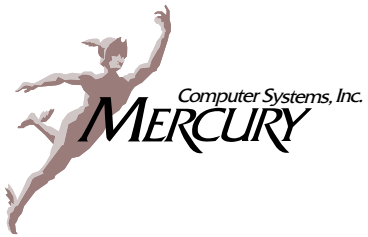
AltiVec Introduction

November 1998, Revision 6.0, No NDA Required



Craig Lund
Consultant, Local Knowledge
Principal Technologist, Mercury Computer Systems
3 Langley Road
Durham, NH 03824-3424
Tel: +1 603 868 2300
Fax: +1 603 868 2301
clund@localk.com

Legal

This presentation was created by Local Knowledge for Mercury Computer Systems leveraging materials owned by Mercury. Mercury hereby grants permission to both Motorola and Local Knowledge to freely create and distribute derivative works, provided Mercury receives prominent credit for its contribution.

Use this information at your own risk. We took care preparing this material. However, we suspect mistakes still exist.

Mr. Barry Isenstein
VP, Advanced Technology
Mercury Computer Systems, Inc.

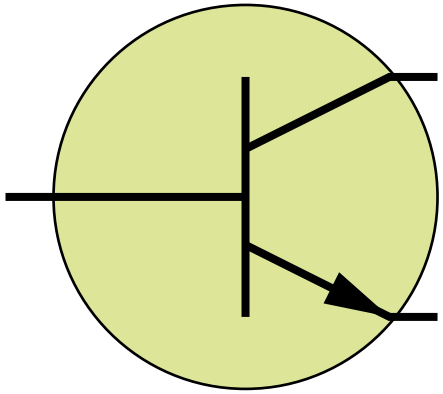
Presentation Summary

- AltiVec introduction
- AltiVec programming
  - Documentation & tools
  - AltiVec example code
  - G4 performance tuning

AltiVec Summary

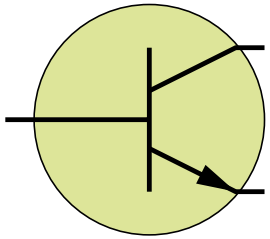
- Convergence of µP and DSP technology
- AltiVec summary
- Market position

What will we do with all the transistors?

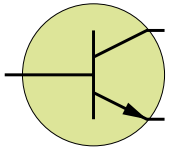


Logic transistors / cm^2
The National Technology Roadmap
for Semiconductors
Technology Needs 1997 Edition
Semiconductor Industry Association

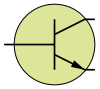
'97
3.7M



'99
6.2M



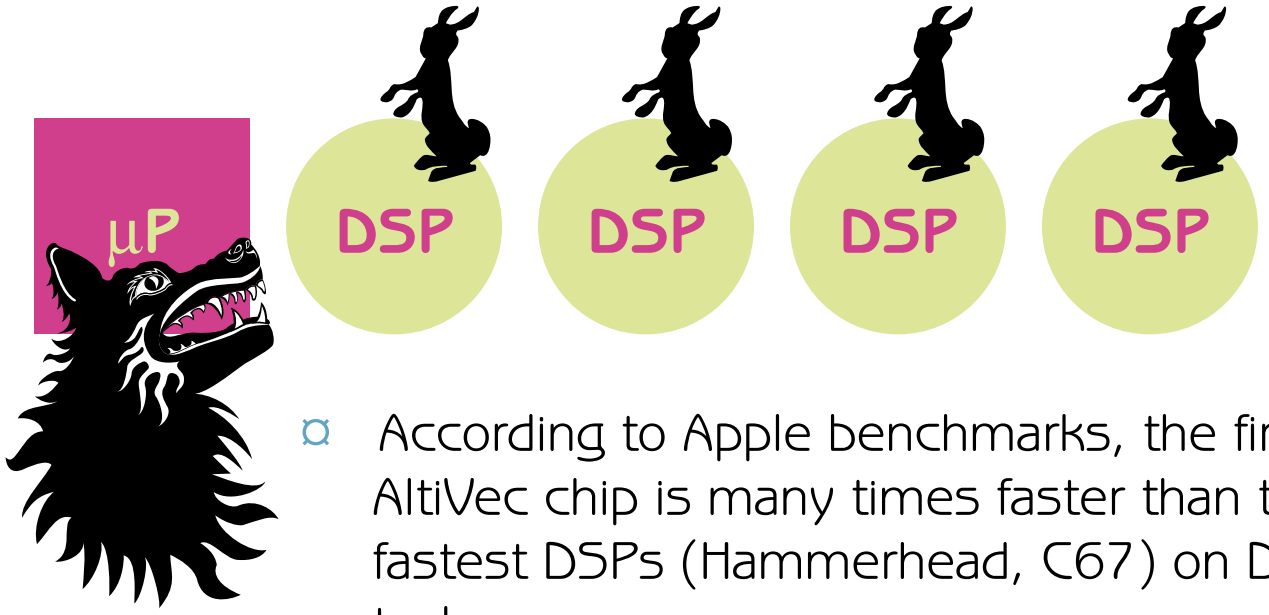
'01
10M



'03
18M

DSP & µP Convergence

- DSP chips are getting easier to use as they incorporate more complete functionality (byte operations, etc.).
- Microprocessors are gaining special-purpose functional units that resemble traditional DSP.

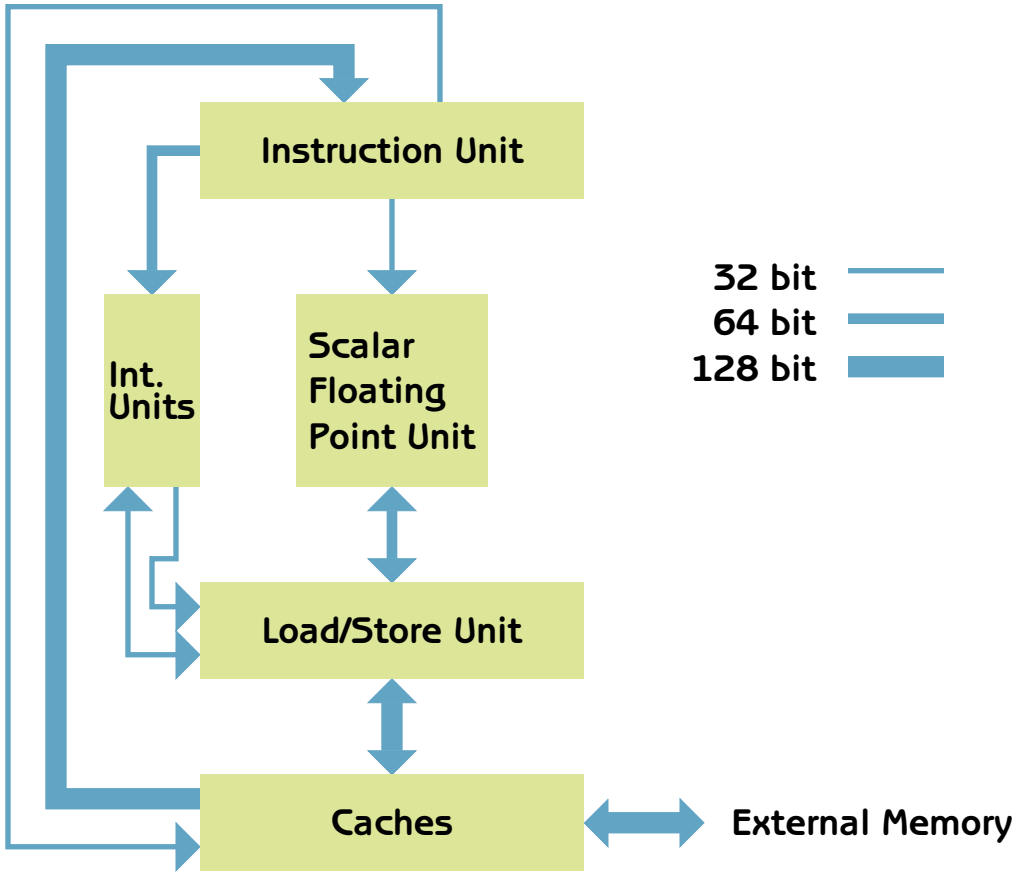


- According to Apple benchmarks, the first AltiVec chip is many times faster than the fastest DSPs (Hammerhead, C67) on DSP tasks.

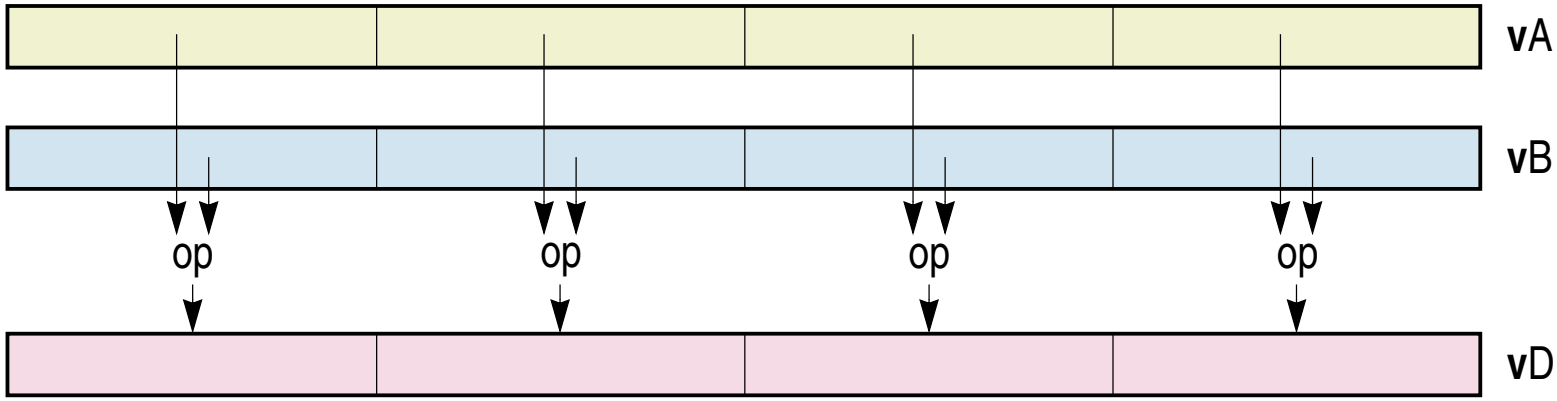
altivec technology

PowerPC Before AltiVec

- Look at all the transistors dedicated to wide data paths and to wide memories. Many sit idle when programs manipulate shorter data types.



AltiVec is SIMD



- Basic concept: always utilize the full bandwidth of data paths both on and off chip.
- Downside: standard compilers cannot automatically generate SIMD instruction sequences.

SIMD Instructions Everywhere

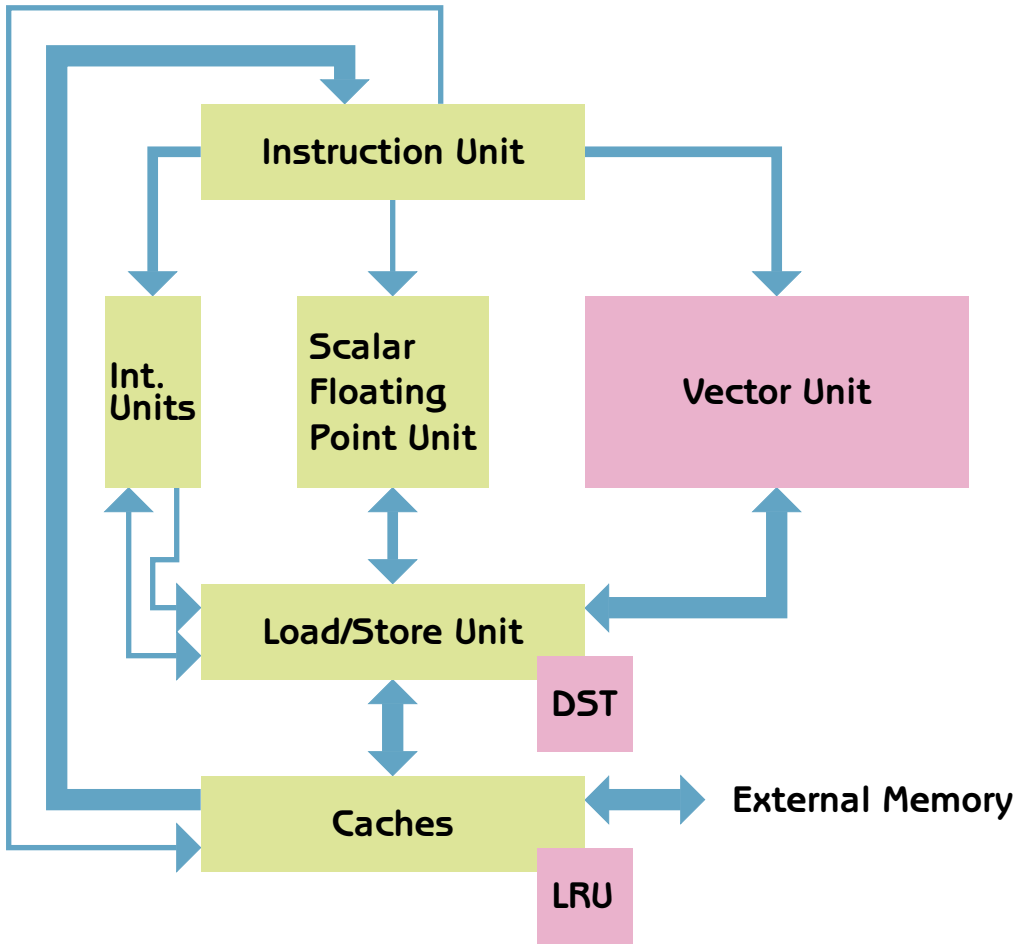
- HP's MAX2, 1994—pack two shorts into 32-bit registers and pipeline.
- Sun VIS, 1994—remap FP registers for arrays of bytes and shorts. Instructions for MPEG compression.
- Intel MMX, 1996—remap FP registers for arrays of bytes, shorts, and words. Two operand encoding.
- Digital's motion-video instructions (MVI), 1997 — five new instructions for MPEG motion estimation.
- 3DNow!, 1998—Intel MMX plus packed singles.
- MIPS V—pack two single floats into double registers and pipeline. MIPS MDMX (MadMax)—remap FP registers for arrays of bytes and shorts. Unique 192 bit accumulator for multiply.
- Intel Streaming SIMD, 1999—eight new 128-bit FP registers with 70 new instructions & "streaming data".

AltiVec History

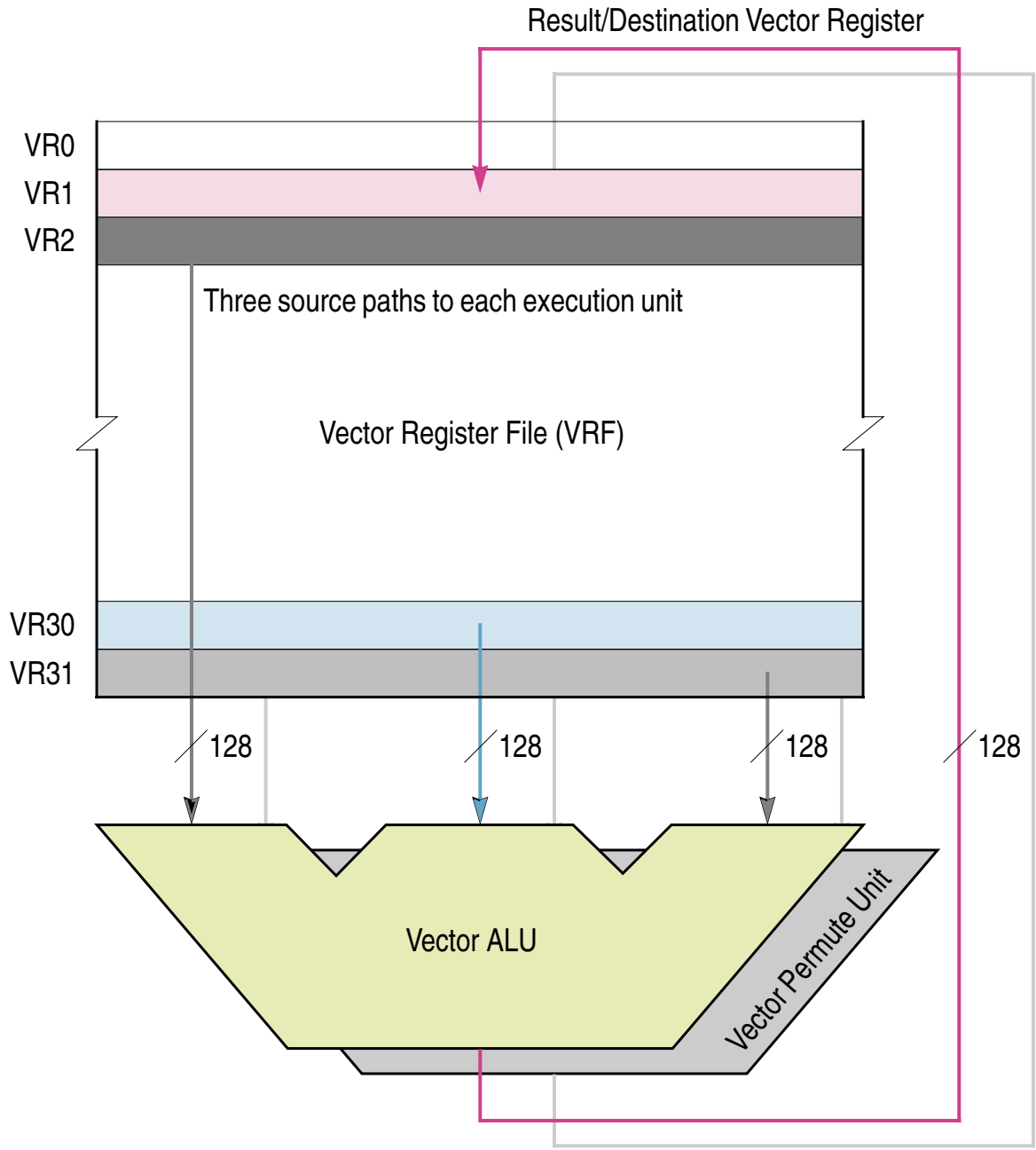
- In 1996 process technology was posed to dramatically shrink the PowerPC core, leaving room for a significant SIMD vector unit. Keith Diefendorff of Motorola, and later of Apple, drove the partnership to create a SIMD extension for PowerPC. As AltiVec's chief architect, Keith helped define a general purpose unit, not just the multimedia extensions other architectures have settled for. Keith is now editor of *Microprocessor Report.*
- IBM's POWER2 chips have two scalar 64-bit FP units. In contrast, the PowerPC has one scalar FP unit and AltiVec (more efficient hardware, less standard software). Different markets, different decisions.
- VeComp is not part of this history. It was a project by a Motorola research group.

Motorola's AltiVec

- Just as important as the AltiVec SIMD vector unit are the four prefetch engines (DST) and new cache behaviors (LRU).



Vector Unit

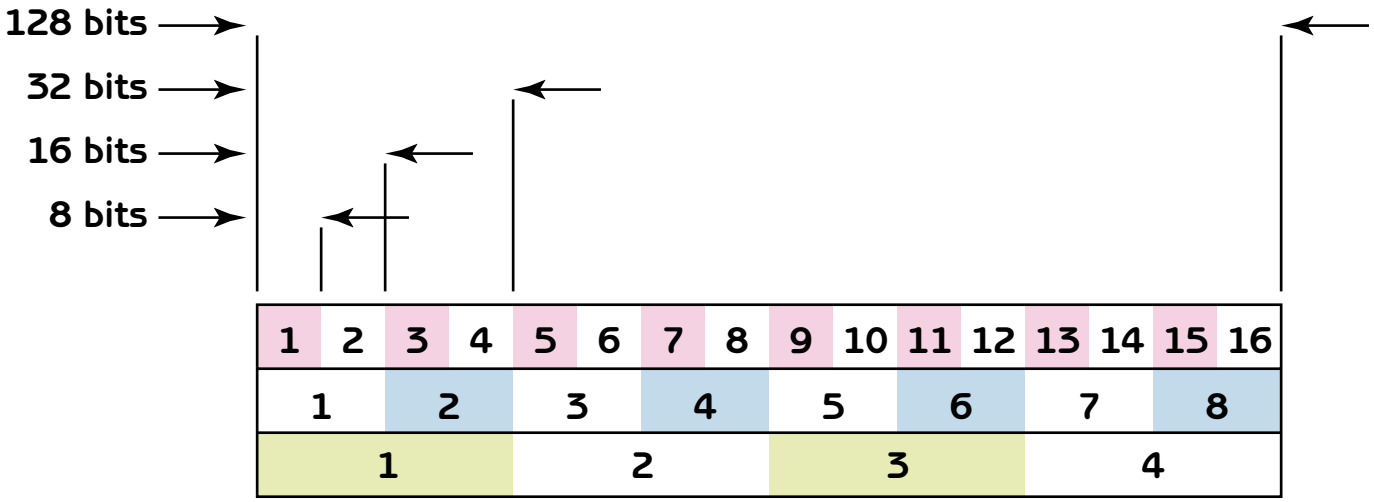


Load/Store paths
not shown

AltiVec Data Types

vector { unsigned / signed / bool } { char / short / long }

vector float
vector pixel



Three alternative vector register data layouts

AltiVec Operations

- 162 new PowerPC instructions—functionality similar to what is offered in the scalar units, just extrapolated into the SIMD domain.
  - Includes: new instructions for field permutation and formatting.
  - Includes: load/store instruction options for cache management.
  - Includes: instructions that control four data prefetch engines.
- The AltiVec vector unit never generates an exception. Default floating point results are reasonable and match the Java standard.

Java Mode

- AltiVec processors always expect floats in IEEE single precision floating point format.
- In Java mode AltiVec will perform gradual underflow (process denormalized results correctly). The cost in G4 is one extra cycle of latency for AltiVec floating point operations. Also, in some cases, a denormalized result will cause the processor to trap. With Java mode on, G4 complies with the Java subset of the IEEE floating point standard. Apple says to keep it on always. Mercury says to keep it off. Different markets, different advice.
- Also note that strict Java compliance requires that programmers not use the multiply-add fuse instruction (Java requires a product and sum to round separately).

Programming Options

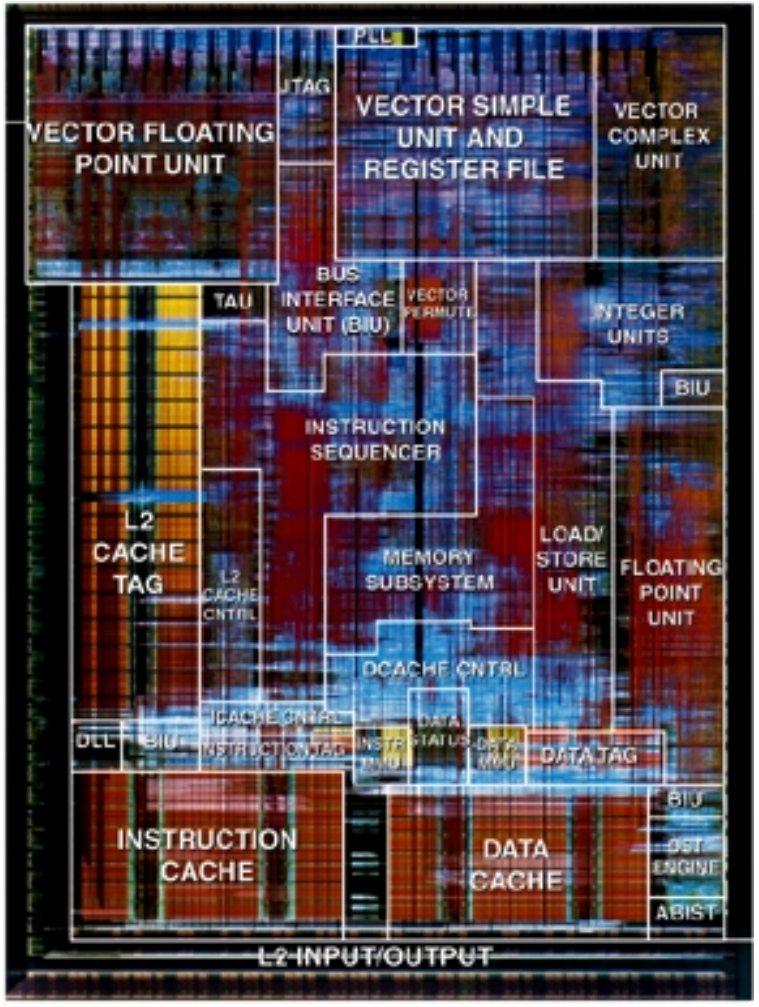
- Compilers cannot automatically vectorize standard C without hints (usually supplied by programmers using #pragma statements). This limitation makes C less popular than FORTRAN within the vector community which, in turn, depresses the quality of the few vector C compilers.
- Thus Motorola has followed the lead of DSP suppliers by mapping AltiVec instructions directly into C extensions. The result fits somewhere between assembly language and high-level languages with respect to both performance and productivity.
- Two third parties are independently working toward optimized VSIP libraries for AltiVec.

Productivity Niche

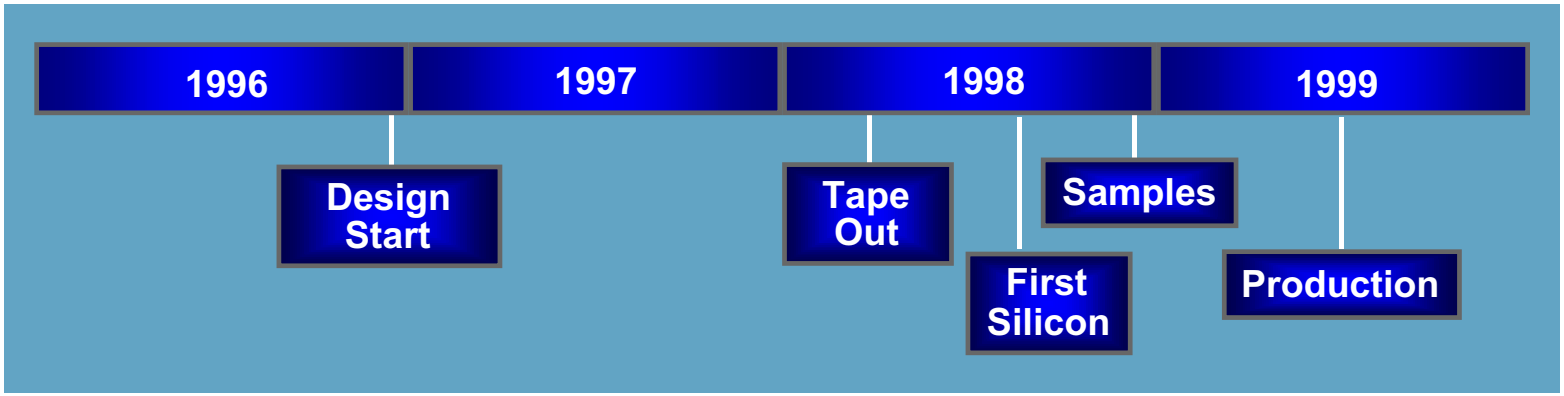
- AltiVec will beat a DSP chip in applications where programmer productivity and/or time-to-market is more of a concern than unit cost and hardware productivity.
  - Compilers offered by DSP vendors can only tie together hand-coded routines. In contrast, little rationale exists for programming a PowerPC core in assembly language.
  - PowerPCs support fully functional operating systems such as VxWorks.
  - An MMU helps debug code.
  - Real floating point is always easier than DSP block floating point (but a precision issue can exist with singles).
- Note: the first AltiVec chip will offer software productivity, not hardware productivity. DSP chips are highly integrated and boast optimized interfaces to the outside world. In contrast, embedded "G4" users must furnish their own DSP-like hardware interfaces.

G4

- The first PowerPC with AltiVec has the Apple code name "G4".
- 1.8 volt core consuming less than 8 watts (typical) at 400 MHz.
- Samples now. Production second calendar quarter.



Motorola G4 Schedule

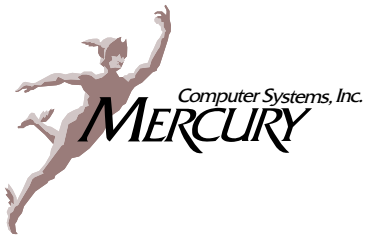


Will floating point DSP survive?

- The market for floating-point DSP is small. Defense applications might even dominate.
- New SIMD extensions to all of the major microprocessor families will hit the DSP vendors first where they are weakest—the high-end, floating-point niche. DSP vendors may elect to flee.

AltiVec Programming

November 1998, Revision 6.0, No NDA Required



Craig Lund
Consultant, Local Knowledge
Principal Technologist, Mercury Computer Systems
3 Langley Road
Durham, NH 03824-3424
Tel: +1 603 868 2300
Fax: +1 603 868 2301
clund@localk.com

Documentation & Tools

- Documentation
- Tools
- Libraries

Documentation

- *AltiVec Technology Programming Environments Manual*, revision 0.2, May 1998 describes the new PowerPC instructions— available on Motorola's web site at <www.mot.com/altivec>.
- *AltiVec Technology Programming Interface Manual*, revision 0.2, June, 1998, describes Motorola's extensions to the C Language. Alternatively, use Apple's *AltiVec Support In MrC[pp]*, Revision 1.1, June 3, 1998, available on Apple's web site;
- *PowerPC xxx Microprocessor Implementation Definition, Book IV*, version 1.3 — Confidential. This tells how AltiVec is implemented. It will be replaced by a public users' manual when G4 is announced.

Documentation (Assembler)

- *PowerPC Compiler Writer's Guide*, dated 9/3/96, available on IBM's web site at <www.chips.ibm.com/products/ppc/documents/compiler/cover.html>. Interesting to assembly programmers, not just compiler writers.
- *PowerPC Embedded Application Binary Interface*, available on the web at <www.esofta.com/softspecs.html>.
- *PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors*, revision 1; *Implementation Variances Relative to Rev. 1 of the Programming Environments Manual*, dated 3/97; *Programmer's Reference Guide*, revised 10/95, all available on Motorola's web site at <www.mot.com/powerpc>.

Apple Tools

- Apple has posted on its web site a complete C/C++ and assembler development environment for AltiVec. See <developer.apple.com/dev/tools/mpw-tools/index.html>.
- Apple has also posted an extension to MacOS 8.1/8.5 which allows existing PowerPC microprocessors (except the 601) to execute AltiVec code.
- There is no question that a Macintosh is the best AltiVec development environment at the moment.
- Apple has additional tools and example code on a CD-ROM available only to Apple developers who have signed NDAs with Apple and with access to Motorola G4 NDA information.
- Apple is planning to distribute an improved CDROM to all Apple developers soon.

Motorola Tools

- Motorola has an AltiVec C compiler and assembler that will run within Apple's MPW environment and within the Metrowerks IDE. However, there is no compelling reason to use Motorola's tools instead of Apple's.
- Motorola's tools are also available for Sun, NT, and AIX environments. However, AltiVec emulators are not. Thus, until G4 ships, there is no compelling reason to use these variants.
- Motorola's compilers are based upon source code licensed from Apogee. Motorola recently promised to turn its AltiVec enhancements over to Apogee for integration into Apogee's commercial products.
- Motorola does not have a support team in place for any of its tools. Thus Motorola prefers customers use third-party development tools.

Other Compilers

- Motorola has extended the popular Gnu C compiler to support AltiVec language extensions. The result is available to compiler suppliers upon request (as an example), but the result is not supported by Motorola.
- Motorola has briefed most existing PowerPC compiler suppliers on AltiVec. At this time only MetroWerks has issued a public commitment. Motorola expects more public statements in conjunction with the G4 announcement.

Simulator

- The G4 "simulator" is called Sim_G4. It is not really a simulator. Sim_G4 requires instruction traces in IBM's TT6 format, created elsewhere. Sim_G4 is a performance prediction tool.
- The only reliable way to get an AltiVec TT6 trace is on a Macintosh. You will need Apple software not found on Apple's web site (Motorola can provide it). Eventually Motorola will bundle another tool, called PowerSim, with Sim_G4 to enable TT6 generation on Suns and NT.
- Motorola is creating an emasculated version of Sim_G4 for distribution by Apple to its developers before the G4 announcement.

Operating System Support

- AltiVec requires operating system support (to save and restore registers). However, adding such support is trivial. Do not expect existing PowerPC OS vendors to release any code before a G4 chip becomes available on somebody's single board computer. Expect broad adoption very quickly thereafter.
- Support for Microsoft Windows CE is another story. Microsoft and Motorola have not yet committed to Windows CE for any high-end PowerPC. If that commitment happens, in the special G4 case, we must also wait for Apogee (Microsoft's PowerPC C supplier) to announce and ship its AltiVec compilers.
- OS support for Java mode is complex and Motorola does not supply the necessary code. Thus do not expect broad support for Java mode from OS vendors.

Libraries

- Motorola has several programmers working on optimized G4 libraries using the C compiler extensions. Motorola will soon post some of the resulting code on its web site.
- MCCI is promised two Navy contracts to produce a "performance" subset of the new VSIP library standard <www.vsip.org>. This library will become public property.
- MPI Software Technology has five programmers working on an optimized AltiVec VSIP library which they plan to resell.
- Motorola understands the importance of optimized libraries to potential customers attracted to AltiVec. Motorola has identified potential partners and is working to extract commitment.

Library Issues

- To complement the instruction set, no scalar math library yet exists for AltiVec (sin, pow, etc.). Apple's CD does have a 32-bit integer multiply and Apple is working on more. Motorola's manuals do show how to accomplish FP division and square root.
- The software support required for Java mode does not exist outside MacOS.
- Some existing DSP library API standards do not optimally map into AltiVec. For example, AltiVec performance benefits if the real and imaginary parts of complex numbers live in separate vectors. Also, AltiVec performance benefits from API knobs that can invoke cache management instructions. Fortunately, some of the designers of the new VSIP standard had access to advanced information on AltiVec.

Example Code

- The example code used in this presentation is available, complete with Makefiles and shell scripts. Apple's MPW environment was used. Contact Motorola or Mercury.
- Elements of the Motorola library will become available as source code. Contact Motorola.
- The MCCI library will become public property. Contact MCCI <mailto:crobbins@mcci-arl-va.com>
- Apple has example code at <http://developer.apple.com/hardware/altivec>.

AltiVec Example Code

- Subroutine vadd many ways
- L1 cache management via stripmining & prefetch
- L2 cache management via transients
- Exercise: dot product

Which vadd standard interface?

- We elected to use the interface from a library standard popular in the DSP world. It is based upon the library shipped with an early array processor from FPS. It is often called the FPS standard, or sometimes the SEG standard, after an organization which formally standardized it. Details at <http://seg.org/catalog/standards/Subroutine.html>.
- Other popular library standards include Blas <http://www.netlib.org/blas/index.html> (used in scientific markets), VSIP <http://vsip.org> (a modern replacement for FPS), and The Math Works <http://www.mathworks.com/products/compilerlibrary/compmath_functions.shtml> (a popular commercial product).

Subroutine vadd in C

```
#include "vadd.h"

void vadd (A, I, B, J, R, M, N)
    const float         *A;   /* input vector       */
    signed long int      I;   /* stride of A        */
    const float         *B;   /* input vector       */
    signed long int      J;   /* stride of B        */
    float               *R;   /* output vector      */
    signed long int      M;   /* stride of R        */
    unsigned long int    N;   /* number of elements*/
{
    for (; N > 0; N--) {
        *R  = *A + *B;
        A += I;
        B += J;
        R += M;
    }
}
```

Subroutine vadd with AltiVec

```
/* A, B, R assumed (vector float) aligned */
/* I, J, M assumed positive one           */
/* N assumed a multiple of                */
/* vec_step(vector float)which is 4       */

#define vA ((const vector float *)A)
#define vB ((const vector float *)B)
#define vR ((vector float *)R)

for (N = N / vec_step(vector float); N > 0; N--) {
   *vR = vec_add (*vA, *vB);
   vA++;
   vB++;
   vR++;
}
```

New Concepts on Prior Page

- AltiVec C extensions map into AltiVec instructions. For example, vec_add() maps into one of four AltiVec instructions (vaddubm, vadduhm, vadduwm, or vaddfp) depending upon the types of the arguments to vec_add().
- AltiVec supports more than one kind of add operation. Thus we also have C extensions vec_addc() and vec_adds(). This is why Motorola did not use "+".
- There are many new builtin C data types to match what AltiVec will support. This example uses "vector float" which is vector aligned and contains four singles.
- The builtin vec_step() maps into a constant. Its use enables source portability between AltiVec implementations with wider, or narrower, vector registers.

Using the Documentation

- Use Motorola's *AltiVec Technology Programming Interface Manual* to determine which AltiVec instruction each C extension will generate (given the types of the parameters that you use). The next slide shows a sample page from that document.
- Use Motorola's *AltiVec Technology Programming Environments Manual* to learn more about any specific AltiVec instruction that the C compiler issues. The slide after next shows a sample page.

Chapter 4
AltiVec Operations

4.1 Generic and Specific AltiVec Operators

The set of tables is organized alphabetically by generic operation name and defines the permitted generic and specific AltiVec operations. Each table describes a single generic AltiVec operation. Each line shows a valid set of argument types for that generic AltiVec operation, the result type for that set of argument types, and the specific AltiVec instruction generated for that set of arguments. For example, vec_add(vector unsigned char, vector unsigned char) maps to “vaddubm”.

In almost all cases, it is also permissible to use a specific AltiVec operator formed by adding “vec_” to the name of the operation in the Maps To column with that line’s set of argument types. For example, vec_vaddubm(vector unsigned char, vector unsigned char) has the same effect as vec_add(vector unsigned char, vector unsigned char). A few cases are prohibited because that set of argument types has been chosen to produce a different result type.

Any operation which is not explicitly permitted by this table is prohibited. The desperate programmer can cast arguments, if necessary, to use operators in bizarre ways. The less desperate programmer can request an extension or modification of the programming model!

4.1.1 vec_add(arg1, arg2)

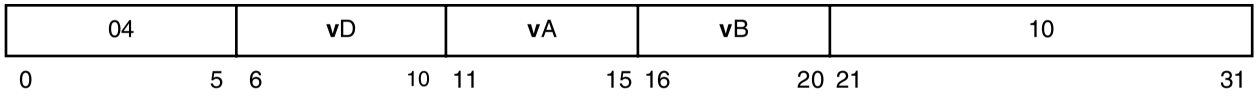
Each element of the result is the sum of the corresponding elements of **arg1** and **arg2**. The arithmetic is modular for integer types.

Table 4-1. vec_add(arg1, arg2)

Result	arg1	arg2	Maps to
vector unsigned char	vector unsigned char	vector unsigned char	vaddubm
vector unsigned char	vector unsigned char	vector bool char	vaddubm
vector unsigned char	vector bool char	vector unsigned char	vaddubm
vector signed char	vector signed char	vector signed char	vaddubm

Vector Add Floating Point

vaddfp **vD,vA,vB**




```
do i = 0,127,32
  (vD)i:i+31 ← RndToNearFP32((vA)i:i+31 +fp (vB)i:i+31)
end
```

The four 32-bit floating-point values in vA are added to the four 32-bit floating-point values in vB. The four intermediate results are rounded and placed in VD.

Other registers altered:

- None

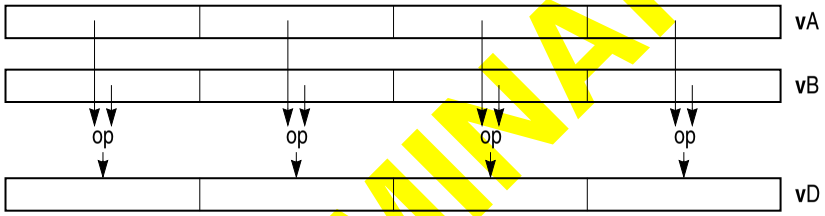


Figure 6-11. Basic 2-Source Operands —32-Bit Elements

New Concepts

- AltiVec consumes floats four at a time. What if your N is not a multiple of four? (this is a floating point example, other AltiVec data types have different multiples associated with them).
- Working with individual AltiVec vector elements is easy, but one must pay attention to the AltiVec load/store alignment rules.
- There is no way to move data between a PowerPC's register files except through memory.

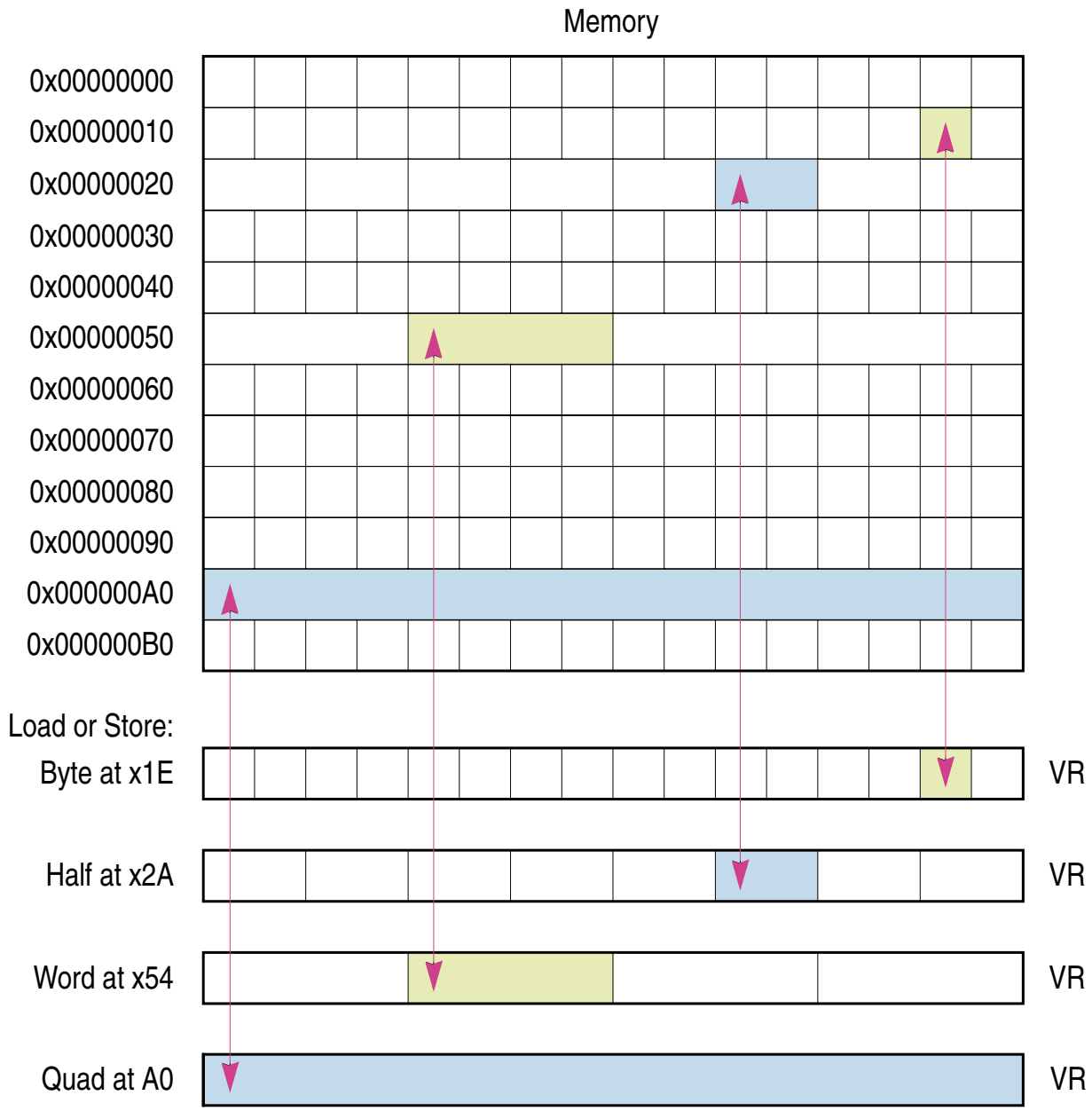
Allow any N

```
const unsigned long int  extra =
   N % vec_step(vector float);

/* Put 'for' loop from the previous slide here.*/

/* Note: we load beyond the end of the array.  */
/* There is no chance of access violation.     */
/* Extra, random values harmless in !Java mode,*/
/* may cause extra cycles in Java mode.        */

if (extra) {
   const vector float tempR=vec_add(*vA, *vB);
   unsigned long int i;
   for (i=0; i<extra; i++)
       vec_ste(tempR, i*sizeof(float), R);
}
```

New Concepts

- It is a very good idea to align AltiVec memory accesses. Doing so improves performance a tiny amount and it requires fewer lines of code if you know that your data is always aligned.
- If you are stuck with an existing interface (like the DSP world's FPS/SEG standard) which cannot guarantee vector alignment, the AltiVec permute unit will rescue you.
- If you don't consider alignment within your program code, and you accidently provide your program unaligned data, you will get wrong answers. You will not get any error message.

AltiVec Load/Store Alignment



Load a Single Element

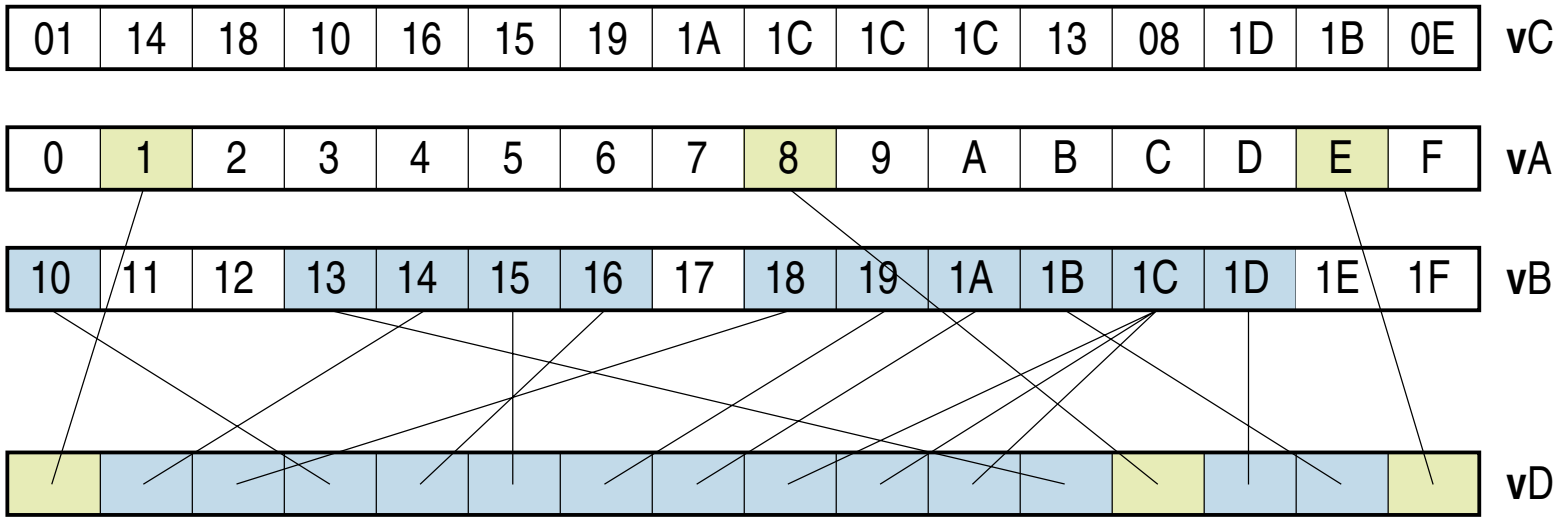
vector register = vec_lde (byte index, typed address)

Example: if you provide an address to a float, and you provide an index of zero, you will grab that float from memory. Exactly where in the vector register your float goes depends upon alignment. All other floats in that register become "undefined".

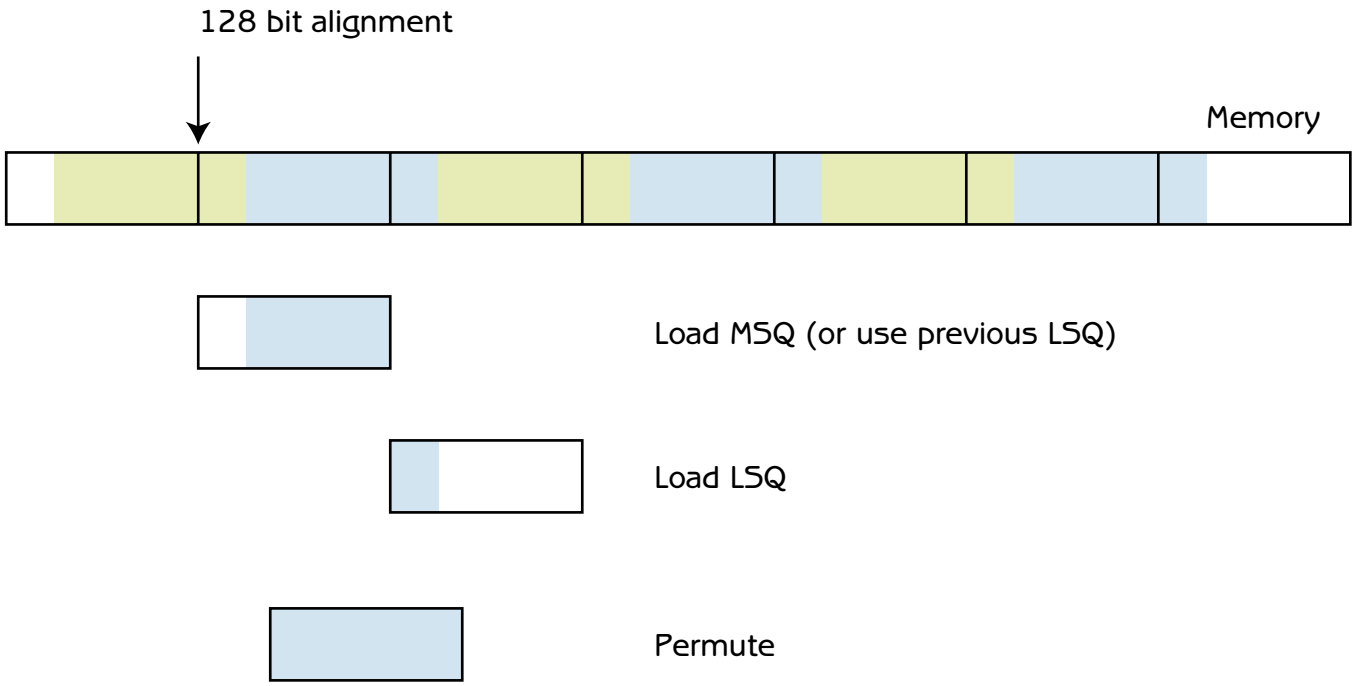
New Concepts

- What do you do if you need to work with data that is not aligned AltiVec's way? Use the permute unit to fix it.
- The permute unit is very general-purpose. For example, it can be used for a very fast small table look up.
- Moving bytes around to fix-up unaligned vectors is a special application. The bridge between the general and this special application are the lvsl and lvsr instructions. These instructions look at addresses and generate steering vectors for the permute unit.

Vector Permute Unit



Unaligned Load



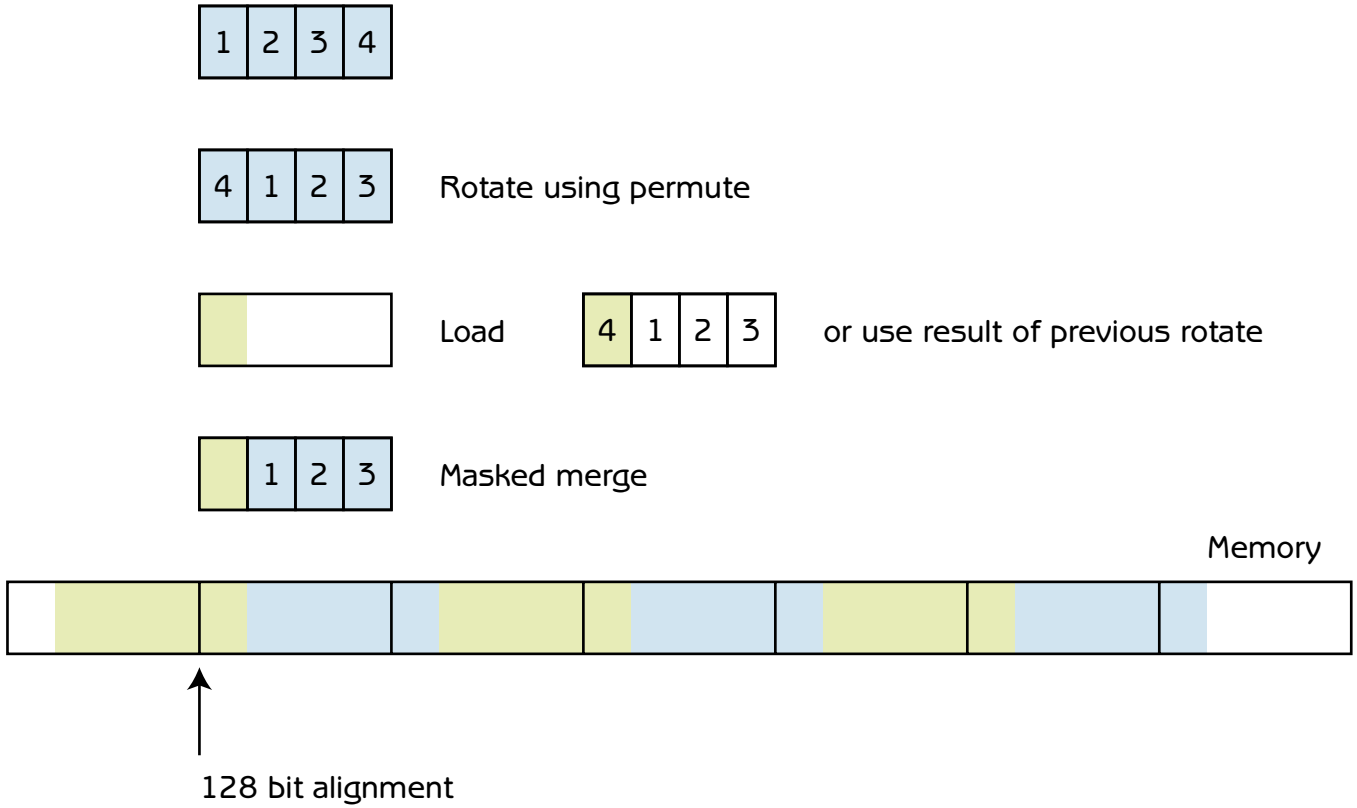
Allow any alignment for A

```
/* A any alignment; B, R still aligned       */
/* If A aligned, access violation can occur */
/* I, J, M assumed positive one              */
/* N a nonzero multiple of 4                 */

/* The vector we want straddles these two   */
/* vectors in memory                         */
vector float msqA, lsqA;

/* big-endian permute vector */
const vector unsigned char permA=vec_lvsl(0, A);

msqA = *vA; /* low order masked, gets MS quad */
for (N = N / vec_step(vector float); N > 0; N--) {
   A += vec_step(vector float);
   lsqA = *vA;
   *vR = vec_add(vec_perm(msqA,lsqA,permA),*vB);
   vB++;
   vR++;
   msqA = lsqA;
}
```

New Concepts

- Unaligned stores use the same concept, but backwards. Instead of extracting data from two aligned locations, one must mask data and merge it into two aligned locations.
- The lvsl and lvsr instructions help build the necessary mask. The vsel instruction does the merge.

Unaligned Stores



Allow any alignment for R

```
/* R any alignment; A, B aligned */
/* I, J, M assumed positive one  */
/* N a nonzero multiple of 4     */

vector float prevR, currR;

const vector unsigned char permR =
    vec_lvsr(0, R); /* big-endian */

const vector unsigned char maskR =
    vec_perm(vec_splat_u8(0), vec_splat_u8(-1), permR);

prevR = *vR; /* read first quad for merge */
for (N = N / vec_step(vector float); N > 0; N--) {
    currR = vec_add(*vA, *vB);
    /* rotate data right */
    currR = vec_perm(currR, currR, permR);
    *vR = vec_sel(prevR, currR, (vector bool long)maskR);
    prevR = currR;
    vA++;
    vB++;
    R += vec_step(vector float);
}
if ((unsigned int)R & 0xF) { /* skip if R was aligned */
    *vR = vec_sel(currR, *vR, (vector bool long)maskR);
}
```

New Concepts

- The "LRU" option on the AltiVec load and store instructions allows programmers to influence the behavior of both the L1 and L2 cache.
- LRU enables something called vector "chaining". This is where programmers tag vectors that are used in one part of a program, and then not used again for a long time. It is good policy not to move such vectors into L2. It is a good idea to evict such vectors from L1 before any other L1 data.

Chaining

```
/* R = A + B + C */
vadd ( A, 1, B, 1, R, 1, N, MMC);
vadd ( R, 1, C, 1, R, 1, N, MMM);
```

- Note the new flags to vadd. In the first instance, flag "MMC" tells vadd to load vector A using the LRU option (lvxl instead of lvx), vector B is also loaded using the LRU option, and vector R is stored normally (this keeps R in cache for reuse in the second vadd). M stands for "memory", C stands for "cache".
- In the second vadd, vector R is read normally, but stored using the LRU option. This store decision protects L2 from an unnecessary update.

Vadd with LRU Flag Support

```
/* A, B, R assumed (vector float) aligned */
/* I, J, M assumed positive one           */
/* N assumed a multiple of 4              */

#define do_it( loadA, loadB, storeR ) \
    for (N = N / vec_step(vector float); N > 0; N--) { \
        storeR(vec_add(loadA( 0, vA), loadB(0, vB)), 0, vR); \
        vA++; vB++; vR++;\
    } break;

switch (Flags) {
    case CCC: /* 0b000 */
        do_it(vec_ld, vec_ld, vec_st);
    case CCM: /* 0b001 */
        do_it(vec_ld, vec_ld, vec_stl);
    case CMC: /* 0b010 */
        do_it(vec_ld, vec_ldl, vec_st);
    case CMM: /* 0b011 */
        do_it(vec_ld, vec_ldl, vec_stl);
    case MCC: /* 0b100 */
        do_it(vec_ldl, vec_ld, vec_st);
    case MCM: /* 0b101 */
        do_it(vec_ldl, vec_ld, vec_stl);
    case MMC: /* 0b110 */
        do_it(vec_ldl, vec_ldl, vec_st);
    case MMM: /* 0b111 */
        do_it(vec_ldl, vec_ldl, vec_stl);
}
```

New Concepts

- If N is large, the benefits of chaining decrease. This occurs whenever the intermediate vectors are too large to fit into cache.
- Cache "stripmining" is the answer. Here we build and use intermediate vectors in pieces that will fit into cache.
- The size of these pieces depends upon the number of vectors, as well as the size and associativity of the caches involved. The largest piece size selected is usually the size in bytes used by a microprocessor's MMU (4096 bytes for a PowerPC).
- When stripmining it helps if your vectors all start with page alignment.

G4's L1 Data Cache

- 8-way Set Associative
- 32 byte line size (8 floats or 2 vectors)
- 32K (8 pages each with 1024 floats)

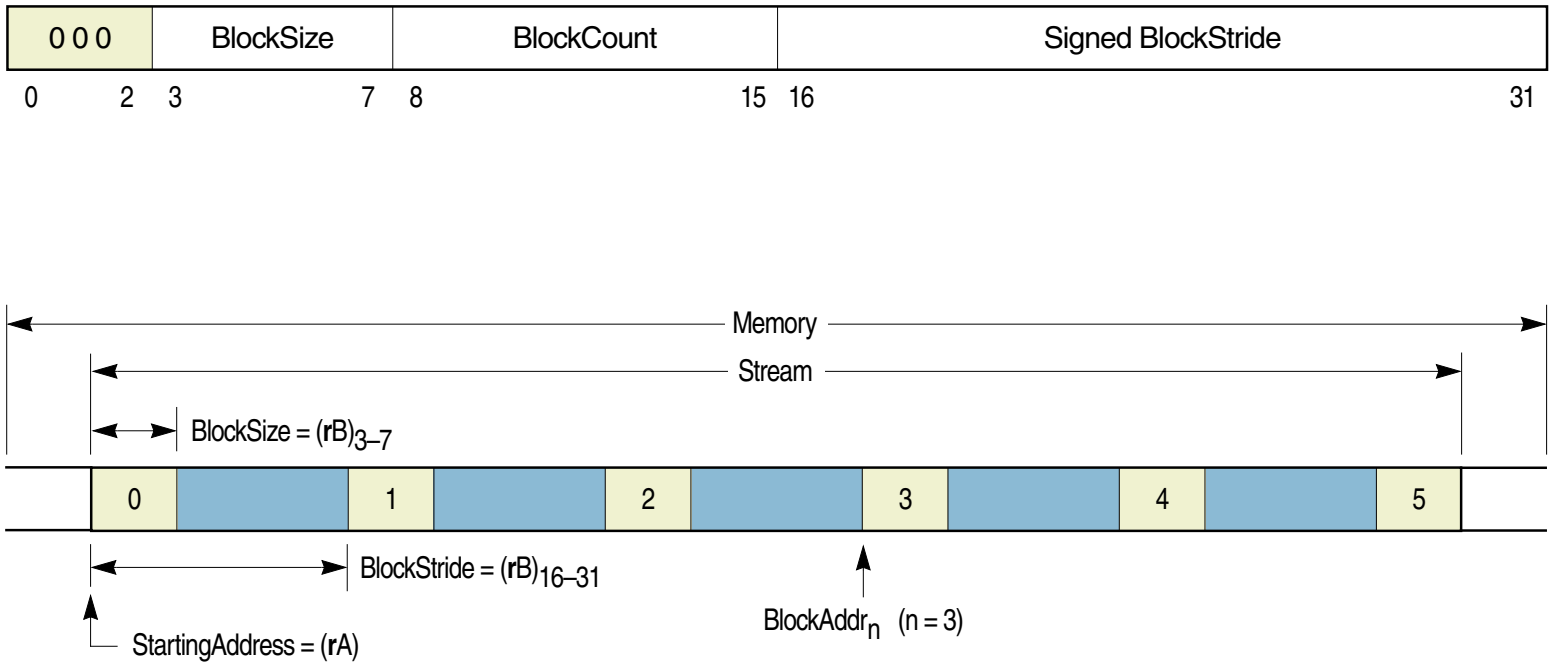
L1 Cache Stripmining

```
/* R = A + B + C */
const unsigned long int page=4096/sizeof(float),
      extra=N%page;

unsigned long int strip;
const float *pA=A, *pB=B, *pC=C;
float *pR=R;

for (strip=N/page; strip>0; strip--) {
   vadd(pA, 1, pB, 1, pR, 1, page, MMC);
   vadd(pR, 1, pC, 1, pR, 1, page, MMM);
   pA+=page; pB+=page; pC+=page; pR+=page;
}

if (extra) {
   vadd(pA, 1, pB, 1, pR, 1, extra, MMC);
   vadd(pR, 1, pC, 1, pR, 1, extra, MMM);
}
```

New Concepts

- Before AltiVec, when a PowerPC processor executed with data out of cache, the memory bus might sit idle. This is not a good use of resources. Ideally one wants to use these otherwise idle memory cycles to fetch data into cache that the processor will need later.
- The AltiVec dst family of instructions makes this happen. In DSP terms, the dst instructions allow "double buffering", where a DMA engine fetchs the "next" buffer while the processor uses the "current" buffer.
- Dst supports four prefetch channels.

Data Stream Touch Instructions




```
/* block size in vectors, number of blocks to prefetch, */
/* signed stride in bytes                                */
#define dstdata(size, count, stride) \
     (((unsigned char)size & 0b00011111)<<24 \
     | (unsigned char)count<<16 \
     | (signed short)stride)
```

Prefetch

```
/* R = A + B + C */

const unsigned long int page=4096/sizeof(float),
extra=N%page,
    dstrCmd=dstdata(1,page/vec_step(vector float),
        sizeof(vector float));
unsigned long int strip;
const float *pA=A, *pB=B, *pC=C;
float *pR=R;

for (strip=N/page; strip>0; strip--) {
    vec_dstt(pC, dstrCmd, 0);
    vadd(pA, 1, pB, 1, pR, 1, page, MMC);
    vec_dstt(pA+=page, dstrCmd, 0);
    vec_dstt(pB+=page, dstrCmd, 1);
    vadd(pR, 1, pC, 1, pR, 1, page, MMM);
    vec_dssall();
    pC+=page; pR+=page;
}
if (extra) {
    vec_dstt(pC, dstdata(1,extra/vec_step(vector float),
        sizeof(vector float)), 0);
    vadd(pA, 1, pB, 1, pR, 1, extra, MMC);
    vadd(pR, 1, pC, 1, pR, 1, extra, MMM);
    vec_dssall();
}
```

dstst and dststt

- Named "for Store"
- Actually used only for read-modify-write
- With G4, avoids bus KILL broadcasts

Transient Loads/Stores

- Data loaded with the transient option will never enter L2 (G4 implementation detail, not AltiVec).
- Use the transient option to protect data already in the L2 cache.
- Use transient on stores that you know will be quickly reused by an external DMA engine.
- All LRU instructions are also transient.
- The dstt and dststt prefetch instructions mark prefetched data as transient.

G4 L2 Data Cache

- Two-way associative
- Combined instruction and data cache
- 512KB, 1MB, or 2 MB
- Castout victim cache for L1
- Managed through selective use of transient load/stores
- Managed using careful placement of vectors within main memory (the Motorola C compiler's -Z, or data skewing option exists for this reason)

Exercise: Dot Product

```
#include "dotpr.h"

void dotpr (A, I, B, J, R, N)
   const float         *A; /* input vector          */
   signed long int      I; /* stride of A           */
   const float         *B; /* input vector          */
   signed long int      J; /* stride of B           */
   float               *R; /* output element        */
   unsigned long int    N; /* number of elements    */
{
   float sum = 0;
   for (; N > 0; N--) {
      sum += *A * *B;
      A += I;
      B += J;
   }
   *R = sum;
}
```

Dot Product Hints

- Use vec_madd() to fuse the multiply with the add.
- Use vec_sld() to help sum the four elements in your final vector (if you were writing an integer dot product you could alternatively use an AltiVec instruction that sums across).
- What if N is not a multiple of four? Hint: use vec_sro().
- There is dot product example code on the CD-ROM. On G4 my inner loop uses 5 cycles to process two vec_madd()s.

G4 Performance Tuning

- Reading instruction traces to find stalls
- Loop unrolling
- Hiding load latency
- Compiler Performance

Subroutine vadd in Assembler

```
            INCLUDE 'vaddAsm.h'

.vadd       slwi      I,I,2      ; I * sizeof(A)
            slwi      J,J,2      ; J * sizeof(B)
            slwi      M,M,2      ; M * sizeof(R)
            cmplwi    N,0        ; Compare N to zero
            beqlr                ; return if N=0
            mtctr     N          ; N now in CTR
            subf      vA,I,vA    ; prime the loop
            subf      vB,J,vB
            subf      vR,K,vR
@loop1      lfsux     f0,vA,I    ; Load *A, A+=I
            lfsux     f1,vB,J    ; Load *B, B+=J
            fadds     f0,f0,f1   ; *A + *B
            stfsux    f0,vR,M    ; Store *R, R+=M
            bdnz+     @loop1     ; N-=1, loop nonzero
            blr+                 ; return
            END
```

New Concepts

- There are an infinite number of ways to write any given program. Some ways execute faster than others. Often faster programs require more lines of code, or access memory locations beyond the end of an array.
- To deliver the ultimate performance, one must understand what processor "stalls" are and how to avoid them.
- Motorola has a tool which will show stalls called Sim_G4. The input to Sim_G4 is a file of instruction traces in a format called TT6.
- You can get a trace from a Macintosh, or on any host when Motorola's PowerSim tool becomes available. To examine TT6 files, IBM has a "TT6 profiler" tool. Apple has a "T-VIZ TT6" profiler tool.

Sim_G4 Instruction Trace

```
16 E E F D R R F F     20:(   12) lfsux   F0,R3,R4         CB
16         -
17 F E F E D - R R     21:(   12) lfsux   F1,R5,R6         MAX
17           - D       22:(   13) fadds   F0,F0,F1
18 R E F F E D D -     23:(   13) stfsux  F0,R7,R8         MAX
18               - F   24:(   14) bc+     16,0,0xfff0
19 D R R F F D E F     25:(   15) lfsux   F0,R3,R4         CB
19 -
20 E D - R R E E F     26:(   17) lfsux   F1,R5,R6         MAX
20   - D               27:(   18) fadds   F0,F0,F1
21 F E D D - E E F     28:(   18) stfsux  F0,R7,R8         MAX
21       - F           29:(   19) bc+     16,0,0xfff0
22 F F D E F F E F                                        CB
22
23 F F E E F R E F                                        CB
23
```

Sim_G4 clock

Pipeline Status

Instruction Number

Clock cycle when fetched

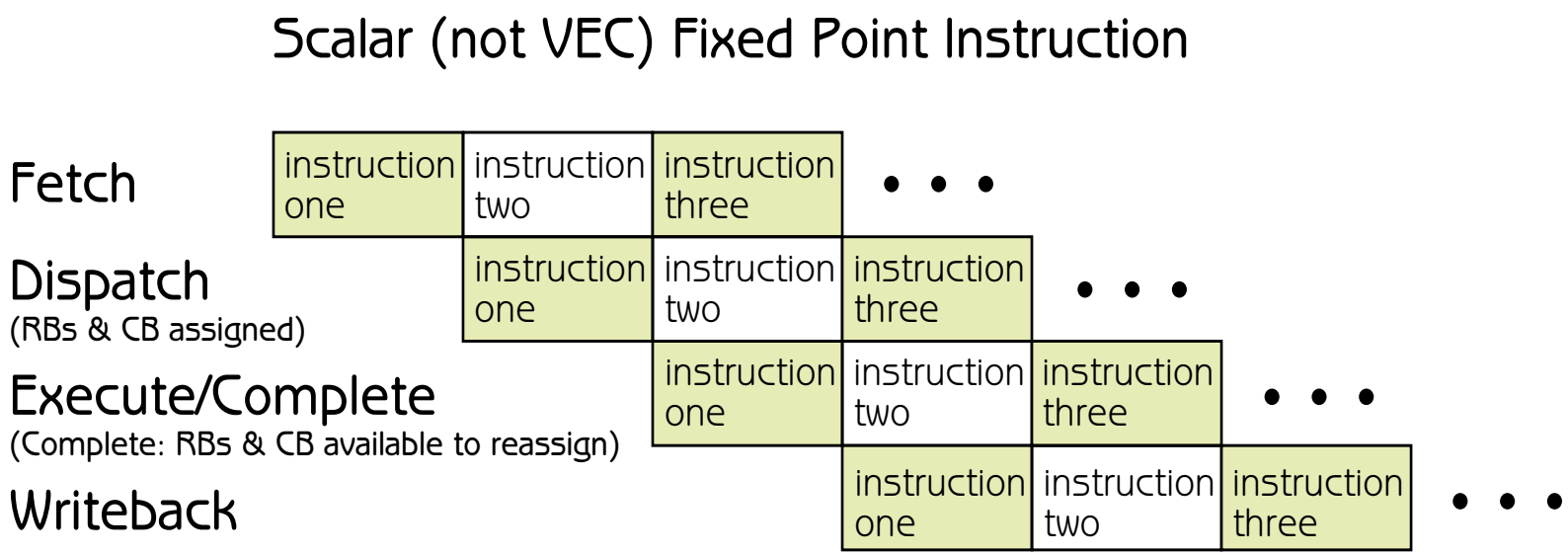
Dispatch Status

Shows one loop iteration

Caches warmed

8 cycles for 2 floats (4 cycles/float)

Sim_G4 Pipeline Status



Sim_G4 Pipeline Status

The Fetch stage is shown as a cycle number, it is not indicated in the pipeline status area.
'D' indicates a dispatched instruction, possibly executing or waiting on a resource conflict.
'E' shows execution.
'F' shows "Finished", i. e. ready to complete, waiting on a branch or resource conflict.
'R' shows writeback and instruction retirement.
'@' indicates a folded branch instruction.

Scalar Floating Point Pipeline

1. fetch
2. dispatch (RBs and CB assigned)
3. execute
4. execute
5. execute (results go into RB here)
6. complete (if exceptions disabled)
7. complete (if exceptions enabled)
7. or 8. writeback

Dependent instructions can start (see 5) before the RB and CB resources are released (in 6 or 7).

AltiVec Floating Point Pipeline

1. fetch
2. dispatch
3. execute
4. execute
5. execute
6. execute (complete if non-java mode)
7. writeback (non-java) or complete (java)
8. writeback (java mode)

G4 Resources

- Completion buffer queue eight deep
- Six rename buffers per register file
- Three register files (GPR, FPR, VR)
- Six instruction buffers
- Two entry "finished" store queue
- Four entry completed store queue
- Cache described previously

Sim_G4 Dispatch Status

CB no completion buffer available

DLS cannot dispatch two loads nor two stores in a single cycle

DS dispatch serialization

FPR need a rename register for a scalar floating point register

FPUB scalar floating point unit busy

FXUB scalar fixed point unit busy

GPR need a rename register for a scalar integer register

IB no instruction buffer available

LSUB LSU busy

LSUF LSU full

MAX maximum dispatch obtained

MD mispredict drain

SPEC maximum speculation reached

SUB system unit busy

TS tail serialization

VAUB AltiVec arithmetic unit busy

VAUF AltiVec arithmentic unit not empty

VPUB AltiVec permute unit busy

VPUF AltiVec permute unit not empty

VR need a rename register for an AltiVec register

New Concepts

- We just saw how programmers can "see" processor stalls. We will now demonstrate techniques for removing them.
- The first step is to introduce a simple AltiVec vadd example that we can build upon.

Simple AltiVec Assembly Code

```
; Assume A, B, R vector aligned
; Assume I, J, M one
; Assume N a nonzero multiple of 4

        INCLUDE 'vaddAsm.h'

off     SET     r10             ; No lvux instruction

.vadd   srwi    N,N,2           ; N = N/4
        mtctr   N               ; N now in CTR
        li      off,0           ; Point to first vector
@loop4  lvxl    v0,off,vA       ; Load *A
        lvxl    v1,off,vB       ; Load *B
        vaddfp  v0,v0,v1        ; *A + *B
        stvx    v0,off,vR       ; Store *R
        addi    off,off,16      ; Point to next vector
        bdnz+   @loop4          ; N-=1, loop nonzero
        blr+                    ; return
        END
```

Cycles from Simple AltiVec Code

- Nine cycles gives eight floats (1.125)

New Concepts

- "Unrolling" a loop amortizes the loop overhead over a longer period of computation. In this vadd example loop overhead is the combination of addi and bdnz+ statements.
- Here is unrolling by two

    start loop:

        add one vector

        add another vector

        loop back to top

Loop Unrolling (by Two)

```
; Assume A, B, R vector aligned
; Assume I, J, M one
; Assume N a nonzero multiple of 8

        INCLUDE 'vaddAsm.h'

off     SET     r10         ; off incremented by two vectors
vA2     SET     r4          ; Points to A in second vector
vB2     SET     r6          ; Points to B in second vector
vR2     SET     r8          ; Points to R in second vector

.vadd   srwi    N,N,3       ; N = N/8
        mtctr   N           ; N now in CTR
        addi    vA2,vA,16   ; A2 is vector after A
        addi    vB2,vB,16   ; B2 is vector after B
        addi    vR2,vR,16   ; R2 is vector after R
        li      off,0       ; Point to first vector pair
@loop8  lvxl    v0,off,vA   ; Load *A
        lvxl    v1,off,vB   ; Load *B
        vaddfp  v0,v0,v1    ; *A + *B
        stvx    v0,off,vR   ; Store *R
        lvxl    v2,off,vA2  ; Load *A2
        lvxl    v3,off,vB2  ; Load *B2
        vaddfp  v2,v2,v3    ; *A2 + *B2
        stvx    v2,off,vR2  ; Store *R2
        addi    off,off,32  ; Point at next two vectors
        bdnz+   @loop8      ; N-=1, loop nonzero
        blr+                ; return
        END
```

Cycles AltiVec Hand Unrolled Two

- Eight cycles gives eight floats (1.0)

New Concepts

- You can save additional cycles by rearranging the code to remove dependency stalls (hide load latency).
- Here we interleave like this:

  loads for add#1

  start loop:

  loads for add#2

  add#1

  loads for add#3

  add#2

  loads for add#4

  etc...

- You could do the same thing for stores, but in this case there is no gain.

Hide Load Latency

```
; This routine loads one vector element beyond the end of both vA & vB.
; An access violation can result.

; Assume A, B, R vector aligned
; Assume I, J, M one
; Assume N a nonzero multiple of 8

          INCLUDE 'vaddAsm.h'

off       SET       r10             ; off incremented by two vectors
vA2       SET       r4              ; Points to vA in second vector
vB2       SET       r6              ; Points to vB in second vector
vR2       SET       r8              ; Points to vR in second vector

.vadd     srwi      N,N,3           ; N = N/8
          mtctr     N               ; N now in CTR
          lvxl      v0,0,vA         ; Load *vA (to prime loop)
          addi      vA2,vA,16       ; vA2 is vector after vA
          lvxl      v1,0,vB         ; Load *vB
          addi      vB2,vB,16       ; B2 is vector after vB
          addi      vR2,vR,16       ; R2 is vector after vR
          addi      vA,vA,32        ; Get vA, vB ready for next loop iteration
          addi      vB,vB,32
          li        off,0           ; Point to first vector pair
@loop8    lvxl      v2,off,vA2      ; Load *A2
          lvxl      v3,off,vB2      ; Load *B2
          vaddfp    v0,v0,v1        ; *vA + *vB
          stvx      v0,off,vR       ; Store *vR
          lvxl      v0,off,vA       ; Load *vA (for next iteration)
          lvxl      v1,off,vB       ; Load *vB (for next iteration)
          vaddfp    v2,v2,v3        ; *A2 + *B2
          stvx      v2,off,vR2      ; Store *R2
          addi      off,off,32      ; Point at next two vectors
          bdnz+     @loop8          ; N-=1, loop nonzero
          blr+                      ; return
          END
```

Hide Load Latency

- Six cycles gives eight floats (0.75)

New Concepts

- Often unrolling further hides additional cycles. That is not the case with vadd. 24 cycles gives 32 floats (0.75 again).
- Eight times is the most that you can unroll most things (before you run out of general purpose registers).

New Concepts

- You can unroll loops and overlap loads/stores using the C compiler too.
- Compiler performance varies. You can get results equivalent to hand code.
- The example which follows hand unrolls in C by eight. This much unrolling stresses the C compiler because almost every general purpose register is required (including the registers holding the function arguments).
- Some C compilers can unroll automatically. Using the simple C code from "Subroutine vadd with AltiVec," the Motorola compiler unrolled by eight automatically, but with poor resulting performance.

Compiler Hand Unrolled Eight

```
/* A, B, R assumed (vector float) aligned */
/* I, J, M assumed positive one */
/* N assumed a multiple of 32 */
/* Will access one vector beyond A, B, and C */
/* Uses 16 vector registers (of 32) */
/* Uses 22 general purpose registers plus 7 for parameters */

unsigned int off=0;
const vector float *const vA2=vA +1, *const vB2=vB +1;
const vector float *const vA3=vA2+1, *const vB3=vB2+1;
const vector float *const vA4=vA3+1, *const vB4=vB3+1;
const vector float *const vA5=vA4+1, *const vB5=vB4+1;
const vector float *const vA6=vA5+1, *const vB6=vB5+1;
const vector float *const vA7=vA6+1, *const vB7=vB6+1;
const vector float *const vA8=vA7+1, *const vB8=vB7+1;
vector float *const vR2=vR +1; vector float *const vR3=vR2+1;
vector float *const vR4=vR3+1; vector float *const vR5=vR4+1;
vector float *const vR6=vR5+1; vector float *const vR7=vR6+1;
vector float *const vR8=vR7+1;
vector float valA, valA2, valA3, valA4, valA5, valA6, valA7, valA8;
vector float valB, valB2, valB3, valB4, valB5, valB6, valB7, valB8;

valA = vec_ld(0,vA); valB = vec_ld(0,vB); vA += 8; vB += 8;
for (N=N/(vec_step(vector float)*8); N>0; N--) {
     valA2 = vec_ld(off,vA2); valB2 = vec_ld(off,vB2);
     vec_st(vec_add(valA,  valB), off,vR);
     valA3 = vec_ld(off,vA3); valB3 = vec_ld(off,vB3);
     vec_st(vec_add(valA2, valB2), off,vR2);
     valA4 = vec_ld(off,vA4); valB4 = vec_ld(off,vB4);
     vec_st(vec_add(valA3, valB3), off,vR3);
     valA5 = vec_ld(off,vA5); valB5 = vec_ld(off,vB5);
     vec_st(vec_add(valA4, valB4), off,vR4);
     valA6 = vec_ld(off,vA6); valB6 = vec_ld(off,vB6);
     vec_st(vec_add(valA5, valB5), off,vR5);
     valA7 = vec_ld(off,vA7); valB7 = vec_ld(off,vB7);
     vec_st(vec_add(valA6, valB6), off,vR6);
     valA8 = vec_ld(off,vA8); valB8 = vec_ld(off,vB8);
     vec_st(vec_add(valA7, valB7), off,vR7);
     valA = vec_ld(off,vA);  valB = vec_ld(off,vB);
     vec_st(vec_add(valA8, valB8), off,vR8);
     off += vec_step(vector float)*sizeof(float)*8; }
```

C Compiler Performance

- Apple MrC (did not allocate enough registers)
  31 cycles gives 32 floats (0.97)
- Apple MrC (auto unroll)
  32 cycles gives 32 floats (1.00)
- Motorola mcc (instruction scheduling off)
  24 cycles gives 32 floats (0.75) — same as hand code
- Motorola mcc (auto unrolled eight, scheduling off)
  32 cycles gives 32 floats (1.00)
- Metrowerks (instruction scheduling off)
  24 cycles gives 32 floats (0.75) — same as hand code
- Metrowerks (auto unrolled, scheduling on)
  37 cycles gives 32 floats (1.16)

C Hand Unrolled Two, Any N

```
/* A, B, R assumed (vector float) aligned     */
/* I, J, M assumed positive one               */
/* N anything                                 */
/* Will access one vector beyond A, B, and C  */
/*  if N is a multiple of 4                   */

unsigned long int off=0, extra=N%8;
const vector float *const vA2=vA+1, *const vB2=vB+1;
vector float *const vR2=vR+1, valA, valA2, valB, valB2;

valA = vec_ld(0,vA); valB = vec_ld(0,vB);
vA+=2; vB+=2;
for (N=N/(vec_step(vector float)*2); N>0; N--) {
    valA2 = vec_ld(off,vA2); valB2 = vec_ld(off,vB2);
    vec_st(vec_add(valA,  valB), off,vR);
    valA = vec_ld(off,vA);  valB = vec_ld(off,vB);
    vec_st(vec_add(valA2, valB2), off,vR2);
    off += vec_step(vector float) * sizeof(float) * 2;
}

if (extra>=vec_step(vector float)) {
    vec_st(vec_add(valA,  valB), off, vR);
    valA = vec_ld(off,vA2);  valB = vec_ld(off,vB2);
    extra-=vec_step(vector float);
    off+=vec_step(vector float) * sizeof(float);
}

if (extra) {
    unsigned long int i;
    const vector float valR = vec_add (valA, valB);
    for (i=0; i<extra; i++)
        vec_ste(valR, i*sizeof(float)+off, R); }
```

Send comments to:

Craig Lund
Local Knowledge
3 Langley Road
Durham, NH 03824-3424
Tel 603 868 2300
Fax 603 868 2301
<clund@localk.com>