLinux
  - Other OSes: TPF, VSE, etc.

z13 Continues the CMOS Mainframe Heritage

- MIPS Tables are NOT adequate for making comparisons of z Systems processors. Additional capacity planning required
- ** Number of cores for customer use
**z/Architecture and Implementation**

- z/Architecture is a 64-bit architecture that is supported by IBM's z Systems microprocessors
- A Complex Instruction Set Computer (CISC) architecture, including highly capable (and thus complex) instructions
- 1200+ instructions in the Instruction Set Architecture (ISA)
- 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 (introduced on z13), including fixed-point, floating-point, and character string operations
- Atomic operations including COMPARE AND SWAP, LOAD AND ADD, and OR (immediate) instructions
- Big-Endian (BE) architecture – bytes are stored with the most significant byte (MSB) at the lower storage address
- Hardware transactional memory, through the Transactional Execution Facility, including the definition of a constrained transaction that can be retried by the hardware
- Two-way Simultaneously Multi-Threading (SMT-2) support introduced on z13
**System Hardware, Firmware, and Software**

- **z/Architecture Level (ISA)**
- **Micro-architecture Level**
  - Hardware
  - Millicode
  - i390 Code
  - LPAR Hypervisor (PR/SM)
  - OS
  - App. Software

---

**z/Architecture ISA and Implementation**

- Implementation is micro-architecture
- Around 1050 instructions implemented entirely in hardware
- Highly complex instructions are implemented through a special firmware layer – millicode
  - Millicode is a form of vertical microcode
  - An instruction implemented in millicode (a millicoded instruction) is executed by the hardware similar to a built-in subroutine call, that transparently returns back to the program when the millicode routine ends
  - Millicode instructions executes in the same hardware pipeline as normal instructions
  - A millicode instruction routine consists a subset of the existing instructions in the z/Architecture. In addition, about 100 special hardware instructions that only millicode can use.
  - Millicode has its own internal registers in addition to program registers
  - Some complex routines may involve operating in conjunction with a private coprocessor or special hardware that is only accessible by millicode
  - The routine is pre-optimized for each processor generation. Written in assembler.
z13 8-Core Central Processor Chip Technology

- Up to eight active cores (CPUs) per chip
  - 5.0 GHz (v5.5 GHz zEC12)
  - L1 cache/ core
    • 96 KB I-cache
    • 128 KB D-cache
  - L2 cache/ core
    • 2M+2M Byte eDRAM split private L2 cache
- On chip 64 MB eDRAM L3 Cache
  - Shared by all cores
- I/O buses
  - One InfiniBand I/O bus
  - Two PCIe I/O buses
- Memory Controller (MCU)
  - Interface to controller on memory DIMMs
  - Supports RAIM design

- 22nm SOI Technology
  - 3.99 Billion Transistors
  - 17 layers of metal
  - 13.7 miles of copper wire
  - >300W max chip power
- Chip Area
  - 678.8 mm²
  - 28.4 x 23.9 mm
  - 17,773 power pins
  - 1,603 signal I/Os

Core Floorplan

- IFB – Branch prediction
- ICM – L1 I-cache
- IDU – Inst. Dispatch
- ISU – Inst. Sequencer (O-O-O handling)
- FXU – Integer execution
- VFU – FP/vector execution
- LSU – L1 D-cache (load/store unit)
- XU – Virtual->Real translation & TLB2
- COP – Crypto & Data Compression
- RU – Recovery
- PC – Pervasive function
- L2I/L2D – L2 caches
High-Level View of the Microprocessor Core

- The z microprocessor cores can be simplified into a number of functional units
  - Branch prediction unit
    - 2 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 fixed-point divides
  - Vector & Floating-Point Unit
    - Handles floating-point arithmetic operations, complicated fixed-point operations, and vector operations
  - Load/Store (or Data-caching) unit
    - Accesses operand data for both fetch (load) or store (update) operations
  - Co-processor unit
    - Supports data compression, cryptographic functions, UTF translations; operates through millicode routine
  - Second Level Translation and Cache unit
    - Maintains the private second level translation-lookaside-buffer (TLB2) and cache (L2)
Branch Prediction Structures

Instruction fetch → Branch Prediction Logic → ACC → BTB1 and BHT1 → Speculative BHT & PHT → PHT → Branch Prediction Overflow Queue (BPOQ) → Instruction fetch

Branch Prediction Structures Sizes

<table>
<thead>
<tr>
<th>Label</th>
<th>Structure Name</th>
<th>Description</th>
<th>Z13</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTBP</td>
<td>Branch Target Pre-buffer</td>
<td>0.5th level branch instruction address and target predictor – look-up in parallel to BTB1, upon usage, transfer to BTB1</td>
<td>128 x 6</td>
</tr>
<tr>
<td>BTB1</td>
<td>L1 Branch Target Buffer</td>
<td>1st level branch inst. addr. and target predictor</td>
<td>1024 x 6</td>
</tr>
<tr>
<td>BHT1</td>
<td>L1 Branch History Table</td>
<td>1st level direction predictor (2-bit) : weakly, strongly taken, or not-taken</td>
<td>1024 x 6</td>
</tr>
<tr>
<td>BTB2</td>
<td>L2 Branch Target Buffer</td>
<td>2nd level branch instruction address and target history buffer</td>
<td>16384 x 6</td>
</tr>
<tr>
<td>ACC</td>
<td>Column Predictor</td>
<td>Accelerate BTB1 throughput in finding the &quot;next&quot; branch</td>
<td>1024</td>
</tr>
<tr>
<td>SBHT/</td>
<td>Speculative BHT &amp; PHT</td>
<td>Speculative direction prediction with updates at (O-O-O) resolution time prior to completion</td>
<td>8 + 8</td>
</tr>
<tr>
<td>PHT</td>
<td>Pattern History Table</td>
<td>Pattern based tagged direction prediction</td>
<td>1024 x 6</td>
</tr>
<tr>
<td>CTB</td>
<td>Changing Target Buffer</td>
<td>Pattern based target prediction predicts branches with multiple targets, typically subroutine returns and branch tables</td>
<td>2048</td>
</tr>
<tr>
<td>SMRU</td>
<td>Super MRU table</td>
<td>Protect certain branches from normal LRU out to make the BTBP more effective</td>
<td>128</td>
</tr>
</tbody>
</table>
Branch Prediction Unit

- Branch prediction is an important feature in any modern microprocessor design.
- 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 in predicting the program path much further into program code than where the instruction fetching unit is currently delivering instructions at (and should be way ahead of where the execution engines are executing).
- The branch prediction logic adapts many advanced algorithms/structures in maintaining and 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) with a pre-buffer (BTBP) used as a transient buffer to filter out unnecessary histories.
  - Accelerators for improving prediction throughput (ACC) by "predicting the prediction" so it can make a prediction every cycle (for a limited subset of branches).
  - Pattern based direction and target predictors (PHT and CTB) to predict based on "how the program gets here" branch history (that represents the program flow), e.g. for predicting an ending of a branch on count loop, or a subroutine return that has multiple callers.
- 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.

Instruction Delivery

- Since z/Architecture instructions are of variable lengths of 2, 4, and 6 bytes, an instruction can start at any halfword (integral 2-byte) granularity.
- Instruction fetching fetches "chunks" of storage aligned data from the instruction cache, starting at a disruption point; e.g. after a taken branch (including subroutine calls and returns), or a pipeline flush:
  - Up to 4 8-byte chunks for z13.
- These "chunks" of 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, notifies the branch prediction logic (to restart its search) and then proceeds based on predefined static prediction rules.
  - Is it going to be miccoded and if true, did the branch prediction logic predict that? If not, resets the front-end to start at the corresponding miccode routine entry instruction.
  - For a complex instruction, does it need to be "cracked" or "expanded" into simpler internal instructions, called micro-operations (µop’s)? For example, a LOAD MULTIPLE instruction will be expanded into multiple "load" µops that fetch from storage and write individual general purpose registers (GRs).
- Instructions (and µop’s) are then bundled to form an instruction group (for pipeline management efficiency), and dispatched (written) into the instruction issue queue.
Instruction Cracking or Expansion

- Always (due to inherent multiple operations needed), e.g.,
  - BRANCH ON COUNT (BCTR) \(\rightarrow\) add register with immediate value of -1
  - code
  - \(\rightarrow\) scratch condition
  - \(\rightarrow\) branch evaluation

- Length based (multiple operations based on length), e.g.,
  - 8 byte MOVE characters (MVC) \(\rightarrow\) load into scratch register
  - \(\rightarrow\) store from scratch register

  - 16 byte LOAD MULTIPLE (LM) \(\rightarrow\) load into register 1
  - \(\rightarrow\) load into register 2(addr, adjusted at dispatch)
  - \(\rightarrow\) load into register 3(addr, adjusted at dispatch)
  - \(\rightarrow\) load into register 4(addr, adjusted at dispatch)

- Although the processor pipeline may be “RISC-like”; typical register-storage instructions, e.g. “ADD” in example below, are handled efficiently using “dual issue”
  - ADD: Register1 <= Register1 + memory((Base register) + (Index register) + Displacement)
  - Register-storage ADD (A) \(\rightarrow\) load from storage into target register
  - \(\rightarrow\) add R1 with target register
  - The instruction is not considered as cracked because it is tracked as 1 instruction by using 1 issue queue entry (and 1 global completion table entry)

Instruction Grouping

- As instructions (and µop’s) are grouped, they are subject to various grouping rules, which prevent certain instructions from being grouped with others
- z13 allows two groups of up to 3 instructions at a time
- Once instructions are dispatched (or written) into the issue queue as a group, they are tracked in the global completion table (GCT) until every instruction in the group has finished processing; then the group is completed and retired
- Some basic rules of grouping
  - Simple instructions, including most “register-register” and “register-storage” type instructions, can be grouped together
  - Branch instructions, if second in the group, or if predicted taken, cannot be grouped with instructions after
    - Best group size if taken branches are the third in a group
  - µops expanded from the same instruction will usually be grouped together
  - If expanded into only 2 µops, can be grouped with one other simple instruction after
  - Storage-storage instructions are usually grouped alone, except for the µop’s that they may be expanded into
  - Other instructions that are alone in a group:
    - Register-pair writers, non-branch condition code readers, explicit floating-point control register readers or writers, instructions with multiple storage operands, EXECUTE-type
    - Max group size will be 2 if any µop has more than 3 register sources (including Access Register usage in AR mode)
z13 high-level instruction & execution flow

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 inflight mappings can be tracked) and physical registers available are key factors of the effectiveness of an out-of-order design.
  - In 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.
    - 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.
  - Two groups can be written at any cycle with one group into each side.

- The issue queue includes a dedicated “virtual branch queue”, 1 per side, that handles relative branch instructions whose targets are within 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

- Once instructions are dispatched into the issue queues, the issue queues will issue the oldest (and ready) instruction for each issue port to the corresponding execution engine.

- Each issue queue side is connected to a number of specific processing units:
  - There are 5 issue ports (per side; 10 total per core); each to a different unit, 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; it 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 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 (for up to 50 outstanding instructions):
    - **BFU**: handles both hexadecimal and binary (IEEE standard) floating-point arithmetic operations, and vector floating-point operations.
    - **DFU**: handles decimal (IEEE standard) floating-point arithmetic operations.
    - **SIMD**: further composes 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**: handles decimal (BCD) fixed-point arithmetic operations.
    - **FPD**: handles divide and square root operations for both binary and hexadecimal floating-point arithmetic.

### Out-of-order resources

<table>
<thead>
<tr>
<th></th>
<th>z13</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR</td>
<td>120 (up to 16 reserved for each thread while in millicode) + 8 immediate value entries</td>
</tr>
<tr>
<td>FPR / VR</td>
<td>127 FPRs / VRs (up to 8 reserved for each thread while in millicode) + a zero value entry</td>
</tr>
<tr>
<td>AR (access register)</td>
<td>96 (up to 8 reserved for each thread while in millicode)</td>
</tr>
<tr>
<td>Issue Queue</td>
<td>30 x 2 sides + 14 x 2 sides of Branch queue</td>
</tr>
<tr>
<td>Global Completion Table</td>
<td>24 x 2 x 3 instructions (complete up to 6 instructions / cycle)</td>
</tr>
<tr>
<td>Unified Mapping Trackers</td>
<td>64 + 64</td>
</tr>
</tbody>
</table>

- **Out-of-order issue window up to 88 instructions.**
- **Up to 144 instructions in-flight between dispatch and completion.**
z13 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

Load/store unit (LSU)

• 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 access of data elements of up to 8-bytes per cycle
  – There is no performance penalty on alignment except for when the element crosses a cache line
  – Vector elements of more than 8 bytes are accessed in two successive cycles
• Besides prefetching of cache misses by the natural behavior of the out-of-order pipeline
  – LSU supports software prefetching through PREFETCH DATA type instructions
  – LSU also includes a stride-prefetching hardware engine that prefetches +1, +2 stride
• To minimize pipeline bubbles typically caused by “store-load” dependencies through storage, LSU provides a sophisticated bypass network allowing pending storage updates that are not yet available in the L1 cache be bypassed into dependent fetches as if the data was in L1 (subject to certain limitations). But in general,
  – Data should be bypass-able by bytes from different storing instructions to a fetch return
  – Data should be bypass-able if the store data is ready a small number of cycles before the fetch request
  – Multiple mechanisms are used to predict dependencies (based on prior pipeline processing history) among fetch and store instructions, and will then stall fetch instructions just enough to enable “perfectly” timed bypasses.
  – If a store operation is performed after its dependent load (due to out-of-order operations), a flush will occur
  – If a store operation is performed before its dependent load, and data is not bypass-able (due to timing or hardware limitations), the load will be rejected and retried
On-chip Cryptographic Co-Processor

- On-chip core co-processors (CoPs) are available to enable hardware acceleration of data compression, cryptography, and Unicode conversions
  - Each COP is private to each core
- The co-processor handles instruction COMPRESSION CALL (CMPSC) that compresses data and cryptographic functions (under the CPACF facility, next page) that supports latest NIST standards
  - In addition, all Unicode conversions (UTF 8<>16<>32) are supported
- The following cryptographic functions are provided: AES (128, 192, 256), DES, T-DES, SHA1, SHA (256, 384, 512), pseudo-random number generation
- 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 operands into the co-processor to be processed
  - Millicode sets up result storage areas
  - Coprocessor works on the instruction with the provided source data and generates output data
    - In the case of CMPSC, the coprocessor will also fetch dictionary tables accordingly
  - Millicode writes into the result storage 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, the co-processor will only handle operations one thread at a time and the other thread will wait until the current thread finishes

Simultaneous Multi-Threading (SMT)

- z13 is the first IBM mainframe to support SMT
  - Explored it for several generations before actually implementing it on z13
  - Workload consistency is extremely important to mainframe customers and SMT makes that difficult
  - Currently, only available on specialty engines (e.g., running Linux) and is not available for z/OS
  - Surprisingly difficult to design hardware and achieve good performance
- 2 hardware threads per core
- Most large structures in the core (caches, TLBs, branch prediction, architected register files) are shared between the threads
- Some smaller structures are dedicated per thread
- Generally, instruction decode/dispatch alternates between the two threads.
- Instructions issue regardless of thread
- Typical SMT hardware throughput improvement versus single-threaded, per core:
  - Average 20-25%
  - Some workloads achieve much better, some much worse.
Vector Facility - Single-Instruction-Multiple-Data (SIMD)

- SIMD, sometimes also referred to as vector, instructions are introduced in z13.
- To support these instructions, new vector registers 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>
- Each SIMD instruction provides fixed-sized vectors ranging one to sixteen elements:
  - Some instructions only operate 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.

Types of SIMD instructions on z13

<table>
<thead>
<tr>
<th>Integer</th>
<th>String</th>
<th>Floating-point</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 x 8b, 8 x 16b, 4 x 32b, 2 x 64b, 1 x 128b</td>
<td>Find 8b, 16b, 32b, equal or not equal with zero character end</td>
<td>Binary Floating-Point operations w/ double precision only</td>
</tr>
<tr>
<td>8b to 128b add, subtract</td>
<td>Range compare</td>
<td>2 BFUs with an effective increase in architected registers</td>
</tr>
<tr>
<td>128b add/subtract with carry</td>
<td>Find any equal</td>
<td>All IEEE trapping exceptions reported through VXC; and will not trigger interrupts</td>
</tr>
<tr>
<td>8b to 64b minimum, maximum, average, absolute, compare</td>
<td>Isolate String</td>
<td></td>
</tr>
<tr>
<td>8b to 16b multiply, multiply/add</td>
<td>Load to block boundary - load/store with length (to avoid access exceptions)</td>
<td></td>
</tr>
<tr>
<td>4 - 32 x 32 multiply/adds</td>
<td>Logical operations, shifts</td>
<td></td>
</tr>
<tr>
<td>Logical operations, shifts</td>
<td>Carryless multiply (8b to 64b), Checksum (32b)</td>
<td></td>
</tr>
<tr>
<td>Memory accesses efficient with 8B alignment; minor penalties for byte alignment</td>
<td>Gather / Scatter by Step, Permute, Replicate</td>
<td></td>
</tr>
<tr>
<td>Gather / Scatter by Step; Permute; Replicate</td>
<td>Pack/Unpack</td>
<td></td>
</tr>
<tr>
<td>Pack/Unpack</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hardware Transactional Memory

- Since 2012, z/Architecture supports hardware transactional (memory) execution through the Transaction execution (TX) facility
  - 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
  - Optional detail debug data can be provided

- Improves performance of software with lock elision, lock-free data structures, and speculative code optimization

- Access (fetch) footprint is limited by L2 associativity and size: 2MB

- Update (store) footprint is limited by L2 associativity and size of an internal store transaction buffer (store-cache)
  - That can contain up to 64 blocks of 128-byte (storage aligned) data changed within a transaction
  - The L1 data cache is updated upon store instruction processing within a transaction, but L2 is deferred until transaction completes

- "Constrained transactions" do not require a software abort handler
  - Not provided in any other architecture than z/Architecture; programmers love it
  - CPU ensures the transaction will eventually complete (if software constraints are followed)
  - Complex millicode uses various escalation techniques on successive aborts to ensure completion
  - Worst case, millicode stops all processors and I/O in the system to allow this core to complete the transaction

Extensive use of hardware speculation

- z/Architecture ISA places many strict constraints on how the CPU has to appear to be behave
  - Example – Strict memory ordering rules
  - Good for software – significantly easier and more robust MP programming than other ISAs
  - Bad for the CPU design team – difficult to achieve good performance

- CPU has to make use of speculative processing techniques
  - Assume things will go well, and have mechanisms to detect and back-off if they do not
  - In computer engineering, "It's OK to cheat as long as you don't get caught."
  - Under the covers, the CPU violates the memory ordering rules in the Architecture, but has extensive/complex logic to detect if software might observe it violating those rules. If it detects possible observation, it needs to redo the operation precisely following Architecture rules.
  - Result is software only can observe the CPU following all rules
Hardware/Millicode Support for Virtualization

- Full logical virtualization via the START INTERPRETIVE EXECUTION (SIE) instruction

- Nested SIE supports two level guests:
  - LPAR Hypervisor (firmware) runs natively. First-level guests are normal OSes (e.g., z/OS, zLinux, z/VM). Up to 85 1st level partitions.
  - First-level guest partitions can be dedicated to physical cores, or more often, share physical cores.
  - If z/VM is running as a first level guest, then it supports hundreds (or thousands) of second level guests (e.g., zLinux)

- Separate hardware Host/Guest-1/Guest-2 facilities:
  - z/Architecture control registers
  - Timing Facility (including interrupt controls)

- All important SIE State Description controls are buffered into hardware control registers during SIE-entry/exit, which is performed by millicode

- Hardware detects most SIE Intercept and Intervention conditions

- Full hardware support for SIE address translation:
  - RRF supports zone relocation (and zone based I/O interrupts)
  - Multi-level pageable guest support (up to 56 table fetches required for a single 2nd level guest ART/DAT translation)
  - MCDS handling of ARs
  - TLB2 holds multiple SIE guest entries simultaneously
  - Appropriate TLB purging on all CPUs for IPTE/IDTE operations with filtering

Processor Reliability, Availability, Serviceability (RAS)

- Error detection – Goal is 100% error detection
  - All latches protected with parity, local duplication, illegal state detection, etc.
  - Register Files and Arrays protected with parity or ECC
  - Micro-architected state protected with ECC or duplication of architected registers

- Processor Error Recovery
  - ECC correction on arrays
  - Processor instruction retry: All arrays and non-architected state is reset, instruction execution resumes at last hardware checkpoint

- Transparent Processor Sparing – If recovery was unsuccessful or after recovery threshold has been reached
  - Several spare cores in the system available for taking over from a failed core
  - Millicode on a functional core transfers most of the micro-architected state to the spare core
  - Spare core executes a Load R-unit State (LRUS) instruction which transfers the last few pieces of micro-architected state that must be done atomically
  - Spare core resumes execution of the software program, or if it was previously with a millicode routine
  - Application software and even the OS do not even know this has occurred
### z13 Drawer Cache Hierarchy Detail

**Single CPC Drawer View – 2 Nodes**

<table>
<thead>
<tr>
<th>Node 1</th>
<th>Node 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>Memory</td>
</tr>
<tr>
<td>L4 Cache</td>
<td>L4 Cache</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>L3 Cache</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>L2 Cache</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>L1 Cache</td>
</tr>
</tbody>
</table>

**Node 1 - Caches**
- L1 private 96k I (6-way), 128k D (8-way)
- L2 private 2 MB i + 2 MB d (8-way)
- L3 shared 64 MB / chip (16-way)
- L4 shared 480 MB / node (30-way)
  - plus 224 MB NIC (14-way)

**Node 1 - Caches**
- L1 private 96k I (6-way), 128k D (8-way)
- L2 private 2 MB i + 2 MB d (8-way)
- L3 shared 64 MB / chip (16-way)
- L4 shared 480 MB / node (30-way)
  - plus 224 MB NIC (14-way)

### z13 System Controller (SC) Chip Detail

- **CMOS 22nm SOI Technology**
  - 15 Layers of metal
  - 7.1 Billion transistors
  - 12.4 Miles of copper wire

- **Chip Area**
  - 28.4 x 23.9 mm
  - 678 mm²
  - 11,950 power pins
  - 1,707 Signal Connectors

- **eDRAM Shared L4 Cache**
  - 480 MB per SC chip
  - 224 MB L3 NIC (non-inclusive)
  - Directory
  - 2 SCs = 960 MB L4 per z13 drawer

- **Interconnects (L4 – L4)**
  - 3 to CPs in node
  - 1 to SC (node – node) in drawer
  - 3 to SC nodes in remote drawers

- **6 Clock domains**
System Cache Structure

- A z system consists of multiple computing nodes, connected through the global fabric interface, each system node includes a number of 3 processor (CP) chips
  - 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 L3 caches
- Each processor (CP) chip includes a number of processor cores
  - There are 8 in z13
  - Each core includes both local L1 instruction and operand data caches, and a local L2 cache
  - A pair of L2 caches supports instruction and operand data separately
  - Each L2 cache is connected to the on-chip (shared) L3
- 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 (modified MESI):
    - "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); and
    - "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 so this processor can be the exclusive owner
  - Similarly, an exclusive line will need to be invalidated before another processor can read or write to it

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 once the instruction performing the update has been processed
  - 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 being 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
- The z/Architecture and the processor design supports self-modifying code
  - However, this can be a costly event due to movement of cache lines between the instruction and data caches (L1 and L2)
  - 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 has to 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 to not share writeable data elements in the same cache line for independent multiprocessor operations
- The associativity of a cache reflects how many available compartments a particular cache line can be stored in
  - For a 8-way associative cache, a cache line (based on its line address) can be saved in one of 8 slots
z13 Memory and RAIM

z13 Memory
- Up to 10 TB total system memory
- 42 GB/s between MCU and L3 cache
- RAIM – Redundant Array of Independent Memory

Layers of Memory Recovery:
ECC
- Powerful 90b/64b Reed Solomon code
DRAM Failure
- Marking technology; no half sparing needed
- 2 DRAM can be marked
- Call for replacement on third DRAM

Lane Failure
- CRC with Retry
- Data – lane sparing
- CLK – RAIM with lane sparing

DIMM Failure (discrete components, VTT Reg.)
- CRC with Retry
- Data – lane sparing
- CLK – RAIM with lane sparing

DIMM Controller ASIC Failure
- RAIM Recovery

Channel Failure
- RAIM Recovery

Physical node: (Two per drawer)
- Chips
  - Three CP chips
  - One SC chip (480 MB L4 cache)
- RAIM Memory
  - Three Memory Controllers: One per CP Chip
  - Five DDR3 DIMM slots per Controller: 15 total per logical node
  - Populated DIMM slots: 20 or 25 per drawer
- SC and CP Chip Interconnects
  - X-bus: SC and CPs to each other (same node)
  - S-bus: SC to SC chip in the same drawer
  - A-bus: SC to SC chips in the remote drawers

Fully Populated Drawer

4 Drawer System Interconnect
z13 CP and SC SCM assembly

- Capped CP
- 6x CP SCMs
- CP Chip
- 2x SC SCMs (Air Cooled)
- 6x CP SCMs under the cold-plates
- Capped SC
- SC Chip

Fully assembled Drawer with the chilled water supply manifold lifted to the left

IBM z13 Cut-away Drawer

- Chilled water plumbing For CP chips
- Air-cooled SC chip
- Processor chip sockets
z13 Water Cooled Model – Front View

- Internal Batteries (optional)
- Power Supplies
- Displays and keyboards for Support Elements
- PCIe I/O drawers numbers 1 to 4
- 2 x 1U Support Elements
- PCIe I/O drawer number 5
- System Control Hubs (used to be BPHs)
- CPC Drawers, PCIe Fanouts, Cooling water manifold and pipes, PCIe I/O interconnect cables, FSPs and ethernet cables
- N+1 Water Cooling Units

* Overhead Power and I/O options not shown. Same as for the Air Cooled System

z13 Radiator-based Air cooled – Front View

- Overhead Power Cables (option)
- Internal Batteries (optional)
- Power Supplies
- Displays and keyboards for Support Elements
- PCIe I/O drawers numbers 1 to 4
- 2 x 1U Support Elements
- PCIe I/O drawer number 5
- System Control Hubs (used to be BPHs)
- CPC Drawers, PCIe Fanouts, Cooling water manifold and pipes, PCIe I/O interconnect cables, FSPs and ethernet cables
- N+2 Radiator Pumps

Overhead I/O feature is a co-req for overhead power option
IBM z13 System

Thank You!

Email: slegel@us.ibm.com

Other Links:
IBM Journal of Research and Development issue on the IBM z13:
http://ieeexplore.ieee.org/xpl/tocresult.jsp?isnumber=7175088
IBM Careers:
http://ibmcampusus.com
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.

These 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:


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, Celeron, Intel Atom, Intel SpeedStep, Intel, 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, other countries, or both.

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 from 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.

All information contained or described in this document or other countries, and the information may be subject to change without notice. Contact 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 from 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.