Supporting the new IBM z13 mainframe and its SIMD vector unit

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

Date: Aug 7, 2015

Agenda

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

Contributors

Andreas Krebbel

Andreas Arnez

Stefan Liebler

Dominik Vogt

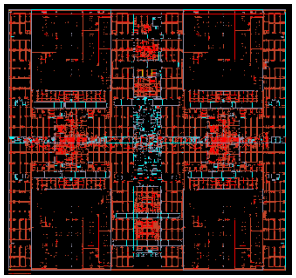
Richard Sandiford

Ulrich Weigand

IBM z13

z Systems processor roadmap

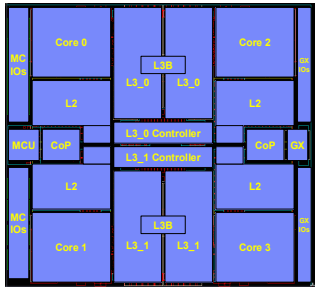
z10 2/2008



Workload Consolidation and Integration Engine for CPU Intensive Workloads

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

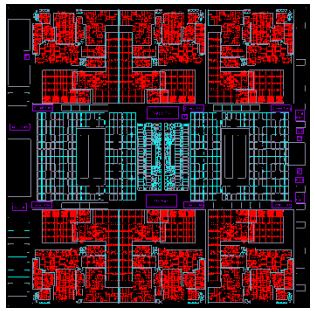
z196 9/2010



Top Tier Single Thread Performance,System Capacity

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

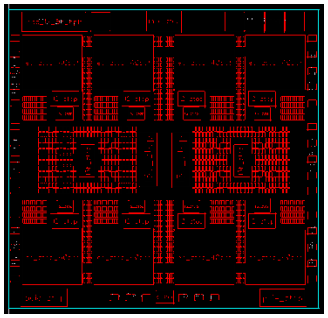
zEC12 8/2012



Leadership Single Thread, Enhanced Throughput

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

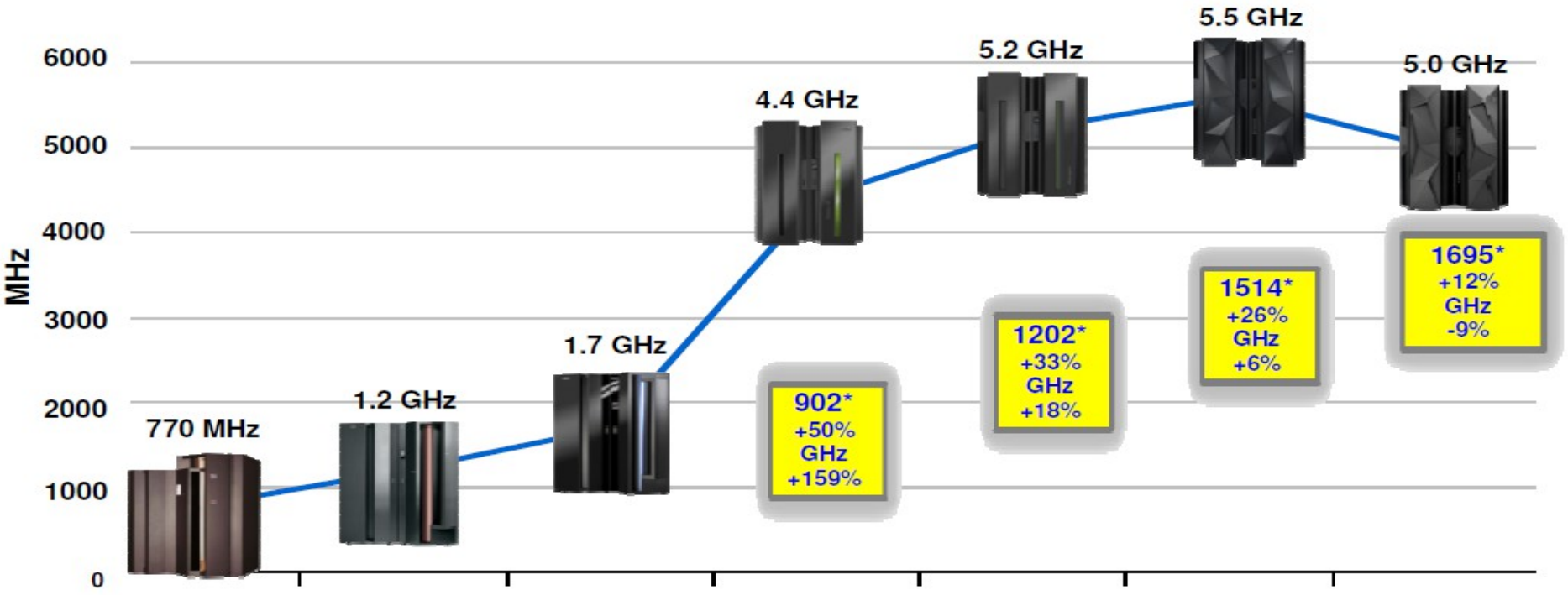
z13 1/2015



Leadership System Capacity and Performance

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

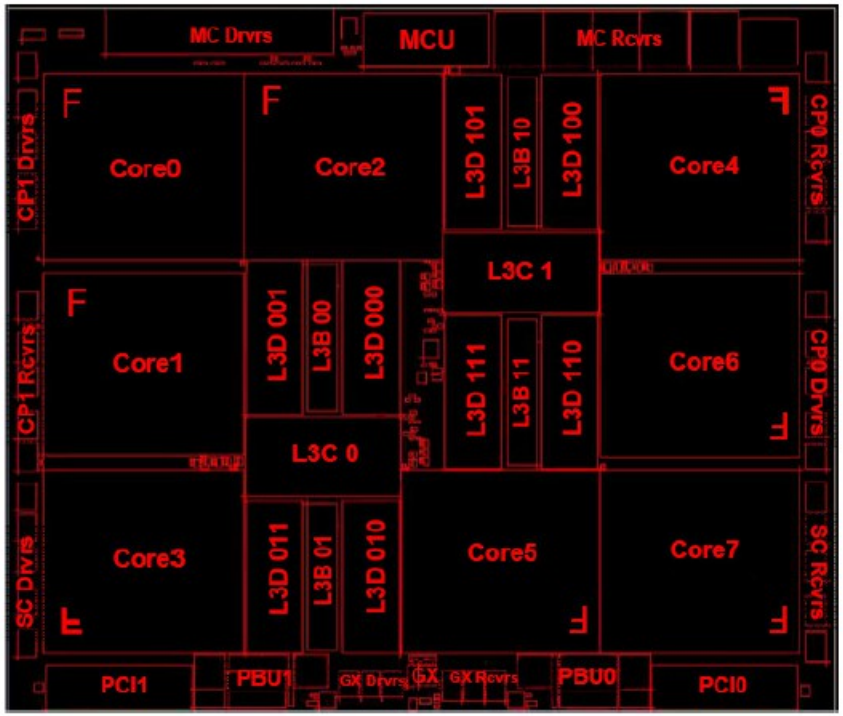
z13 continues the CMOS mainframe heritage



* 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

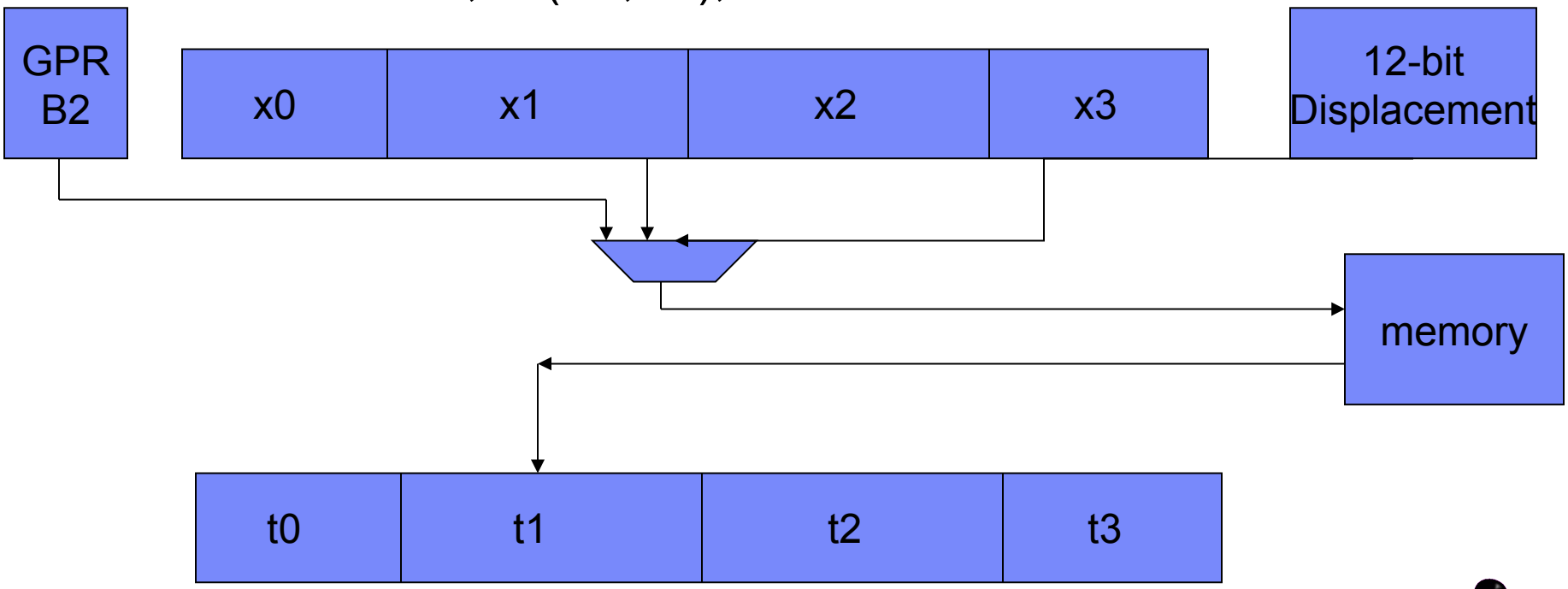
- **Single Instruction Multiple Data instruction set**
  - Support
    - Vector load/store, pack/unpack, merge, permute, select
    - Vector gather/scatter element
    - Vector load/store with length; load to block boundary
  - Integer
    - 8b...128b add/subtract (with/without carry/borrow)
    - 8b...64b 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

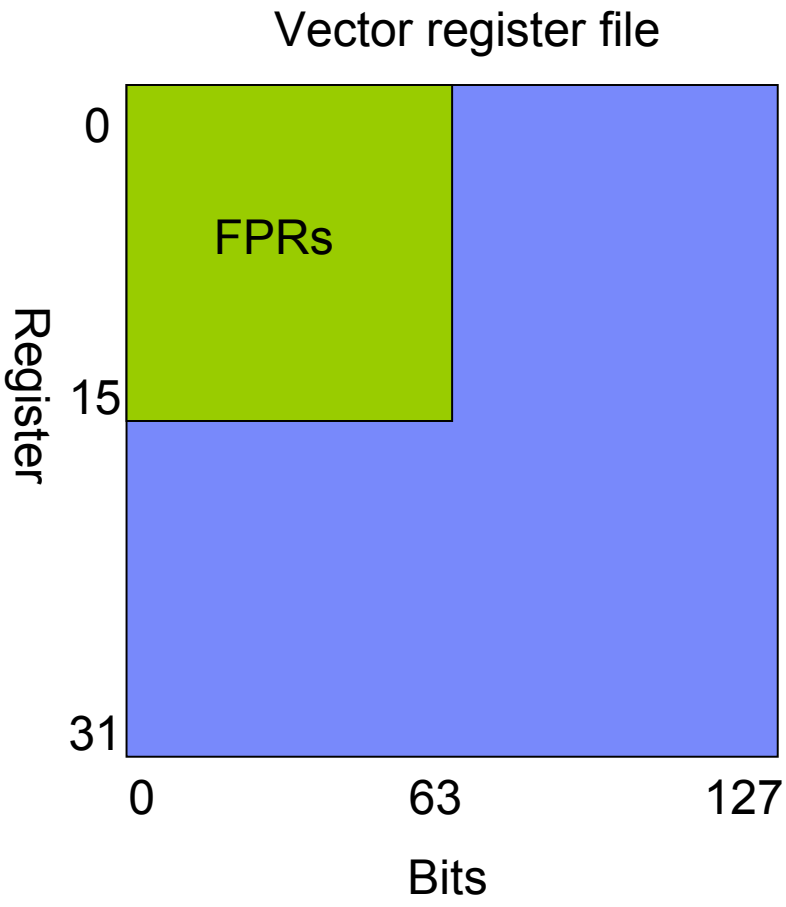
z13 SIMD – Business analytics vector processing

- **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
- **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 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>
  - Named __builtin_s390_vll, __builtin_s390_vstl, ...
  - Intended to be a 1:1 match to vector instructions

System z vector extension: example

```
#include <vecintrin.h>

vector signed int absdiff (
  vector signed int x,
  vector signed int y) {

  vector bool int cond = x > y;
  vector signed int vt = x - y;
  vector signed int vf = y - x;

  return vec_sel(vt, vf, cond);
}
```

```
vsf     %v2,%v26,%v24
vsf     %v0,%v24,%v26
vchf    %v24,%v24,%v26
vsel    %v24,%v2,%v0,%v24
br      %r14
```

```
gcc -march=z13 -mzvector -O
```


Implementation status

Implementation status – overview

- **Linux kernel support**
  - Upstream since 3.19 (some fixes in 4.0)
- **GNU Toolchain**
  - Binutils support upstream (will be in 2.26)
  - GCC support upstream (will be in 6, backported to 5.2)
  - Glibc: Optimized memory/string routines posted
  - GDB: Register & ABI support upstream (will be in 7.10)
- **LLVM & clang**
  - Full support upstream (will be in 3.7)
  - Intended to be feature-compatible with GCC

Automatic vectorization – loop example

```
#define N 1600

int a[N], b[N], c[N];

int test (void)
{
  int i;

  for (i = 0; i < N; i++)
    {
      int t = b[i] + c[i];
      a[i] = (t >= 0 ? t : 0);
    }

[...]
```

```
        larl    %r3,a
        larl    %r5,b
        larl    %r4,c
        vzero   %v2
        lghi    %r1,0
        lghi    %r2,400

.Lloop:
        vl      %v0,0(%r1,%r5)
        vl      %v4,0(%r1,%r4)
        vaf     %v0,%v0,%v4
        vmxf    %v0,%v0,%v2
        vst     %v0,0(%r1,%r3)
        aghi    %r1,16
        brctg   %r2,.Lloop
```

```
gcc -march=z13 -O3
```


Automatic vectorization – basic block example

```
unsigned int out[N], in[N];

int test (unsigned int x,
          unsigned int y)
{
  unsigned int a0, a1, a2, a3;

  a0 = in[0] + 23;
  a1 = in[1] + 142;
  a2 = in[2] + 2;
  a3 = in[3] + 31;

  out[0] = a0 * x;
  out[1] = a1 * y;
  out[2] = a2 * x;
  out[3] = a3 * y;
[…]
```

```
larl    %r1,in
vl      %v4,0(%r1)

larl    %r13,.Lconstant
vl      %v6,0(%r13)
vaf     %v2,%v4,%v6

vzero   %v0
vlvgf   %v0,%r2,0      # x
vlvgf   %v0,%r3,1      # y
vlvgf   %v0,%r2,2      # x
vlvgf   %v0,%r3,3      # y
vmlf    %v0,%v0,%v2

larl    %r12,out
vst     %v0,0(%r12)
```

```
gcc -march=z13 -O3
```


Implementation status

- **Future work**
  - Compiler performance enhancements
    - Micro-architecture tuning (e.g. scheduling)
    - Improved auto-vectorization
  - Additional SIMD-enabled libraries
    - E.g. libc math routines
  - Improved debugging tools
    - E.g. valgrind support for vectorized code
  - Resolve open issues

Open issues: Vector binary operators

- **Argument types for binary operators like “x + y”**
  - If x and y are the same vector type, no problem
  - If x and y are signed/unsigned variants?
    - Should be rejected according to GCC docs, OpenCL, AltiVec, and System z vector extension specifications
    - But is accepted by GCC and G++
    - But GCC and G++ differ in the result type of “x + y” !
  - How to handle “vector bool”?
    - AltiVec / System z allow “vector int + vector bool”
    - But the middle-end is not aware of “vector bool”, bool is just another variant of unsigned int
    - Rejecting “signed + unsigned” in the middle-end would therefore reject “signed + bool” too

Open issues: “opaque” vector type

- **What is an “opaque” type**
  - Originally used to implement SPE “__ev64_opaque__”
    - Allows implicit conversion to other vector types
    - Does not allow vector initializer
  - Now also used by the middle-end as result type of vector relational operators (<, >, ==, !=, …)
- **Should “vector bool” be an opaque type?**
  - Might solve the implicit conversion issue
  - Does not work as-is due to initializer problem
- **Should relational operators return “vector bool”?**
  - Would require target-specific decision

Open issues: ABI selection

- **Determine proper setting for ABI .gnu_attribute**
  - Check for use of vector type as external function argument or return type?
    - Misses struct type layout changes due to differing vector alignment
  - Check for any use of vector type in the source file?
    - Will set vector ABI marker for code that only uses vectors locally, guarded by a run-time check
  - Current GCC implementation
    - Attempts to check for use of any type affected by ABI in external interface (global variable, external function)
    - But misses some rare cases
      - Use of ABI-affected sizeof (…) as array size

Open issues: ABI selection (cont.)

- **Vector ABI vs. target attributes**
  - Vector ABI tied to -mvx/-mno-vx (i.e. -march=z13)
  - What about functions built with target attribute?
- **Option A: Target attribute affects ABI**
  - Potential incompatibility is user's problem?
    - Attempt to detect and report as error?
  - What about globally defined types or variables?
    - Also affected by vector ABI
    - Can those be used inside a target attribute function?
- **Option B: Uncouple ISA from ABI**
  - Yet more options … confusing the user?

Questions