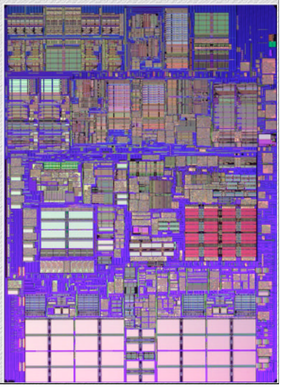
Vector Processing and Altivec

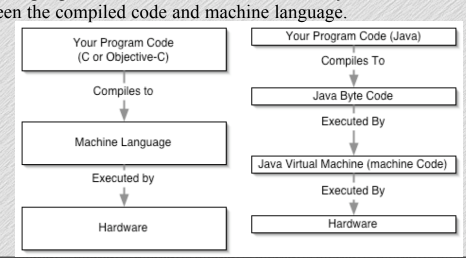
By: Steven Stanek

Important News:
NYY 9, BOS 2



A tour of the abstraction layers

- When we write code in C or Objective-C, we are not writing code that the machine can natively execute.
- Instead, the code must be **compiled** down to **machine language**, the instructions which the processor can execute.
- Machine language is specified by the **Instruction Set Architecture** (ISA) of a particular processor.
- Other languages such as Java and Scheme actually have a virtual machine between the compiled code and machine language.



How RISC Assembly Languages Work

- Think of an **Assembly Language** as an Englishified version of the machine language; humans like to use words, not strings of numbers of specify operations. Assembly Language is (usually) a one to one mapping of machine instructions to hopefully pronouncable words.
- **Registers**: There are a small number (32 in MIPs) of general purpose (integer) registers. The machine language operations are only between registers except for memory instructions.
- Assembly languages consist of very simple operations such as:
  - Subtract, Add or do logical operations on two values and store in a third
  - Branch: Go somewhere else in the code if a certain condition is met
  - Jump: Go somewhere else in the code regardless
  - Load Values from Memory to Registers
  - Store Values from Memory to Registers
- A program called an **assembler** converts the English descriptions to the actual binary code which computers run.
  - ex: The MIPs instruction "add $1, $2, $3" in assembly is "0x00430820" in binary machine language. Likewise "0x00430820" always means "add $1, $2, $3".

C vs. Compiled Code Example

Code C:
```
int fact(int k)
{
    int i, result;
    result=1;
    for(i=k; i!=0 ; i--)
        result =result* i;

    return result;
}
```

MIPS1 Assembly:
```
#Some Function Entering Code
#k is in a0
addiu $v0, $0, 1 #Let v0 be result
addiu $t2, $0, $a0 #Let t2 be i
L1: #Test i!=0
bne $t2, $0, L2 #Go to Beginning of Loop if i!=0
j L3 #Otherwise go to end of loop
L2: #Go Here i!=0
mult $v0, $t2 #Do the result *i multiply
mflo $v0 #Store Multiply result in i
addiu $t2, $t2, -1  #decrement i
j L1 #Loop again
L3: #Go Here if i==0
jr $ra #return
```

Processor Design and Performance

Processor designers are always trying to make their processors go faster. Increasing clock speed alone is not sufficient to see the increases in performance predicted by Moore's Law. Therefore in addition to making chips with faster clock rates, processor designers must chips them do more per clock cycle.

One way of making chips do more per clock cycle is for them to try to get more "bang for the buck" out of instructions. They can:

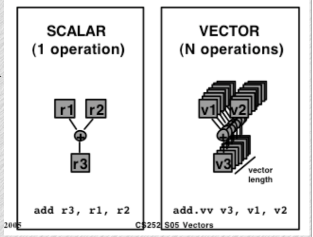
- Issue more than one instruction per clock cycle. This can be done two ways:
  - The compiler can specify packages of instructions which are allowed to execute at the same time. (Itanium)
  - The processor can figure out what instructions can execute together on its own. (Most modern processors)
- Allow instructions which operate on more than one value per clock cycle.
  - This is what Altivec is about.

These techniques are not mutually exclusive. Processors can use combinations of several of them.

Vector Processing

When we talk about instructions that operate on more than value, we are talking about vector operations. Operations which are applied in ways such as the following:

A = <a1, a2, a3...an>
B = <b1, b2, b3...bn>
A+B=<a1+b1, a2+b2, a3+b3...an+bn>



It is easy to see that we can add a1 & b1 at the same time we add a2&b2, a3&b3 and so forth. That is, these operations are independent of eachother.

We can add instruction which do these vector operations to the instruction set architecture of a processor.

C vs. Matlab

C has several very interesting properties:

- C is designed to be very low level. Operations in C are implemented in a small constant number of assembly instructions.
- C is also designed to be platform independent. Basic C code can be ported from one platform to another with relative ease. (This of course does not include system calls.)
- Because of the above two properties C is a scalar language. Its operations are only on scalar variables which represent a single quantity.

Contrast this with Matlab

- Individual Matlab operations can add arbitrary sized matrices or vectors.
  - This means that you can be far more expressive with a short piece of Matlab code than you can with C.
- The individual Matlab operations do not however map to hardware. Instead lower level functions must be called which do lots of scalar computations in the same manner as your linear algebra book.

Altivec Basics

- Altivec provides instructions which operate on 128 bit registers.
- The 128 bits can be divided up into a number of ways
  - 128 X 1 bit booleans
  - 16 X 8 bit signed or unsigned chars
  - 8 X 16 bit signed or unsigned shorts
  - 4 X 32 bit signed or unsigned longs (ints)
  - 4 X 32 bit floats
- Different operations are specified for each representation. These include
  - Addition & Subtraction
  - And, OR, XOR, NOR
  - Independent Shifts on individual elements
  - Taking averages
  - Various comparisons
  - Estimates of: logs, 2 to the power, reciprocals, square roots
- A full list of all the supported operations (instructions) is available at: http://developer.apple.com/hardware/ve/instruction_crossref.html

How to use Altivec in C

Altivec Vectors are declared as one of the following types:

- vector {unsigned/signed/bool} char
- vector {unsigned/signed/bool} short
- vector {signed/unsigned/bool} int
- vector float
- vector pixel (for 1/5/5/5 pixels)

We call operations in code as if they were functions (though they aren't)

- *NOTE*: If you look at the Altivec specs, there are "general" and "specific" versions of the operations. The general versions will use types to figure out which specific instructions to call.
- examples
  - c=vec_add(a,b) //Adds a & b and puts the result in c
  - c=vec_abs(a)   //Takes the absolute value of a and puts the result in c
  - d=vec_madd(a,b,c) //Multiply a & b, add c and put the result in d
    - This is for floats only similar operations exist for integer types: vec_madds, vec_mladd, vec_mladds
  - c=vec_re(a) //Puts an estimate of the reciprocal of a in c (floats only)

How to use Altivec in C (continued)

All of these operations map directly to single hardware instructions. In addition to the hardware instructions, Apple provides a very large Altivec optimized libary called veclib.h. It contains libraries for:

- A basic operations library which implements some operations such as integer division which are not included supported in hardware and support for 64 or 128 bit numbers.
- Additional floating point operations including more precise versions of the approximated operations.
- A signal processing library including FFTs
- A big number library
- A "Basic Linear Algebra System" which consists mostly of code to deal with matricies of various fixed sizes.

How to use Altivec in C (continued)

There is no magical instruction for moving the contents of a single int into a vector type variable. Instead, you have to go through memory.

Code on how to do this:

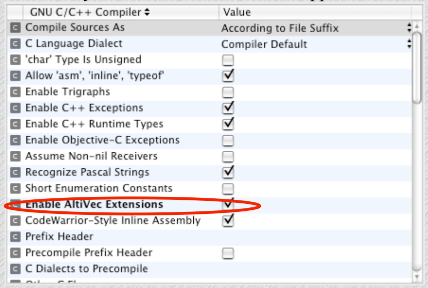
```
vector float vecFromFloats(float a, float b, float c, float d) {
        vector float returnme;
        float *returnme_ptr;
        returnme_ptr = &returnme;
        returnme_ptr[0] = a;
        returnme_ptr[1] = b;
        returnme_ptr[2] = c;
        returnme_ptr[3] = d;
        return returnme;
}
```

Note: Due to memory alignment issues you CANNOT cast arrays to vectors!!! That is NEVER DO THIS:

```
void func(float *farr1) {
        vector float *a;
        a = farr1;
        //Some vector op on *a
}
```

How to Use Altivec in Xcode and GCC

In Xcode, you have to enable Altivec support:



GNU C/C++ Compiler	Value
Compile Sources As	According to File Suffix
C Language Dialect	Compiler Default
'char' Type Is Unsigned	☐
Allow 'asm', 'inline', 'typeof'	✓
Enable Trigraphs	☐
Enable C++ Exceptions	✓
Enable C++ Runtime Types	✓
Enable Objective-C Exceptions	☐
Assume Non-nil Receivers	☐
Recognize Pascal Strings	✓
Short Enumeration Constants	☐
Enable AltiVec Extensions	✓
CodeWarrior-Style Inline Assembly	✓
Prefix Header	
Precompile Prefix Header	☐
C Dialects to Precompile	

To use Apple's library, add the vecLib.framework. Include it in the appropriate files as "#include <veclib/vecLib.h>".

In gcc: compile with the -faltivec flag

Example Altivec Program

Example: Find the center of mass of a bunch of points. We assume that the array of vectors we are passed stores point information in the following form: <xcoord, ycoord, zcoord, pointMass>

```
vector float centermass(vector float *points, int len) {
        vector float returnme = vector float <0,0,0,0>;
        vector float masssum = vector float <0,0,0,0>;
        float *returnme_ptr = &returnme;
        int i;

        for(i=0; i < len; i++){
                vector float massvar;
                massvar = vec_splat(points[i], 4);
                returnme = vec_madd(massvar, points[i], returnme);
                masssum = vec_add(masssum, massvar);
        }

        returnme=vdivf(returnme, masssum);
        return returnme;
}
```

Some Notes

- The Altivec unit is really a “short vector processor”. It deals with fixed length (128 bit) vectors and can issue at least one vector operation per clock cycle.
- The traditional term “Vector Processor” refered to a processor which dealt with variable length vectors (with some high top limit, say around 1024 64 bit words) and took many clock cycles to do computations. (ex: Cray-1)
- Vector processing is often known as Single Instruction Multiple Data or SIMD processing.
- There are printf and scanf conversion specifiers for reading in and writing out vectors. (See the Altivec documentation)
- It is probably quite costly to go through memory to move scalar values in and out of vector variables, so I’d suggest trying to minimize such operations.
- In general, vector processing puts lots of stress on memory systems because you tend to use it for applications which step through memory as opposed to those which exploit spatial locality.

PowerPC Under the Hood

- The reason we have to go through memory to move from scalars to vectors is that the registers used for integer, floating point and vector operations are seperate! (32 GPR, 32 FPR, 32 VRF)
- The G4: Can issue up to
  - 2 Altivec instructions per cycle
  - 3 General purpose (integer) instructions per cycle
  - 1 Regular Floating Point instruction per cycle
  - Can fetch a maximum of 4 instructions per cycle

- The G5: Can issue up to
  - 4 Altivec instructions per cycle
  - 2 Regular Floating Point instructions per cycle
  - 2 Load/Store instructions per cycle
  - 4 General purpose (integer) instructions per cycle
  - Can fetch & issue a maximum of 8 instructions per cycle

Other Resources

- Apple’s Velocity Engine Site: http://developer.apple.com/hardware/ve/
- Freescale’s (formerly Motorola Semiconductor) Altivec Technology Programming Environments Manual: http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPEM.pdf
- The chapter on Vector processing from Hennessey and Patterson, the Graduate version of Patterson and Hennessey is available online at: www.mkp.com/CA3/ . It is appendix G.
- Professor Culler’s Slides on Vector Processing including some techniques in how to use vector processing are available at: http://www.cs.berkeley.edu/~culler/courses/cs252-s05/lectures/cs252s05-lec10-vectors.pdf