Supporting the new IBM z13 mainframe and its SIMD vector unit

Dr. Ulrich Weigand
Senior Technical Staff Member
GNU/Linux Compilers & Toolchain

Date: Apr 13, 2015

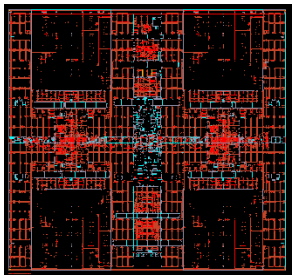
Agenda

- **IBM z13**
- **Vector ABI considerations**
- **Vector language extension**
- **Implementation status**

IBM z13

z Systems processor roadmap

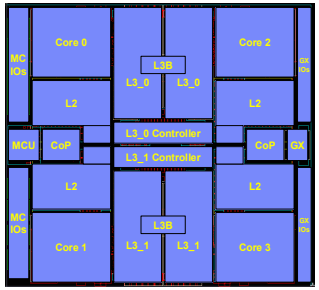
z10 2/2008



Workload Consolidation and Integration Engine for CPU Intensive Workloads

- Decimal FP
- Infiniband
- 64-CP Image
- Large Pages
- Shared Memory

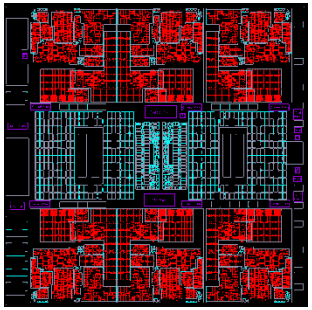
z196 9/2010



Top Tier Single Thread Performance,System Capacity

- Accelerator Integration
- Out of Order Execution
- Water Cooling
- PCIe I/O Fabric
- RAIM
- Enhanced Energy Management

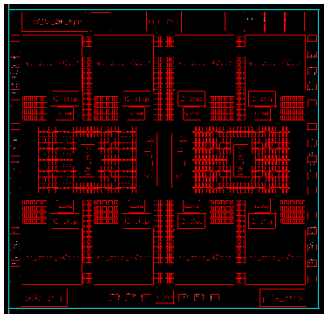
zEC12 8/2012



Leadership Single Thread, Enhanced Throughput

- Improved out-of-order
- Transactional Memory
- Dynamic Optimization
- 2 GB page support
- Step Function in System Capacity

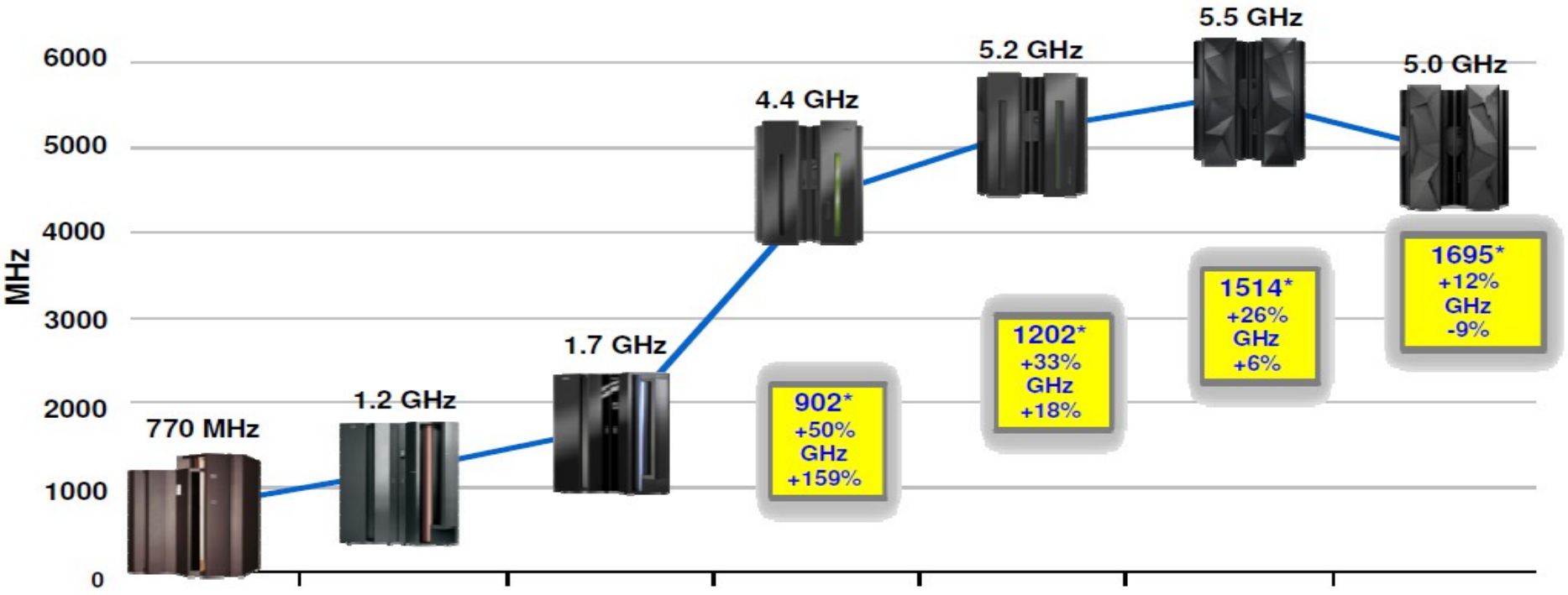
z13 1/2015



Leadership System Capacity and Performance

- Modularity & Scalability
- Dynamic SMT
- Supports two instruction threads
- SIMD
- PCIe attached accelerators (XML)
- Business Analytics Optimized

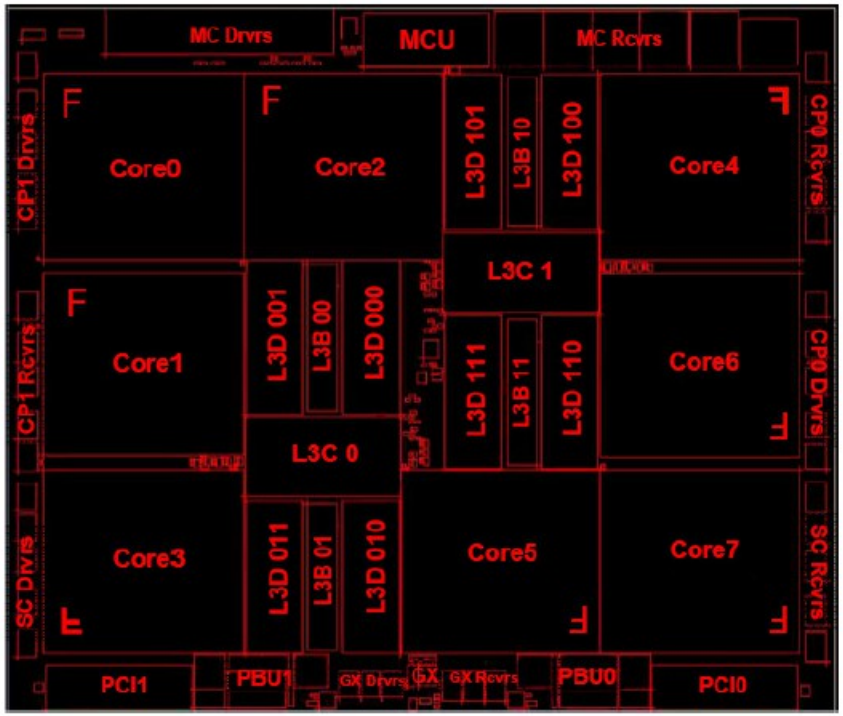
z13 continues the CMOS mainframe heritage



* MIPS Tables are NOT adequate for making comparisons of z Systems processors. Additional capacity planning required

** Number of PU cores for customer use

z13 8-core processor chip



- **14S0 22nm SOI Technology**
  - 17 layers of metal
  - 3.99 Billion Transistors
  - 13.7 miles of copper wire
- **Chip Area**
  - 678.8 mm^2
  - 28.4 x 23.9 mm
  - 17,773 power pins
  - 1,603 signal I/Os

- **Up to eight active cores (PUs) 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
- **Single Instruction/Multiple Data (SIMD)**
- **Single thread or 2-way simultaneous multithreaded (SMT) operation**
- **Improved instruction execution bandwidth:**
  - Greatly improved branch prediction and instruction fetch to support SMT
  - Instruction decode, dispatch, complete increased to 6 instructions per cycle*
  - Issue up to 10 instructions per cycle*
  - Integer and floating point execution units
- **On chip 64 MB eDRAM L3 Cache**
  - Shared by all cores
- **I/O buses**
  - One GX++ I/O bus
  - Two PCIe I/O buses
- **Memory Controller (MCU)**
  - Interface to controller on memory DIMMs
  - Supports RAIM design

* zEC12 decodes 3 instructions and executes 7

z13 SIMD – Business analytics vector processing

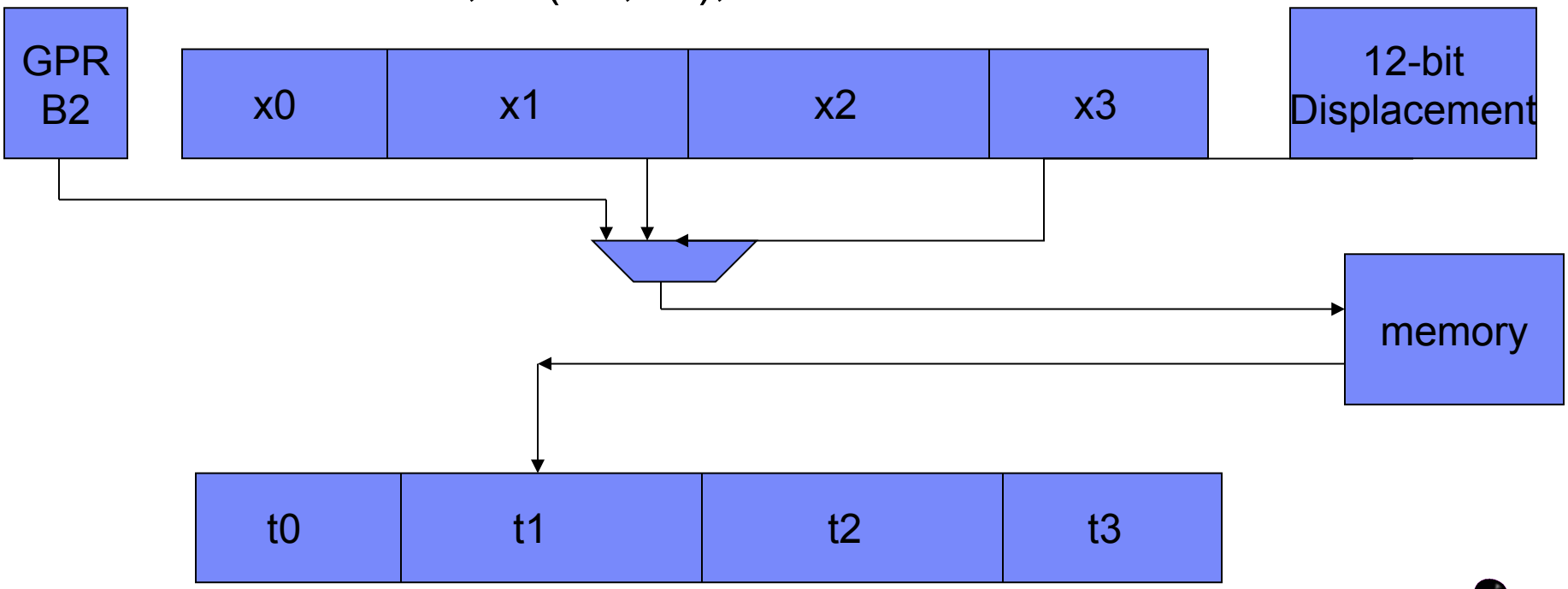
- **Single Instruction Multiple Data instruction set**
  - Support
    - Vector load/store, pack/unpack, merge, permute, select
    - Vector gather/scatter element
    - Vector load/store with length; load to block boundary
  - Integer
    - 8b...128b add/subtract (with/without carry/borrow)
    - 8b...64b min, max, average, complement/neg/pos
    - 8b...64b vector compare; single element compare
    - 8b...32b multiply, multiply/add [low/high/even/odd]
    - Full-vector bitops & shifts, 8b..64b element shifts/rotates
    - Sum-across, population count, checksum
    - Galois field multiply sum / and accumulate

z13 SIMD – Business analytics vector processing

- **Single Instruction Multiple Data instruction set**
  - Floating-point
    - DP add, sub, mul, div, sqrt, multiply-and-add/sub
    - Conversions (integer vs. DP, SP vs. DP)
    - Compare & test data class
    - Scalar forms of all instructions (single-element DP)
    - Full IEEE support (rounding modes, exceptions)
  - String
    - Supported character types: 8b, 16b, 32b
    - Vector Find Any Element [Not] Equal [Or Zero]
    - Vector Find Element [Not] Equal [Or Zero]
    - Vector Isolate String
    - Vector String Range Compare

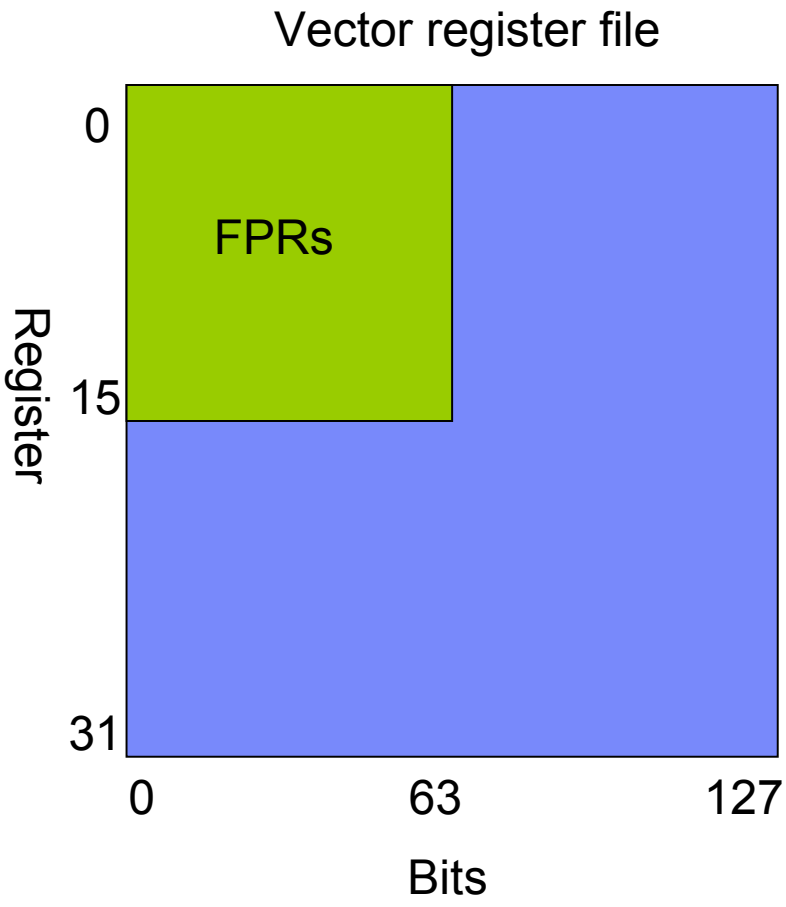
z13 SIMD – Business analytics vector processing

- **Example: Vector gather / scatter element**
  - VGEF V1,D2(V2,B2),M3
  - VSCEF V1,D2(V2,B2),M3



Overlaid vector / floating point register file

- **Overlaid register file**
  - Bits 0:63 of SIMD registers 0-15 will correspond to FPRs 0-15
  - When writing to an FPR, bits 64:127 of the corresponding vector register will become unpredictable
- **SIMD width 128 bits**
  - 1x128b, 2x64b, 4x32b, 8x16b, 16x8b integer
  - 2x64b, 1x64b floating-point



Vector ABI considerations

Vector registers

- **Kernel support**
  - Save/restore VRs on context switch
    - "Lazy allocation": first vector instruction traps to kernel
    - *Note: visible to user space via data-exception code*
  - Save/restore VRs across signal handler invocation
    - Compatible handler stack layout, extended at end
  - Debugger access (ptrace/core file) to VR register set
    - NT_S390_VXRS_LOW: low 8 bytes of VRs 0-15
    - NT_S390_VXRS_HIGH: full VRs 16-31
  - Kernel indicates support via "vx" feature bit
    - Reported via /proc/cpuinfo "features" string
    - Also indicates hardware support
    - *Note: **Only** checking machine type **not** sufficient!*

Vector registers (cont.)

- **Function calling convention**
  - All VRs are defined as call-clobbered
  - No extension of user-space context data structures
    - jmp_buf (setjmp/longjmp), struct ucontext_t (*context)
  - *Not optimal, but only option that does not break ABI*
- **Why no call-saved VRs?**
  - Would require extending jmp_buf, struct ucontext_t
  - ABI change can be *mostly* hidden via version flags and symbol-versioning of glibc routines (setjmp etc.)
  - Still breaks user code that embeds jmp_buf into struct
    - Broke critical applications (e.g. Perl modules, libpng)

Vector data types

- **Already exist with current compilers!**
  - GCC extension: attribute((vector_size(...))
  - Passed via reference, operations fully scalarized
  - *Note: ABI of using those types **does** change!*
- **New function calling convention**
  - Pass in up to 8 VRs (VR 24–31)
  - Excess arguments passed on stack (not by reference)
    - One or two DW slots, short vectors aligned to the left
  - Unnamed arguments to variable argument routines always passed on the stack
    - Leaves va_list data type compatible between ABIs
  - No vector arguments to unprototyped routines!

Vector data types (cont.)

- **Alignment of vector data types**
  - Current ABI: always naturally aligned
    - Default GCC rule was automatically applied ...
  - Vector ABI: maximum alignment of 8 bytes
    - Vector load/store already efficient with 8 byte alignment
    - ABI only guarantees 8 byte stack pointer alignment
  - *Note: Alignment change applies both at the C source level **and** at the LLVM IR level (DataLayout string)*
- **ABI selection**
  - Vector ABI tied to vector facility (-mvx/-mno-vx)
  - Vector facility/ABI default when using -march=z13
  - Object files marked via .gnu_attribute tags

Vector language extension

Compatibility goals

- **IBM XL C/C++ for z/OS**
  - Defines vector extensions for z13
  - Similar to Linux variant, not 100% identical
- **AltiVec/VSX vector language extensions**
  - Vector data types (“vector” keyword)
  - Vector builtins defined in <altivec.h> header file
  - C operators defined on vector types (later addition)
- **GCC vector extension**
  - Data types defined via attribute((vector_size(...)))
  - C operators defined on vector types

System z vector extension: types

- **Closely modeled after AltiVec/VSX**
  - Context-sensitive “vector” keyword
  - Integer: vector [un]signed (char|short|int|long long)
    - *Note: “vector long” is not allowed!*
  - Boolean: vector bool (char|short|int|long long)
  - Floating-point: vector double
    - *Note: “vector float” not supported at this time*
  - No equivalent to AltiVec “vector pixel”
- **“Syntactic sugar” only**
  - Data types defined via “vector” keyword behave identical to equivalent “attribute((vector_size))” types
    - Exception: vector bool

System z vector extension: operators

- **Vector integer / floating-point types**
  - Operators follow GCC vector extension
    - Vector types are identical to underlying GCC types!
  - Challenge: relational/comparison operators
    - GCC extension: returns vector signed integer type
      - Marked as “opaque” to allow implicit conversion
    - Cell/B.E. AltiVec extension: returns scalar bool (“all”)
    - XL z/OS extension: returns vector bool type
- **Vector bool types**
  - Do not exist in GCC vector extension
    - Mapped to “opaque” vector unsigned integer types
  - Implicit conversion to signed/unsigned types

System z vector extension: builtins

- **Header file <vecintrin.h>**
  - Builtins modeled after <altivec.h> builtins
    - Builtins overloaded by data type, even in C
  - Adapted to cover all System z vector instructions
  - No builtins for operations implemented by operators
    - Work around via e.g. #define vec_add(x, y) ((x) + (y))
- **Low-level builtins – not formally documented**
  - Used to implement <vecintrin.h>
  - LLVM implementation (mostly) compatible with GCC
    - Named __builtin_s390_vll, __builtin_s390_vstl, ...
    - Intended to be a 1:1 match to vector instructions
    - Map to LLVM IR target intrinsics (mostly)

Implementation status

Linux kernel and GNU toolchain

- **Kernel support**
  - Upstream since 3.19 (some fixes will be in 4.0)
- **Binutils support**
  - Vector instructions upstream (will be in 2.26)
  - Vector ABI tags still missing
- **GCC support**
  - Internal patch set available, not yet public
- **glibc support**
  - Optimized memory/string routines, not yet public
- **GDB support**
  - Register support upstream, ABI support t.b.d.

LLVM changes – to be posted

- **Core infrastructure**
  - Support z13 processor and vector facility
  - Vector register set as superset of FP register set
  - Native processor & feature detection
- **MC support**
  - All vector core, integer, floating-point, string instructions
  - Vector ABI tags still missing
  - Assembler support (e.g. vector gather address format)

LLVM changes – to be posted (cont.)

- **Code generation support**
  - Implement vector ABI if vector facility is present
    - DataLayout changes for 8-byte vector type alignment
    - Calling convention to use vector registers
      - Detect “unnamed arguments” – no generic feature?
  - Core instructions
    - Support general load/store/move/replicate
    - Exploit permute/select/merge/pack/unpack
    - Attempt to exploit vector gather/scatter element
  - Integer instructions
    - Usual arithmetic & bitwise operations
    - Comparisons (exploit condition code if feasible)

LLVM changes – to be posted (cont.)

- **Code generation support (cont.)**
  - Floating-point instructions
    - Full arithmetic on <2 x double>
    - Partial support for <4 x float>
      - Expand/scalarize non-supported operations
    - Exploit instructions for scalar “double” in 32 VRs
  - Short vector types
    - Accept <16-byte vector types, extend to full size
    - Optimize pack/unpack – useful for llvmpipe
  - New LLVM IR target intrinsics
    - Directly model all z13 instructions (that are not already directly modeled via standard LLVM IR)
    - Optimize CC result comparison

Clang changes – to be posted

- **Core infrastructure**
  - Support z13 processor and vector facility
    - Support -march=z13 option
    - New -mvx / -mno-vx command line options
  - Implement vector ABI
    - Vector type alignment
      - C/C++ language via MaxVectorAlign setting
      - LLVM DataLayout change
    - Calling convention
      - All vector types passed “direct” at the LLVM IR level
      - Handle “vector-like” single-element aggregates
      - Expand va_arg for vector types

Clang changes – to be posted (cont.)

- **Language extension**
  - Enabled via new option -mzvector / -mno-zvector
    - New internal flag getLangOpts().ZVector
    - Largely shares implementation with -maltivec code
  - Changes vs. AltiVec – data types
    - No “vector pixel”, “vector float”, “vector long”
    - Always support “vector long long”, “vector double”
  - Changes vs. AltiVec – operators
    - Some differences w.r.t. which implicit conversions are allowed (signed vs. unsigned vs. bool)
      - Mostly no-op in -flax-vector-conversions mode
      - Maybe incorrect for AltiVec too – to be verified …
    - Comparison operators handled like for GCC types

Clang changes – to be posted (cont.)

- **Low-level builtins**
  - Mostly straightforward via LLVM IR (GCCBuiltin)
    - Except for those that have additional CC return value
  - Some require compile-time literal argument verification
- **New header file <vecintrin.h>**
  - Implements documented System z vector builtins
  - Builtins implemented as always-inline function, or macros (where required due to constant arguments)
    - Plain C code, using vector operators or low-level builtins
  - Overloaded via clang attribute((overloaded))
  - Argument verification using attribute((enable_if))

Summary

Summary

- **New z13 mainframe first to support SIMD**
  - Intended to optimize business analytics workloads
- **System-wide changes required to exploit SIMD**
  - New ABI for vector registers and vector types
  - Source-language vector extensions
- **Implementation status**
  - Kernel support available
  - Core GNU toolchain support in progress
    - Waiting for GCC mainline to re-open after GCC 5.1
  - Clang/LLVM implementation in progress
    - To be submitted in parallel with GCC changes

Questions