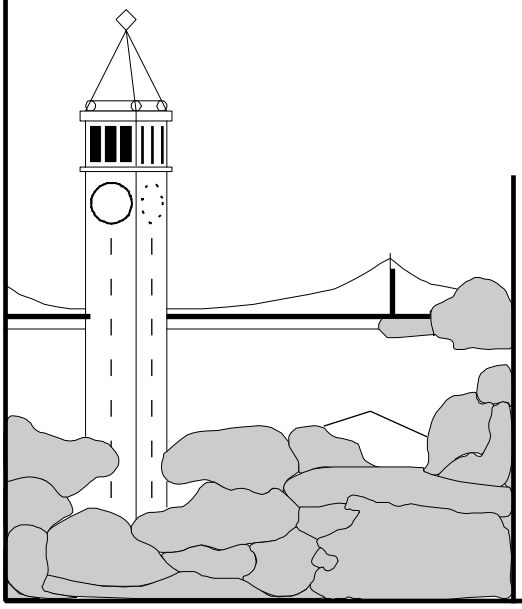
Measuring the Performance of Multimedia Instruction Sets

Nathan T. Slingerland and Alan Jay Smith



Report No. UCB/CSD-00-1125

December 2000

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Measuring the Performance of Multimedia Instruction Sets

Nathan Slingerland and Alan Jay Smith
{slingn, smith}@cs.berkeley.edu

Computer Science Division
EECS Department
University of California at Berkeley

December 21, 2000

Abstract

Many microprocessor instruction sets include instructions for accelerating multimedia applications such as DVD playback, speech recognition and 3D graphics. Despite general agreement on the need to support this emerging workload, there are considerable differences between the instruction sets that have been designed to do so. In this paper we study the performance of five instruction sets on kernels extracted from a broad multimedia workload. Each kernel was recoded in the assembly language of the five multimedia extensions. We compare the performance of each extension against other architectures as well as to the original compiled C performance. From our analysis we determine how well multimedia workloads map to current architectures, what was useful and what was not. We also propose two enhancements to current architectures: strided memory operations, and superwide registers.

1 Introduction

Specialized instructions have been introduced by microprocessor vendors in order to support the specialized computational demands of multimedia applications. The mismatch between wide data paths and the relatively short data types found in multimedia applications has lead the industry to embrace SIMD (single instruction, multiple data) style processing. Unlike traditional forms of SIMD computing in which multiple individual processors execute the same instruction, multimedia instructions are executed by a single processor, and pack multiple short data elements into a single wide (64 or 128-bit) register, with all of the subelements being operated on in parallel.

The goal of this paper is to quantify how architectural differences between multimedia instruction sets translate into differences in performance. Prior studies have primarily focused on a single instruction set in isolation and have measured the performance on sets of kernels taken from vendor provided libraries [Nguy99], [Bhar98], [Allen99], [Chen96], [Rice96], [Naka96], [Rang99]. Our contribution is unique as we do not focus exclusively on a single architecture, and we study the performance of kernels derived from a real measured workload rather than those that have been established a priori. Our results are obtained from actual hardware measurements rather than through simulation, instilling confidence in our results.

Section 2 summarizes the multimedia workload we studied, and details the sixteen computationally important kernels which we extracted from it. Our methodology for recoding the kernels with multimedia instructions and their measurement is described in Section 3. An overview of the five instructions sets, and their implementations, is given in Section 4.

Our analysis is divided into two parts. Section 5 reflects our experience in coding the kernels, and gives insight into the useful and less than useful features of the multimedia instruction sets studied. In Section 6 we compare the performance of the five instruction sets both against one another, as well as their relative improvement over compiled (optimized) C code. Finally, in Section 7 we propose two new directions for multimedia architectures on general purpose microprocessors: strided memory operations and superwide registers.

2 Workload

The lack of a standardized multimedia benchmark has meant that workload selection is the most difficult aspect of any study of multimedia. It was for this reason that we developed the Berkeley multimedia workload [Sling00a]. In selecting the component applications we strove to cover as many types of media processing as possible: image compression (DjVu, JPEG), 3D graphics (Mesa, POVray), document rendering (Ghostscript), audio synthesis (Timidity), audio compression (ADPCM, LAME, mpg123), video compression (MPEG-2 at DVD and HDTV resolutions), speech synthesis (Rsynth), speech compression (GSM), speech recognition (Rasta) and video game (Doom) applications. Open source software was used both for its portability (allowing for cross platform com-

Funding for this research has been provided by the State of California under the MICRO program, and by Cisco Corporation, Fujitsu Microelectronics, IBM, Intel Corporation, Maxtor Corporation, Microsoft Corporation, Sun Microsystems, Toshiba Corporation and Veritas Software Corporation.

parisons) and the fact that we could analyze the source code directly.

In order to evaluate the various multimedia instruction sets, we hand coded kernels selected from the elements of the Berkeley multimedia workload. Those kernels were chosen based on their computational significance and their suitability for SIMD optimization. Table 1 lists the kernel codes examined. Both Mesa kernels appear to take up a relatively small amount of application CPU time, as software rendering (computing all stages of a rendering pipeline) was used in the portable version of these applications. It was for this reason that rasterization kernels were not included in the kernels studied due to the ubiquity of 3D accelerator cards which offload this from the CPU, and will continue to do so in the foreseeable future. Although we expect that when enough CPU cycles become available, much of the 3D rendering workload will be moved back onto the CPU and done in software (for cost savings), the current trend is moving in the opposite direction. First generation 3D accelerator cards took care of the rasterization stage, but not 3D geometry computations. Current 3D accelerator cards have also taken on the burden of geometry computations, indicating that the growth in complexity of 3D environments is outpacing that of CPU performance, despite the best efforts of multimedia extensions.

3 Methodology

3.1 Berkeley Multimedia Kernel Library

Our goal is to measure the performance of existing multimedia instruction sets on our set of important multimedia kernels. Our first step was to we distill from our Berkeley multimedia workload [Sling00a] a set of computationally important kernel functions, from which we formed the *Berkeley multimedia kernel library* (BMKL). All of the parent applications in the Berkeley multimedia workload were modified to make calls to BMKL rather than internal functions. From a performance standpoint, no piece of code can be realistically extracted and studied in isolation. By measuring a piece of code from within a real system we can realistically see how the shared resources of a computer system, such as CPU, caches, TLB, memory, and registers, affect the code and are affected by it.

Encapsulating the kernels within a library with a well defined interface allowed for: 1) low overhead measurement code to be placed around library functions in order to make measurements as non-invasive as possible, 2) different versions of the library to be quickly substituted in order to aid testing, 3) the addition of new architectures to our study by starting with a copy of the C reference library and then implementing and debugging replacement SIMD assembly functions one at a time.

3.2 Coding Process

As with DSPs, the most efficient way to program with multimedia extensions is to have an expert programmer tune software using assembly language [Kuro98]. Although this is more tedious and error prone than other methods such as hand coded standard shared libraries or automatic code generation by a compiler, it is a method that is available on every platform, and allows for great flexibility and precision when coding. We chose to code each kernel in assembly language ourselves rather than measure vendor supplied optimized libraries in order to prevent differences in programmer ability and time spent coding between the vendors' from potentially skewing our comparison.

All of the assembly codes in our study were written by the same programmer (Slingerland) with an approximately equal amount of time spent on each platform. Programming tools for cycle-accurate instruction scheduling through simulation were not used in order to equalize differences between the tools available on each platform. Instructions were scheduled sanely by keeping data consumption as far from data use as possible, unrolling loops when of performance benefit, and manually scheduling instructions so as to take advantage of multiple functional units.

For reasons of practicability, we limited our optimizations to the kernel level; we did not rewrite entire applications. This had ramifications for data alignment and data structure layout on some architectures. In some cases it was not practical to code a SIMD version of a kernel if an instruction set lacked the requisite functionality. For example, Sun's VIS and DEC's MVI do not support partitioned floating point, so floating point kernels were not recoded for these platforms. DEC's MVI also does not contain any data communication (e.g. permute, mix, merge) or partitioned integer multiply instructions. If there was no compelling opportunity for performance gain, kernels were not recoded from C. The C version was considered to be a particular platform's "solution" to a kernel when the needed SIMD operations were not provided. It might be supposed that hand coding would be superior to compiler generated code anyway, even without SIMD instructions. Although this may have been true at one time when instruction set architectures were designed with assembly language programmers in mind, modern instruction sets are targeted at compilers [Lawl92], [Patt96].

3.3 Measurement

Performance Monitoring Counters All of the microprocessors studied include performance monitoring counters to allow for interesting architectural events to be counted in real time during program execution. Although performance counters were sometimes used to guide our optimizations, their primary purpose was to be nearly cycle-accurate timers with which to measure the very short execution times of the kernels in the BMKL.

C Compilers and Optimization Flags The most architecturally tuned compiler on each architecture was used to compile the C reference version of the Berkeley Multimedia Kernel library. The optimization flags used were those which give the best general speedup and highest level of optimization without resorting to tuning the compiler flags to a particu-

Kernel Name	Source Application	Data Type	Sat Arith	Native Width	Src Lines	Satic Instr	%Static Instr	% CPU Cycles
Add Block	MPEG-2 Decode	8-bit (U)	√	64-bits	46	191	0.1%	13.7%
Block Match	MPEG-2 Encode	8-bit (U)		128-bits	52	294	0.4%	59.8%
Clip Test and Project	Mesa*	FP		limited	95	447	0.1%	0.8%
Color Space Conversion	JPEG Encode	8-bit (U)		limited	22	78	0.1%	9.8%
DCT	MPEG-2 Encode	16-bit (S)		128-bits	14	116	0.1%	12.3%
FFT	LAME*	FP		limited	208	981	4.4%	14.5%
Inverse DCT	MPEG-2 Decode	16-bit (S)	√	128-bits	75	649	0.3%	29.7%
Max Value	LAME*	32-bit (S)		unlimited	8	39	0.2%	12.0%
Mix	Timidity	16-bit (S)	√	unlimited	143	829	1.0%	35.7%
Quantize	LAME*	FP		unlimited	55	312	1.4%	15.3%
Short Term Analysis Filter	GSM Encode	16-bit (S)	√	128-bits	15	79	0.1%	20.2%
Short Term Synthesis Filter	GSM Decode	16-bit (S)	√	128-bits	15	114	0.2%	72.7%
Subsample Horizontal	MPEG-2 Encode	8-bit (U)	√	88-bits	35	244	0.3%	2.6%
Subsample Vertical	MPEG-2 Encode	8-bit (U)	√	unlimited	76	478	0.6%	2.1%
Synthesis Filtering	mpg123*	FP	√	512-bits	67	348	0.4%	39.6%
Transform and Normalize	Mesa*	FP		limited	51	354	0.1%	0.7%

Table 1: **Multimedia Kernels Studied** - from left to right, the colums list 1) primary data type specified as N-bit ({**U**nsigned,**S**igned}) integer or floating point (FP), 2) if saturating arithmetic is used, 3) native width; the longest width which does not load/store/compute excess unused elements, 4) static C source line count (for those lines which are executed), 5) percentage of total static instructions, 6) percentage of total CPU time spent in the kernel. The later three statistics are machine specific and are for the original C code on a Compaq DS20 (dual 500 MHz Alpha 21264, Tru64 Unix v5.0 Rev. 910) machine, compiled with GCC v2.8.1 (*) or DEC C v5.6-075.

lar kernel. The specific compiler and associated optimization flags used on each platform are listed given in the Appendix.

4 Instruction Sets

In this paper we study five architectures with different multimedia instruction sets (Table 2 lists the parameters for the exact parts we used). Most of the instruction sets (AMD'S 3DNow!, DEC's MVI, Intel's MMX, Sun's VIS) use 64-bit wide registers, while Motorola's AltiVec and Intel's SSE are 128-bits wide. The size of the register file available on each architecture varied widely, ranging from 8 64-bit registers on the x86 based AMD Athlon and Intel Pentium III to 32 128-bit wide registers with Motorola's AltiVec extension.

All of the multimedia extensions support integer operations, although the types and widths of available operations vary greatly. The earliest multimedia instruction sets (e.g. Sun's VIS and DEC's MVI) had design goals limited by the immaturity of their target workload and unproven benefit to performance. Because of this, any approach which greatly modified the overall architecture or significantly affected die size was out of the question. Leveraging as much functionality as possible from existing chip architectures was a high priority. Thus, the Sun and DEC extensions do not implement partitioned floating point instructions. Witness also the difference between Intel's first multimedia extension, MMX (1997), which had as one of its primary design goals to not require any new operating system support, and Intel's SSE (1999) which added new operating system maintained state (8 128-bit wide registers) for the first time since the introduction of the Intel386 instruction set (1985).

The processors studied vary in terms of instruction latency as well as the throughput per cycle. Processor clock rates were all 500 MHz, with the exception of the Sun UltraSPARC IIi, for which only a 360 MHz system was available. Although multimedia extensions primarily focus on extracting data level parallelism, most modern microprocessors are also superscalar, and thereby allow for multiple multimedia instructions to be issued every cycle. All of the architectures we looked at are fully pipelined, so barring any data dependencies, one new SIMD instruction can begin per functional unit each clock cycle. Typically, there are separate functional units for SIMD integer and SIMD floating point processing, although on the x86 architectures they are combined. [Sling00c] surveys existing multimedia instruction sets in more detail.

Sun Sun's UltraSPARC IIi processor incorporates the VIS multimedia extension, which implements a set of SIMD integer instructions that share the existing UltraSPARC floating point register file. Partitioned multiplication is done through 8-bit multiplication primitive instructions. A graphics status register (GSR) is used to support data alignment and scaling for pack operations.

Intel Intel's Pentium III processor includes their original SIMD integer MMX extension, as well as the newer SIMD floating point SSE instruction set. MMX is a 64-bit wide SIMD integer extension, which is mapped onto the existing x87 floating point architecture and registers and introduces no new architectural state (registers or exceptions). SSE is a

	AMD Athlon	DEC Alpha 21264	Intel Pentium III	Motorola G4	Sun UltraSPARC IIi
Clock [Current] (MHz)	500 [1000]	500 [667]	500 [1000]	500 [500]	360 [480]
SIMD Extension(s)	MMX/3DNow!+	MVI	MMX/SSE	AltiVec	VIS
SIMD Instructions	57/24	13	57/70	162	121
First Shipped	June 1999	February 1998	February 1999	August 1999	November 1998
Transistors ($x10^6$)	22.0	15.2	9.5	6.5	5.8
Process (μm) [Die Size (mm^2)]	0.25 [184.0]	0.25 [225.0]	0.20 [104.6]	0.20 [83.0]	0.25 [147.5]
L1 $I Cache, $D Cache (Kbytes)	64, 64	64, 64	16, 16	32, 32	32, 64
L2 Cache (Mbytes)	0.5	4	0.5	1	2
Register File (# x width)	FP(8x64b)/FP(8x64b)	Int (31x64b)	FP(8x64b)/8x128b	32x128b	FP(32x64b)
Reorder Buffer Entries	72	35	40	6	12
Int, FP Multimedia Units	2 (combined)	2, 0	2 (combined)	3, 1	2, 0
Int Add Latency	2 [64b/cycle]	-	1 [64b/cycle]	1 [128b/cycle]	1 [64b/cycle]
Int Multiply Latency	3 [64b/cycle]	-	3 [64b/cycle]	3 [128b/cycle]	3 [64b/cycle]
FP Add Latency	4 [64b/cycle]	-	4 [64b/cycle]	4 [128b/cycle]	-
FP Multiply Latency	4 [64b/cycle]	-	5 [64b/cycle]	4 [128b/cycle]	-
FP ~1/sqrt Latency	3 [32b/cycle]	-	2 [64b/cycle]	4 [128b/cycle]	-

Table 2: **Microprocessors Studied** - All parameters are for the actual chips used in this study. Clock lists the speed for the specific part studied as well as the current (December 2000) maximum shipping clock speed. A SIMD register files may be shared with existing integer (Int) or floating point (FP) registers, or be separate. Note that some of the processors implement several multimedia extensions (e.g. Pentium III has MMX and SSE) - the corresponding parameters for each are separated with "/". A dash (-) indicates that a particular operation is not available on a given architecture, and so no latency and throughput numbers are given. DEC's MVI extension (on the 21264) does not include any of the listed operations, but all MVI instructions have a latency of 3 cycles [64b/cycle].[Burd] [Noer] [AMD99] [AMD00] [AMDWP] [Carl97] [Comp00] [Intel97] [Intel99a] [Kesh99] [Kohn95] [Moto00] [Norm98] [Sun97]

follow on to MMX which is primarily a SIMD floating point extension but also incorporates feedback on MMX from software vendors in the form of new integer instructions. Unlike MMX, the floating point side of SSE *does* add new architectural state to the Intel architecture with the addition of an 8 x 128-bit register file and exceptions to support IEEE compliant floating point operations. Although the SSE instruction set architecture and register file are defined to be 128-bits wide, the Pentium-III SSE execution units are actually 64-bits (two single precision floating point elements) wide in hardware. The instruction decoder translates 4-wide (128-bit) SSE instructions into pairs of 2-wide (64-bit) internal micro-ops.

AMD The AMD Athlon processor implements MMX (which was licensed from Intel), in addition to AMD's own 3DNow! extension which utilizes the same x87 floating point registers and basic instruction formats as MMX, but adds a partitioned single precision floating point data type. The Athlon processor actually extends 3DNow! with Enhanced 3DNow! that adds floating point and integer operations to make 3DNow! functionally equivalent to Intel's SSE extension.

DEC The DEC (now Compaq, but we will refer to it as DEC for historical consistency) Alpha 21264A processor includes the SIMD integer Motion Video Instructions (MVI) multimedia extension. It is the smallest of the multimedia instruction sets, weighing in with only 13 instructions. MVI shares the existing Alpha 32-register integer register file. Notably, no SIMD saturating addition/subtraction, multiplication, or shift instructions are included.

Motorola Motorola's MPC7400 (also known as the G4) processor utilizes their 128-bit wide SIMD AltiVec extension which supports a wide variety of integer data types, as well as partitioned single precision floating point. A dedicated 32 x 128-bit register file is implemented, along with four non-identical parallel, pipelined vector execution units. Hardware assisted software prefetching is implemented, where by a prefetch stream is set up by software, and fetched into the cache independently by hardware.

5 Analysis

In the next section we use our experience coding each of the sixteen kernels with five different multimedia extensions to determine: 1) existing architectural features that are useful, 2) features that have been implemented, but don't appear to be useful, and 3) significant bottlenecks in current multimedia architectures. Illustrating our discussion are code fragments both from the original C source code of each kernel algorithm, as well as the different SIMD implementations. The code fragments consist of a few of the key central lines of code from a given kernel. This gives an idea about the types of operations and data types used. The data types of all of the variables in our sample C code are specified in a platform independent way such that the prefix indicates the type: INT: signed integer, UINT: unsigned integer, FP: floating point, followed by

N, the number of bits. The complete original C source code for each kernel can be found in Appendix B. Source code for the SIMD implementations of the BMKL are available on the web at *http://www.cs.berkeley.edu/~slingn/research/*.

5.1 Register File and Data Path

Multimedia instruction sets can be broadly categorized according to the location and geometry of the register file upon which SIMD instructions operate. Alternatives include the reuse of the existing integer or floating point register files, or implementing an entirely separate one. The type of register file affects the width and therefore the number of packed elements that can be operated on simultaneously (vector length).

Integer Data Path Implementing multimedia instructions on the integer data path has the advantage that the functional units for shift and logical operations need not be replicated. Partitioned addition and subtraction are easily created by blocking the appropriate carry bits. Modifications to the integer data path to accommodate multimedia instructions can potentially adversely affect the critical path of the integer data-path pipeline [Kuro98]. On the x86 (AMD, Intel) and PowerPC (Motorola) architectures the integer data path is 32-bits wide, making a shared integer data path approach less compelling due to the limited amount of data level parallelism possible.

Floating Point Data Path The reuse of floating point rather than integer registers has the advantage of not being shared with pointers and loop and other control flow variables. In addition, multimedia and floating point instructions are not typically used simultaneously [Kuro98]. All of the architectures examined have floating point data paths which support at least double-precision (64-bit wide) operations, which for many architectures is wider than the integer data path.

Separate Data Path A separate data path has the advantage of simplifying pipeline control and increasing the overall number of registers. Disadvantages include the need for saving and restoring the new registers on context switches, as well as the relative difficulty and high overhead of moving data between register files.

SIMD Register Width Perhaps one of the most critical factors in SIMD instruction set design is deciding how long vectors will be. If an existing data path is to be reused, there is little choice, but when a new data path is to be designed it makes sense to ask how wide is wide enough. Too short of a vector length limits the ability to exploit data parallelism, while an excessively long vector length can degrade performance and increase the amount of clean up code overhead. The "native width" column of Table 1 specifies how each of the multimedia kernels fits into one of the following three categories:

1. unlimited width - Kernels that operate on data elements which are truly independent and are naturally arranged so as to be amenable to SIMD processing. The inner loops of these kernels can be strip mined at almost any width, with the increase in performance of a longer vector being directly proportional to the increase in vector length.
2. limited width - Although data elements are independent, there is overhead involved in rearranging input data so that it may be operated on in a SIMD manner with longer vectors. Thus, the performance advantage of longer vectors is limited by the overhead (which typically increases with vector length) required to employ them.
3. exact width - A kernel which has a precise natural width which can be considered to be the right match for the kernel. This width is the longest width which does not load, store, or compute excess unused elements.

Long vectors can be a problem when their length exceeds the natural width of an algorithm. A good example of this problem is the add block kernel, which operates on MPEG subblocks (8 x 8 arrays of pixels). One input array (`bp`) consists of signed 16-bit integer values and the other (`rfp`) of unsigned 8-bit integer (Algorithm 1).

Algorithm 1 Add Block - computed for each pixel in a subblock

```
INT16 *bp; UINT8 *rfp; INT32 tmp;
tmp = *bp++ + *rfp; /* Add */
*rfp = tmp>255 ?  255 :  (tmp<0 ?  0 :  tmp); /* Clip */
```

During the block reconstruction phase of motion compensation in the decoder, a block of pixels is reconstituted by summing the pixels in different subblocks. Consider the AltiVec implementation of the add block kernel (Algorithm 2). Motorola's AltiVec is the only SIMD integer extension examined which is 128-bits wide; Intel's SSE is only 128-bits wide for packed floating point operations. Each time a row of the 8x8 `rfp` subblock is loaded on a 128-bit wide architecture, one half of the vector is useless data which will be thrown away when the `rfp` values are expanded to 16-bits.

Algorithm 2 AltiVec Add Block Fragment

```
;; unaligned vector load
lvx     v3, 0, r3        ; v3: vector MSQ for initial bp0..bp7 vector
lvx     v4, r11, r3      ; v4: vector LSQ
;; unaligned vector load
lvx     v0, 0, r4        ; v0: vector MSQ
lvx     v1, r11, r4      ; v1: vector LSQ
lvsl    v2, 0, r4        ; v2: vector alignment mask for vperm
vxor    v10, v10, v10    ; v10: 0
vperm   v0, v0, v1, v2   ; v0: rfp:|0|1|2|3|4|5|6|7|X|X|X|X|X|X|X|X|
vperm   v3, v3, v4, v5   ; v3: | bp0 | bp1 | bp2 | bp3 | bp4 | bp5 |
bp6 | bp7 |
addi    r3, r3, 16       ; r3: bp += 8 (pointer to INT16)
vmrghb  v1, v10, v0      ; v0: | rfp0| rfp1| rfp2| rfp3| rfp4| rfp5|
rfp6| rfp7|
vaddshs v1, v3, v1       ; v1: bp + rfp [0..7]
vpkshus v1, v1, v1       ; v1: rfp:|0|1|2|3|4|5|6|7|0|1|2|3|4|5|6|7|
stvewx  v1, 0, r4        ; store rfp [0..3]
stvewx  v1, r12, r4      ; store rfp [4..7]
add     r4, r4, r5       ; r4: rfp += (iincr + 8) (pointer to UINT8)
vmov    v3, v4           ; move current LSQ to next MSQ
```

As we saw in Table 1, most multimedia kernels are either unlimited/limited or have most often have an exact required

width of 128-bits. The remaining exact native widths (64-, 88- and 512-bits) came up only once each. Thus, we consider a total vector length of 128-bits to be best.

Number of Registers Multimedia applications (and their kernels) can generally take advantage of quite large register files. Not coincidentally, MicroUnity's dedicated media processor chip has a 64 x 64-bit register file, which can also be accessed as 128 x 32-bits [Hans96], while the Philips Trimedia TM-1 has a 128 x 32-bit register file [Rath96].

As an example of where large numbers of registers are useful in our workload, consider the DCT and IDCT kernels (fragments of the original codes are given in Algorithms 3 and 4). The discrete cosine transform (DCT) is the algorithmic centerpiece to many lossy image compression methods. It is similar to the discrete Fourier transform (DFT) in that it maps values from the time domain to the frequency domain, producing an array of coefficients representing frequencies [Kien99]. The inverse DCT maps in the opposite direction, from the frequency to the time domain.

Algorithm 3 DCT

```
extern FLOAT64 c[8][8]; /* transform coefficients */
INT16 block[8][8]; FLOAT64 sum; FLOAT64 tmp[8][8];
  for (INT32 i=0; i<8; i++)
    for (int j=0; j<8; j++) {
      sum = 0.0;
      for (int k=0; k<8; k++)
        sum += c[j][k] * block[i][k];
      tmp[i][j] = sum;
    }
```

Algorithm 4 Inverse DCT - only row computation shown

```
INT32 x0,x1,x2,x3,x4,x5,x6,x7,x8
  x7 = x8 + x3;
  x8 -= x3;
  x3 = x0 + x2;
  x0 -= x2;
  x2 = (181*(x4+x5)+128)>>8;
  x4 = (181*(x4-x5)+128)>>8;
```

A 2D DCT or IDCT is efficiently computed as 1D transforms on each row followed by 1D transforms on each column and then scaling appropriately. A SIMD approach requires that multiple data elements from several iterations be operated on in parallel for the greatest efficiency. This is straightforward for the 1D column DCT, since the corresponding elements of each loop iteration are adjacent in memory (assuming a row-major storage format). A 1D row DCT is more problematic since the corresponding elements of adjacent rows are not. A matrix transposition (making corresponding "row" elements adjacent in memory), then performing the desired computation, and transposing the matrix back again (to put the resulting data back in the correct configuration) can be an effective way to compute the 1D row step of a 2D transform. However, this was only of performance benefit for those architectures whose register files were able to hold the entire 16-bit 8x8 matrix at once. Since the DCT and IDCT both operate on 8x8 2D matrices of 16-bit signed values, they require at least 16 64-bit registers, or 8 128-bit registers.

5.2 Data Types

The data types supported by the different multimedia instruction sets include {signed, unsigned}{8, 16, 32, 64} bit values, as well as single precision floating point. Most of the instruction sets do not support all of these types, and usually only a subset of operations on each. In order to determine which data types and operations are useful, we broke down the dynamic SIMD instruction counts on each architecture in two ways: 1) data type distribution per instruction class (e.g. add, multiply) and 2) data type distribution per kernel. Tables of these categorizations are available in the Appendix.

In general, the video and imaging kernels (add block, block match, color space, DCT, IDCT, subsample horizontal, subsample vertical) utilize 8- and 16-bit operations. Audio kernels (FFT, max val, mix stereo, quantize, short term analysis filtering, short term synthesis filtering, synthesis filtering) either use 16-bit values or floating point, while the 3D kernels (clip test, transform) are limited almost exclusively to floating point.

Integer Although image and video data is typically stored as packed unsigned 8-bit values, intermediate processing usually requires precision greater than 8-bits. Other than for width promotion, most 8-bit functionality is wasted on our set of multimedia kernels. In general, *storage data types* (how data is stored in memory or on disk), although narrow and therefore potentially offering the greatest degrees of parallelism, are simply too narrow for intermediate computations to occur without overflow. A few operations inherently produce results that can not overflow the input data type. For example, although an N-bit average operation internally utilizes $N+1$ bits of precision to sum its two operands, the result is rounded and shifted back to N-bits before being stored to a register. Other operations such as the sum of absolute differences (SAD) produce a scalar result which fits in a destination register of the same width as the packed operands or a scalar register.

The signed 16-bit data type is the most heavily used because it is both the native data type for audio and speech data, as well as the typical intermediate data type for video and imaging. On the wide end of the spectrum, 32-bit and longer data types are typically only used for accumulation and simple data communication operations such as alignment. Operations tend to be limited to addition and subtraction (for accumulation), width promotion and demotion (for converting to a narrower output data type) and shifts (for data alignment).

Floating Point Single precision floating point plays an important role in many of the multimedia kernels such as the geometry stage of 3D rendering (the clipping and transform kernels) and the FFT, where the wide dynamic range of floating point is required. Only Intel's recently announced SSE2 extension will offer a packed double precision data type, to be targeted at applications other than multimedia such as scientific and engineering workloads, as well as advanced 3D geometry such as is used in raytracing [Intel00a], [Intel00b].

5.3 Operations

One of our primary goals is to separate useful SIMD operations from those that are not. The large differences in current multimedia instruction sets for general purpose processors are fertile ground for making such a determination because many different design choices have been made. In the Appendix we provide a table of SIMD instruction set functionality broken down per kernel, the important points of which we discuss here. Our analysis assumes that SIMD extensions are targeted solely at the domain of multimedia applications. In some cases, the targeted applications during the design of a multimedia extension included DSP applications and others which are not reflected in the Berkeley multimedia workload.

5.3.1 Arithmetic

Modulo/Saturation *Modulo arithmetic* "wraps around" to the next representable value when overflow occurs, while *saturating arithmetic* clamps the output value to the highest or lowest representable value for positive and negative overflow respectively. Saturating arithmetic is useful both because of its desirable aesthetic result in pixel based computations (video and imaging) as well as the fact that it allows for overflow in multiple packed elements to be dealt with efficiently. When adding pixels, modulo addition is undesirable since if overflow occurs a small change in operand values may result in a glaring visual difference (e.g. adding two white pixels results in a black pixel). If overflow within packed elements were to be handled similar to traditional scalar arithmetic, an overflowing SIMD operation would have to be repeated and tested serially to determine which element overflowed. The added cost of saturating arithmetic is that unlike modulo operations, for which the same instruction works for both unsigned and signed (2's complement) values, saturating arithmetic necessarily requires separate instructions since the values must be interpreted by the hardware as a particular data type.

Modulo computations are important because they allow for the results of SIMD optimized codes to be numerically identical to existing scalar algorithms. This is sometimes an important consideration for the sake of compatibility and comparability. The kernels in the BMKL which employ saturating arithmetic are noted in Table 1. From this it is clear that the most important types for saturating arithmetic are unsigned 8-bit and signed 16-bit integers. The IDCT kernel clamps to a signed 9-bit range [-256..+255], which can be accomplished through a pair of max/min operations; we discuss these in more detail later. Saturating 32-bit operations are of little value since overflow is usually not a concern for such a wide data type.

Shift SIMD shift operations are extremely important for supporting fixed point integer arithmetic. A common sequence of operations is the multiplication of an N-bit integer by an M-bit fixed point fractional constant, producing an $(N+M)$-bit result, with a binary point at the M^{th} most significant bit. At the end of computation, the final result is rounded by adding the fixed point fraction representing $\frac{1}{2}$, and then shifting the sum right M-bits to eliminate the fractional bits. Shifts are important operations for all data types, and are critical for fixed point integer arithmetic, as well as providing an inexpensive way to perform multiplication and division by powers of two.

Min/Max Min and max output the minimum or maximum values of the corresponding elements in two partitioned input registers, respectively. A max instruction is clearly useful in the maximum value search kernel, which searches through an array of signed 32-bit integers for the greatest maximum absolute value in the array. Max and min instructions have other less obvious uses as well. Signed minimum and maximum operations are often used with a constant second operand to saturate results to arbitrary ranges. The IDCT kernel clips its output value range to -256...+255 (9-bit signed integer), which does not correspond to the data types supported by any of the multimedia extensions. Algorithm 5 demonstrates clamping to arbitrary boundaries for the Intel implementation of the IDCT.

Algorithm 5 Intel MMX/SSE IDCT

```
pmaxsw  mm0, [CLIP_MIN] ;; compute first element
pminsw  mm0, [CLIP_MAX] ;; in order to free mm0
movq    [esi + 0], mm0  ; store x0 [0..3]
movq    mm0, [CLIP_MIN]  ; clip to -256
pmaxsw  mm1, mm0
pmaxsw  mm2, mm0
pmaxsw  mm3, mm0
pmaxsw  mm4, mm0
pmaxsw  mm5, mm0
pmaxsw  mm6, mm0
pmaxsw  mm7, mm0
movq    mm0, [CLIP_MAX]  ; clip to +255
pminsw  mm1, mm0
pminsw  mm2, mm0
pminsw  mm3, mm0
pminsw  mm4, mm0
pminsw  mm5, mm0
pminsw  mm6, mm0
pminsw  mm7, mm0
```

Although max and min can be synthesized through simpler operations (operations which are useful in their own right), the additional execution cost is simply too great to be practical. Rather than a single independent instruction, three dependent instructions are required. An arbitrary clamping operation can also be simulated with packed signed saturating addition. The representable range of a signed fixed point number (a bits to the left of the binary point, b bits to the right) is $-2^a \leq x \leq 2^a - 2^{-b}$. For example, if we need to limit a value, X, to the range $-j..+k$:

1. $(2^a - 2^b) - k \longrightarrow T_{pos}$
2. $X + T_{pos} \Longrightarrow X$
3. $X - T_{pos} \longrightarrow X$

These three steps limit X to $+K$. ($\Longrightarrow$ represents saturating overflow, where as a $\longrightarrow$ symbolizes modulo overflow.) Three more operations are required to limit X to the desired floor value:

1. $-2^a + j \longrightarrow T_{neg}$

2. $X + T_{neg} \Longrightarrow X$

3. $X - T_{neg} \longrightarrow X$

Architecturally, the implementation cost of max and min instructions should be low since the necessary comparators must already exist for saturating arithmetic. The only difference is that instead of comparing to a constant, a register value is used instead. An added advantage that we have found is that in many cases where comparisons are required, max and min instructions are sufficient.

Comparisons We have found that integer control flow instructions (e.g. packed comparisons) are seldom needed, except on architectures without max/min operations (e.g. Sun's VIS). We found one instance where a specialized floating point comparison was useful. In the project and clip test kernel, 3D objects are first mapped to 2D space through a matrix multiplication of 1x4 vectors and 4x4 matrices. Objects are then clipped to the viewable area to avoid unnecessary rendering. The code fragment listed in Algorithm 6 is computed for each vertex in a 3D scene every time a frame is rendered.

Algorithm 6 Clip Test and Project

```
FLOAT32 ex = vEye[i][0], ey = vEye[i][1], ez = vEye[i][2], ew =
vEye[i][3];
FLOAT32 cx = m0 * ex + m8 * ez, cy = m5 * ey + m9 * ez;
FLOAT32 cz = m10 * ez + m14 * ew, cw = -ez;
UINT8 mask = 0;
vClip[i][0] = cx; vClip[i][1] = cy; vClip[i][2] = cz; vClip[i][3] = cw;
if (cx > cw) mask |= CLIP_RIGHT_BIT;
else if (cx < -cw) mask |= CLIP_LEFT_BIT;
if (cy > cw) mask |= CLIP_TOP_BIT;
else if (cy < -cw) mask |= CLIP_BOTTOM_BIT;
if (cz > cw) mask |= CLIP_FAR_BIT;
else if (cz < -cw) mask |= CLIP_NEAR_BIT;
```

Motorola's AltiVec includes a specialized comparison instruction, `vcmpbfp`, which deals with boundary testing. This is done by testing all of the clip values in parallel to see if any clipping is needed, and branching to act as a fast out if no clipping is necessary. This technique is extremely effective because no clipping is the common case, with most vertices within the screen boundaries. An example of this is shown in the clip test kernel from Mesa's 3D rendering pipeline (Algorithm 6). For the Mesa "gears" application, the fast out case held true for 61946 of the 64560 (96.0%) clipping tests performed in our application run of 30 frames rendered at 1024x768 resolution.

Sum of Absolute Differences A sum of absolute differences (SAD) instruction operates on a pair of packed 8-bit unsigned input registers, summing the absolute differences between the respective packed elements in two registers and placing (or accumulating) the scalar sum in another register. The block match kernel is the only one in which sum of absolute differences (SAD) instructions is used. Algorithm 8 lists the core lines of the block match kernel utilized by MPEG-2 encoding. Block match sums the absolute differences between the corresponding pixels of two 16x16 macroblocks. The original application code also includes three other variations on block match which compute horizontal, vertical or both horizontal and vertical interpolation before calculating the sum of absolute differences.

Algorithm 7 Motorola G4 Clip Test and Project

```
;;                      ; v1:  | cx | cy | cz | cw |
vspltw   v2, v1, 3      ; v2:  | cw | cw | cw | cw |
vcmpbfp. v3, v1, v2     ; v3:  bit mask of clip comparisons
;; set cr6 to 0x2 if all test values are within boundaries
li       r12, 0
mcrf     cr0, cr6
bc       COND_TRUE, 0x2, fast_out
vcmpgtsw v4, v0, v3     ; v4:  > test, - mask if clipping
vcmpgtsw v5, v3, v0     ; v5:  < test, + mask if clipping
vand     v4, v4, v27
vand     v5, v5, v28
vor      v4, v4, v5     ; v4:  | mask0| mask1| mask2| mask3|
vsldoi   v5, v4, v4, 8
vor      v4, v4, v5
vsldoi   v5, v4, v4, 4
vor      v4, v4, v5     ; v4:  | mask | mask | mask | mask |
vspltb   v4, v4, 15     ; v4:  | M| M|...| M| M| [0..15]
stvebx   v4, 0, r5      ; store mask to mask_temp
lbz      r12, 0(r5)     ; r12:  mask
lbz      r15, 0(r7)     ; r15:  clipMask[i]
or       r15, r15, r12
stb      r15, 0(r7)     ; r15:  clipMask[i] |= mask
or       r13, r13, r12 ; r13:  tmpOrMask |= mask
fast_out:
addi     r7, r7, 1      ; r7:  clipMask++ (pointer to UINT8)
and      r14, r14, r12 ; r14:  tmpAndMask &= mask
addic.   r3, r3, -1     ; n--
bc COND_FALSE, ZERO_RESULT, loop
```

Algorithm 8 Block Match

```
UINT8 blk_1[16][16]; UINT8 blk_2[16][16]; INT32 sad=0; INT32 diff;
for(j=0; j<h; j++)
 for(i=0; i<16; i++) {
  if ((diff = blk_1[j][i] - blk_2[j][i])<0)
    diff = -diff;
  sad+=diff;
 }
```

Although DEC's MVI extension is quite small (only 13 instructions), one of the few operations that DEC did include was SAD. DEC architects found (in agreement with our experience) that this operation provides the most performance benefit of all multimedia extension operations [Rubi96]. Intel's MMX, although a much richer set of instructions, did not include this operation (it *was* later included in both AMD's 3DNow!+ and Intel's SSE extensions to MMX). Sun's VIS also includes a sum of absolute differences instruction.

The Motorola G4 microprocessor was the only CPU in our survey which did not include some form of SAD operation, forcing us to synthesize the SAD operation from other instructions (Algorithm 9). Although Intel's SSE extension (see Algorithm 10) includes the `psadbw` instruction, this offers only a one cycle performance advantage when compared to the AltiVec implementation. In some ways this comparison is misleading since Intel's extension is 64-bits wide, while Motorola's is 128-bits; the question of performance should be absolute, not relative to Intel. In order to estimate the latency of a hypothetical SAD instruction for a 128-bit extension such as AltiVec, we examine the latency of this instruction on the other (64-bit) architectures:

Processor	Instruction	Latency:Throughput
Intel Pentium III	PSADBW	5 cycles : 1 every 2 cycles
AMD Athlon	PSADBW	3 cycles : 1 every 1 cycle
Sun UltraSPARC IIi	PDIST	3 cycles : 1 every 1 cycle
DEC Alpha 21264	PERR	2 cycles : 1 every 1 cycle

The latency of a 128-bit instruction would be higher than the 64-bit instructions listed in the table because this instruction requires a cascade of adders to sum (reduce) the differences between the elements. An N-bit SAD instruction ($M = N/8$) can be broken down into steps: 1) calculate M 8-bit differences, 2) calculate the absolute value of the differences, 3) perform $log_2 M$ cascaded summations. The architects of the 64-bit DEC MVI extension comment that a 3-cycle implementation of PERR would have been easily achievable, but in the end the architects achieved a more aggressive 2-cycle instruction [Carl97]. If a SAD operation were to be implemented in Motorola's AltiVec, we estimate it would have a latency of 4 cycles. This would certainly be a superior solution compared to the 9 cycle solution shown in Algorithm 9.

Algorithm 9 Motorola G4 Block Match - SAD portion (starting with `vmaxub` instruction) takes 9 cycles

```
;; v1:  block #1, line #1, pixels [0..F]
;; v4:  block #1, line #2, pixels [0..F]
;; v6:  block #2, line #1, pixels [0..F]
vavgub v1, v1, v4     ; v1:  vertically interpolated p0..pF
vmaxub v7, v1, v4     ; v7:  max [0..F]
vminub v8, v1, v4     ; v8:  min [0..F]
vsububs v7, v7, v8    ; v7:  abs_diffs [0..F]
vsum4ubs v31, v7, v31 ; v31:  | SAD_0 | SAD_1 | SAD_2 | SAD_3 |
vsumsws v31, v31, v0  ; v31:  |   0   |   0   |   0   |  SAD  |
```

Algorithm 10 Intel Pentium III Block Match - SAD portion (starting with first `psadbw` instruction) takes 8 cycles

```
;; mm1:  block #1, line #1, pixels [0..7]
;; mm5:  block #1, line #1, pixels [8..F]
;; mm3:  block #1, line #2, pixels [0..7]
;; mm4:  block #1, line #2, pixels [8..F]
;; mm2:  block #2, line #1, pixels [0..7]
;; mm6:  block #2, line #1, pixels [8..F]
pavgb mm1, mm3     ; mm1:  vertically interpolated p0..p7
pavgb mm5, mm4     ; mm5:  vertically interpolated p8..pF
psadbw mm1, mm2    ; mm1:  | 0 | 0 | 0 |SAD0| Pixels 0..7
psadbw mm5, mm6    ; mm5:  | 0 | 0 | 0 |SAD1| Pixels 8..F
paddd mm1, mm5     ; mm1:  | 0 | 0 | 0 | SAD|
```

Average. In addition to compute the sum of absolute differences, half-pixel interpolation, for which MPEG-2 encoding offers three varieties, is also important; vertical interpolation is shown in Algorithms 9 and 10. Interpolation is done by averaging a set of pixel values with pixels offset by one horizontally, vertically or both. The original C MPEG-2 code first performs the interpolation, and then computes the sum of absolute differences on the result. SIMD interpolation can be performed through 8-bit unsigned average instructions (again see Algorithms 9 and 10). DEC's MVI extension does not include a packed average instruction, but a similar interpolation operation can be done by averaging the result of several SAD operations using scalar integer arithmetic (since the result of a SAD instruction is a scalar value).

Integer average operations were only used in the block match kernel. This kernel operates on 8-bit unsigned values, so this is the only type of "average" instruction that was useful within our workload.

High Latency Function Approximation. Applications such as 3D rendering have kernels which use floating point mathematical functions, such as reciprocal and square-root, that are very high latency. On some architectures these scalar functions are computed in software, while others have hardware instructions. Full IEEE compliant operations return 24-bits of mantissa. The computation of these functions is iterative, so the number of bits of precision returned is directly proportional to an operation's latency. It is for this reason that all of the SIMD floating point extensions (AMD's 3DNow!, Intel's SSE and Motorola's AltiVec) include approximation instructions for $\frac{1}{x}$ and $\frac{1}{\sqrt{x}}$. These are typically implemented as hardware lookup tables, returning k-bits of precision. In Intel's SSE, for example, approximate reciprocal (`rcp`) and reciprocal square root (`rsqrt`) return 12-bits of mantissa. Motorola's AltiVec also returns 12-bits of precision for both the reciprocal and reciprocal square root approximation instructions.

In the transform and normalize kernel, graphics primitives are transformed to the viewer's frame of reference through matrix multiplications. The code shown in Algorithm 11 is computed for each vertex in a 3D scene.

Algorithm 11 Transform and Normalize

```
FLOAT64 tx, ty, tz, len, scale;
FLOAT32 ux = u[i][0],  uy = u[i][1],  uz = u[i][2];
tx = ux * m[0] + uy * m[1] + uz * m[2];
ty = ux * m[4] + uy * m[5] + uz * m[6];
tz = ux * m[8] + uy * m[9] + uz * m[10];
len = sqrt( tx*tx + ty*ty + tz*tz );
scale = (len>1E-30) ?  (1.0 / len) :  1.0;
v[i][0] = tx * scale; v[i][1] = ty * scale;
v[i][2] = tz * scale;
```

The transform kernel has at its heart a floating point reciprocal square root operation ($\frac{1}{\sqrt{x}}$). One unique aspect of Intel's SSE instruction set is that not only does it include 22-bit precise (mantissa) approximations of $\frac{1}{x}$ and $\frac{1}{\sqrt{x}}$, but it also includes full precision (24-bits of mantissa) versions of division and $\sqrt{x}$. Of course, this added precision comes at a price - namely much higher latency (their full precision instructions are not pipelined) than the pipelined 22-bit approximations derived from processor internal lookup tables. All of the floating point multimedia extension vendors, including Intel, point out the Newton-Raphson method for improving the accuracy of approximations through specially derived functions. In the case of $\frac{1}{\sqrt{x}}$ it is possible to iteratively increase the precision of an initial approximation through the equation:

$$x_1 = x_0 - (0.5 \cdot a \cdot x_0^3 - 0.5 \cdot x_0) = 0.5 \cdot x_0 \cdot (3.0 - a \cdot x_0^2) \qquad (1)$$

Employing an approximation instruction in conjunction with the Newton-Raphson method to achieve full precision is actually faster than the full precision version of the instruction that Intel provides. Compare the code fragments in Algorithms 12 and 13, which have execution times of 25 vs.

36 cycles. One iteration of the Newton-Raphson method is enough to improve a 22-bit approximation to the full 24-bit precision of IEEE single precision.

Algorithm 12 Intel Approximated Square Root - 25 cycles

```
; xmm3:  tx^2+ty^2+tz^2 [0..3] (len=sqrt(tx^2+ty^2+tz^2))
; xmm7:  0.5 [0..3]
; xmm5:  3.0 [0..3]
rsqrtps  xmm4, xmm3        ; xmm4:  rsqrtps(a)
movaps   xmm6, xmm3
mulps    xmm6, xmm4        ; xmm6:  a*rsqrtps(a)
mulps    xmm6, xmm4        ; xmm6:  a*rsqrtps(a)*rsqrtps(a)
mulps    xmm4, xmm7        ; xmm4:  0.5*rsqrtps(a)
subps    xmm5, xmm6        ; xmm5:  3.0 - a*rsqrtps(a)*rsqrtps(a)
mulps    xmm4, xmm5        ; xmm4:  |1/len1|1/len2|1/len1|1/len0|
;; sqrt(a) = a*(1/sqrt(a))
mulps    xmm3, xmm4        ; xmm3:  | len3 | len2 | len1 | len0 |
```

Algorithm 13 Intel Full Precision Square Root - 36 cycles

```
; xmm3:  tx + ty + tz [0..3] (len=sqrt(tx+ty+tz))
; xmm3:  a [0..3]
sqrtps   xmm3, xmm3        ; xmm3:  sqrt(a)
```

The added cost of the Newton-Raphson method is of the additional register space needed to hold intermediate values and constants. AMD's 3DNow! extension circumvents this by including instructions to internally perform the Newton-Raphson method, rather than having the programmer implement it (Algorithm 14). The only odd thing about AMD's reciprocal square root instructions are that they are actually scalar; they only produce one result value, based on the lower packed element.

Algorithm 14 AMD Approximated Square Root - 20 cycles

```
; mm3:  tx^2+ty^2+tz^2 [0] (len=sqrt(tx^2+ty^2+tz^2))
pfrsqrt         mm4, mm7     ; mm4: |~1/len0 |~1/len0 |
movq            mm5, mm4
pfmul           mm4, mm4
pfrsqit1        mm4, mm7
pfrcpit2        mm4, mm5     ; mm4: | 1/len0 | 1/len0 |
pfmul           mm7, mm4     ; mm7: |  len0  |  len0  |
```

5.3.2 Exceptions

Techniques for handling exceptions that occur during SIMD processing are very similar to those employed when dealing with packed overflow. Checking result flags or generating an exception from a packed operation requires considerable time to determine which packed element caused the problem. In most cases where an exception might be raised it is possible to fill in a value which will give reasonable results for most applications. This speeds execution because no error condition checking need be done, and is is similar to saturating integer arithmetic where maximum or minimum result values are substituted rather than checking for and reporting positive or negative overflow. Both AMD's 3DNow! and Motorola's AltiVec extensions do not implement IEEE compliant floating point exceptions. Only Intel's SSE implements full IEEE compliant SIMD floating point exceptions, and includes a control/status register (`MXCSR`) to mask or unmask packed floating point numerical exceptions.

5.3.3 Floating Point Rounding

Intel's SSE offers two modes of rounding: IEEE compliant and another, faster, flush to zero (FTZ) mode. Flush to zero (FTZ) clamps to a minimum representable result in the event of underflow (a number too small to be represented in single precision floating point). Fully compliant IEEE floating point supports four rounding modes. Most real time 3D applications use the FTZ rounding mode since they are not particularly sensitive to a slight loss in precision [Thak99]. 3DNow! supports only truncated rounding (round to zero). All of Motorola's AltiVec floating point arithmetic instructions use the IEEE default rounding mode of round to nearest. The IEEE directed rounding modes are not provided.

5.3.4 Type Conversion

Width promotion is the expansion of an N-bit value to some larger width. For unsigned fixed point numbers this requires zero extension or filling any additional bits with zeros. Zero extension is usually not specified as such in a multimedia architecture because it overlaps in functionality with data rearrangement instructions such as unpack or merge. If packed values are merged with another register which has been zeroed prior to merging the result is zero extension. Signed element unpacking is not as simple, but is rarely supported directly by hardware; only the AltiVec instruction set includes it. It can be synthesized with multiplication by one since a multiplication yields a result that is the overall width of both its operands.

Video and imaging algorithms use an 8-bit unsigned data type. Audio and speech algorithms, on the other hand, typically employ signed 16-bit values, but because multiplication by a fractional fixed point constant is a common operation, these values are often unpacked as a natural consequence of computation. So, although a signed unpack operation would likely be faster than multiplication by 1, it is seldom necessary to resort to this in practice.

All data types that occur in multimedia should be supported for packing and unpacking even for those widths not directly supported by arithmetic operations. It should always be possible to convert to a width that is supported for computation. Although we do not otherwise examine HP's MAX-1/MAX-2 extensions, as no hardware employing them was available to us at the time of this work, they are good examples of where not following this guidline can cause problems. We have noted the importance of the 16-bit data width. HP's MAX-1/MAX-2 instruction sets only support operations on 16-bit wide values. Partitioned 8-bit operations were considered, but rejected due to insufficient precision. Wider packed data types (e.g. 32-bit) were not included due to insufficient parallelism. What this approach overlooks is that fact that even though many intermediate computations require greater precision than 8-bits, many types of video and imaging data

are stored this way in existing multimedia file formats. Thus, packing and unpacking to and from 8-bit precision is a very common operation, which is not supported in hardware, making HP's extensions inefficient at processing this type of data.

5.3.5 Data Rearrangement

SIMD instructions perform the same operation on multiple data elements. Because all of the data within a register must be treated identically, the ability to efficiently rearrange data bytes within and between registers is critical for performance. We will refer to these types of operations as "data communication" instructions. *Interleave* instructions (also referred to as *mixing*, *unpacking* or *merging*) merge alternate data elements from the upper or lower half of the elements in each of two source registers. *Align* or *rotate* operations allow for arbitrary byte-boundary data realignment of the data in two source registers; essentially a shift operation that is done in multiples of 8-bits at a time. Both interleave and align type operations have hard coded data communication patterns. *Insert* and *extract* operations allow for a specific packed element to be extracted as a scalar or a scalar value to be inserted to a specified location. *Shuffle* (also called *permute*) operations allow greater flexibility than those operations with fixed communication patterns, but this added flexibility requires that the communication pattern be specified either in a third source register or as an immediate value in part of the instruction encoding.

The sufficiency of simpler data communication operations is to some degree dependent on the vector length employed. For example, 128-bit AltiVec vectors contain up to sixteen elements, while a shorter extension such as Intel's 64-bit MMX contain at most eight of the same type of element. This means that simple data rearrangement operations (e.g. merge) cover a relatively larger fraction of all possible mappings in the case of the shorter vector length. [Lee00] presents a novel set of simple data communication primitives which can perform all 24 permutations of a 2x2 matrix in a single cycle on a processor with dual data communication functional units. This is useful because any larger data communication problem can be decomposed into 2x2 matrices. Although this approach might make some very complex data communication patterns slow to compute, we have found that most multimedia algorithms have patterns which are relatively simple. Because of this we endorse [Lee00]'s technique for covering the data communication needs of multimedia applications.

A related class of instructions that the AltiVec extension included, that was quite useful, was a set of "splat" instructions, which place either an immediate scalar or specified element from a source register into every element of the destination register. This was very useful when constants were required; on other architectures it is necessary to statically store these types of values in memory, and then load them to a register when required.

5.3.6 Prefetching

Prefetching is a hardware or software technique which tries to predict data access needs in advance, overlapping memory access with useful computation. Although we will not otherwise examine the performance implications of prefetch instructions (which would be a useful extension to this study), we mention them briefly because they are often a part of multimedia instruction sets due to the highly predictable nature of data accesses in multimedia applications.

Software prefetch instructions are used to fetch data into the cache from main memory without blocking true load/store instruction accesses. Determining the ideal location for prefetch instructions in a piece of code depends on many architectural parameters. Unfortunately, these include such things as the number of CPU clocks for memory latency and the number of CPU clocks to transfer a cache line, which are both highly machine dependent and not readily available to the programmer.

Rather than issuing an explicit prefetch instruction for each desired data prefetch, Motorola's AltiVec uses a single *data stream touch instruction* (`dst`) which indicates the memory sequence or pattern that is likely to be accessed. We will refer to this hybrid of hardware and software prefetching as *software directed prefetching* to indicate that a separate prefetch instruction need not be issued for each data element. A data stream is defined by a sequence starting address, size of each unit (up to 32 128-bit blocks), total number of units (up to 256), bytes between units (-32768..+32767) and a 2-bit ID tag for the stream. Hardware optimizes the number of cache blocks to prefetch so it is not necessary for the programmer to know the parameters of the cache system. A stream is fetched either until all of the requested blocks have been brought into the cache or another `dst` instruction is issued with the same tag ID. The stream construct eliminates the instruction issue overhead as well as the problem of determining the optimal prefetch distance.

5.4 Bottlenecks and Unnecessary Features

In this section we discuss those features which appear in multimedia instruction sets, do not appear to be useful, and are not "free"; i.e. they aren't a low (or no) cost side effect of some other useful feature.

Instruction Primitives The VIS instruction set does not include full 16-bit multiply instructions. It instead offers multiplication primitives, the results of which must be combined through addition (see Algorithms 15 and 16).

Algorithm 15 Sun VIS 16-bit x 16-bit →16-bit Multiply

```
fmul8sux16   %f0, %f2, %f4
fmul8ulx16   %f0, %f2, %f6
fpadd16      %f4, %f6, %f8
```

The Mix stereo kernel is a good example of the high cost of synthesizing needed instruction functionality from other primitives. Audio mixing consists of multiplying a vector of

Algorithm 16 Sun VIS 16-bit x 16-bit →32-bit Multiply

```
fmuld8sux16  %f0, %f2, %f4
fmuld8ulx16  %f0, %f2, %f6
fpadd32      %f4, %f6, %f8
```

count input signals (*sp*[]) by a vector of mixing coefficients $chan_1, chan_2$ and summing the result (Algorithm 17). In the Timidity MIDI music synthesis application, fixed point integer computations are used to mix the various signed 16-bit instrument sounds into a 32-bit output buffer.

Algorithm 17 Mix Stereo

```
INT16 **sp_p; INT32 **lp_p; INT32 count; INT32 chan_1; INT32 chan_2;
  INT16 s, *sp = *(sp_p); INT32 *lp = *(lp_p);
  while (count--) {
    s = *sp++;
    *lp++ += s*chan_1;
    *lp++ += s*chan_2;
  }
  *(sp_p) = sp;
  *(lp_p) = lp;
}
```

Comparing the code snippet in Algorithm 18 to Algorithm 19 we can see that Sun's approach of synthesizing functionality from primitives (especially in the case of synthesizing a 16-bit merge) is much more costly than using a single instruction.

Algorithm 18 Intel Mix

```
movq    mm0, [esi]          ; mm0:  |  s3 |  s2 |  s1 |  s0 |
movq    mm1, [edi]          ; mm1:  |    lp1    |    lp0    |
movq    mm2, [edi + 8]      ; mm2:  |    lp3    |    lp2    |
pshufw  mm5, mm0, 00000000b ; mm5:  |  s0 |  s0 |  s0 |  s0 |
pshufw  mm6, mm0, 01010101b ; mm6:  |  s1 |  s1 |  s1 |  s1 |
pmaddwd mm5, mm7            ; mm5:  | s0*right  |  s0*left  |
pmaddwd mm6, mm7            ; mm6:  | s1*right  |  s1*left  |
paddd   mm1, mm5            ; mm1:  |    lp1'   |    lp0'   |
paddd   mm2, mm6            ; mm2:  |    lp3'   |    lp2'   |
```

The reason that the architects of VIS divided up 16-bit multiplication in this way was to decrease die area. Not providing a full 16x16 multiplier subunit cut the size of the arrays in half [Trem96b]. Unfortunately, dividing an operation into several instructions (which are not otherwise useful in and of themselves) increases register pressure, decreases instruction decoding bandwidth and creates additional data dependencies. Splitting SIMD instructions (which have been introduced for their ability to extract data parallelism) can actually cripple a superscalar processor's ability to extract instruction level parallelism. A multi-cycle operation can be a better solution than a multi-instruction operation because instruction latencies can be transparently upgraded in future processors, while poor instruction semantics can not be repaired without adding new instructions.

Unused High Latency Approximation Instructions Floating point approximation of $\frac{1}{x}$ instructions, although available on several platforms, did not find application in any of the kernels we studied. AltiVec also includes approximate log_2 and exp_2 instructions, which find application in the lighting stage of a 3D rendering pipeline; this is currently handled by 3D accelerator cards, and not the CPU.

Algorithm 19 Sun Mix

```
wr         %g0, 6, %gsr     !set alignment for right shift by 16
ld         [%l1 +  0], %f16 !%f16:|    lp0    |XXXXXXXXXXX|
ld         [%l1 +  4], %f17 !%f17:|    lp0    |    lp1    |
ld         [%l1 +  8], %f18 !%f18:|    lp2    |XXXXXXXXXXX|
ld         [%l1 + 12], %f19 !%f19:|    lp2    |    lp3    |
!  simulate 16-bit merge
fpmerge    %f28, %f28, %f4  !%f4: |s0_|s0_|_s0|_s0|s1_|s1_|_s1|_s1|
fpmerge    %f29, %f29, %f10 !%f10:|s2_|s2_|_s2|_s2|s3_|s3_|_s3|_s3|
faligndata %f4, %f4, %f6    !%f6: |_s1|_s1|s0_|s0_|_s0|_s0|s1_|s1_|
faligndata %f10, %f10, %f12 !%f12:|_s3|_s3|s2_|s2_|_s2|_s2|s3_|s3_|
fpmerge    %f6, %f4, %f8    !%f8: |XXX|XXX|XXX|XXX|   s0  |   s0  |
fpmerge    %f12, %f10, %f14 !%f14:|XXX|XXX|XXX|XXX|   s2  |   s2  |
fpmerge    %f7, %f5, %f2    !%f2: |XXX|XXX|XXX|XXX|   s1  |   s1  |
fpmerge    %f13, %f11, %f26 !%f26:|XXX|XXX|XXX|XXX|   s3  |   s3  |
fsrc1s     %f9, %f2         !%f2: |  s0 |  s0 |  s1 |  s1 |
fsrc1s     %f15, %f26       !%f26:|  s2 |  s2 |  s3 |  s3 |
!  simulate 16x16 -> 32-bit multiply
fmuld8sux16 %f0, %f2, %f4
fmuld8ulx16 %f0, %f2, %f6
!  simulate 16x16 -> 32-bit multiply
fmuld8sux16 %f0, %f3, %f8
fmuld8ulx16 %f0, %f3, %f10
fpadd32    %f4, %f6, %f4    !  %f4:  |  s0*chan1 |  s0*chan2 |
fpadd32    %f8, %f10, %f6   !  %f6:  |  s1*chan1 |  s1*chan2 |
fpadd32    %f16, %f4, %f16  !  %f16: |    lp0'   |    lp1'   |
fpadd32    %f18, %f6, %f18  !  %f18: |    lp2'   |    lp3'   |
```

Unused Pixel Conversion Instructions Motorola's AltiVec extension includes pixel pack (`vpkpx`) and pixel unpack (`vupkhpx`, `vupklpx`) instructions for converting between 32-bit true color and 16-bit color representations. These did not find application within the BMKL, although it is possible that they might be of utility in situations where AltiVec needs to operate on 16-bit color data; many video games use 16-bit textures, for example.

Unused Memory Access Instructions Sun's VIS includes two sets of instructions for accelerating multimedia operations with sophisticated memory addressing needs. The first, `edge8`, `edge16`, and `edge32`, produce bit vectors to be used in conjunction with partial store instructions to deal with the boundaries in 2D images. The second group of addressing instructions include `array8`, `array16` and `array32` which find use in *volumetric imaging* (the process of displaying a two dimensional slice of a three dimensional data). An array instruction converts (x, y, z) coordinates into a memory address. The Berkeley multimedia workload does not include any volumetric imaging applications, so it is unsurprising that these instructions found no utility in our workload.

Singular, Highly Utilized Resources Although we usually think of SIMD architectures as extracting data level parallelism, all of the implementations of the instruction sets we have examined are also superscalar, with multiple parallel SIMD functional units. In fact, unless the SIMD vector length is long enough to hold the entire data set being operated on, there is almost always the potential to extract instruction level parallelism as well. In coding the kernels with Sun's VIS extension, it became clear that instruction level parallelism was being compromised by the over utilized graphics status register (GSR).

Sun's VIS architecture does not include partitioned shift instructions, the GSR has a 3-bit `addr_offset` field which is used implicitly for byte granularity alignment, as well

as a 4-bit `scale_factor` field for packing/truncation operations. The VIS GSR is a serializing bottleneck because any time packing or alignment functionality is needed, it must be pushed through the GSR. Because VIS lacks partitioned shift operations, we found ourselves synthesizing such operations with the packing and alignment operations where no other algorithmic path was possible. Even with careful planning of packing and alignment operations it was often necessary to write to the GSR several times in each iteration of the loops of our multimedia kernels. The serializing effect of this singular resource prevented VIS operations from proceeding at the fullest possible degree of parallelism.

5.5 Alignment and Memory Traffic

Factors such as register file geometry (the number of registers and their width), data path location (pre-existing integer or floating point, or separate) and alignment issues are reflected in the uncached memory traffic - the data accesses as seen by the L1 data cache. Table 3 lists the average number of bytes loaded or stored per function call. This was computed by multiplying the number of dynamic load and store instructions executed by their widths in bytes.

From Table 3 we can see that Motorola's and Sun's implementations sometimes seem to transfer (load and store) more bytes in each function call than the AMD, DEC or Intel implementations of the same kernel. We would expect the Motorola and Sun implementations to spill registers to memory less frequently due to their larger register files (on average we see that the register file geometry does affect memory traffic as we might expect). This surprising result is actually an artifact of how we computed memory traffic, rather than an indication of an architectural shortcoming. Dynamic instruction counts alone are not a completely accurate predictor of actual memory traffic, since some of the architectures (AMD, DEC, Intel) support unaligned loads (hiding some loads issued by the hardware which handles unaligned memory access in the CPU) and the rest do not. Hardware support to efficiently handle memory accesses that are not aligned are expensive in both area and timing [Thak99].

Transparently Forced Alignment The AltiVec instruction set architecture does not provide for alignment exceptions when loading and storing data. Alignment is maintained by forcing the lower four bits of any address to be zero. This is transparent to the programmer, so the programmer is responsible for guaranteeing alignment, otherwise incorrect data may be loaded or stored. We believe it is better that performance and correctness issues due to alignment be made explicit. The loading of incorrect data due to a mistaken assumption about alignment would be an extremely difficult bug to track down.

5.6 Overall Instruction Mix

Table 4 shows what types of instructions comprised the total mix of dynamic instructions executed by each architecture. Counts are for the code within the kernels only, and do not include instructions from the rest of each application. Control state instructions are those which interact with special purpose machine registers (e.g. the GSR on Sun's VIS). Branches and other data dependent codes such as conditional moves are categorized as "control flow" type instructions.

We can see that SIMD kernels utilize a significant number of scalar operations for pointers, loop variables and clean up code; evidence that sharing the integer data path is not a good idea. Intel's SSE is a 128-bit wide extension, as compared to AMD's 64-bit wide 3DNow!, explaining why Intel's overall instruction count is lower by about 1 billion instructions. The same reasoning applies to Motorola's G4 which has the 128-bit wide (both packed integer and floating point) AltiVec extension. DEC's bloated instruction count is due to the fact that their MVI extension is very limited in functionality (13 instructions in total), and so many operations need to be done with scalar instructions.

Data communication operations represent the overhead necessary to put data in a format amenable to SIMD operations. Ideally, these types of instructions should make up as small a part of the dynamic instruction mix as possible. Table 4 reveals that the Motorola AltiVec extension executes the largest percentage of these instructions. This is due to two factors: 1) the wider register width means that it is less likely that the data is arranged correctly as first loaded from memory and 2) data communication operations are used by AltiVec to simulate unaligned access to memory.

6 Performance Comparison

In order to establish a base line for performance, the average execution time of the C and SIMD assembly versions of the BMKL were measured on each platform (Table 5). A speedup from 2x-5x was typical, although some kernels were sped up considerably more than this. Note that the C compiler for the Apple system running a pre-release version of OS X (developer's preview 3) is known to be weak, making the speedup of AltiVec over C look unrealistically good.

All of our performance measurements utilize the metric of time rather than cycle or instruction counts. If all of the architectures in our study had equal cycle times, then comparing cycle times would be valid, since time, in that case, would simply be proportional. This is not the case, since the Sun UltraSPARC IIi used in our study has a 360 MHz clock, while the remainder of the chips are 500 MHz parts. Instruction counts are not valid measure for cross architectural comparisons, as each instruction set does not necessarily do the same amount of work (computation) in the same number of instructions, nor take a the same amount of time to perform similar instructions.

In order to measure how well or how poorly a given platform performs relative to the competition we use the metric of *percent deviation from the mean*:

$$\%DM = 100 \cdot \left(\frac{\overline{t} - t_i}{\overline{t}} \right) \tag{2}$$

where t_i is the time taken on platform i, and $\overline{t}$ is the average performance (time) across all of the platforms examined for

	AMD		DEC		Intel		Motorola		Sun		Average
Register File Geometery	8x64b (FP)		31x64b (Int)		8x64b (FP), 8x128b		32x128b		32x64b		
Add Block	296	*2.8%*	249	*18.2%*	290	*4.7%*	431	*-41.7%*	256	*16.0%*	304
Block Match	356	*27.7%*	572	*-16.2%*	356	*27.8%*	730	*-48.2%*	448	*9.0%*	492
Clip Test & Project	5,261	*-10.8%*	*6,755*	*-42.3%*	2,840	*40.2%*	2,139	*54.9%*	*6,735*	*-41.9%*	4,746
Color Space	14,517,360	*-101.9%*	2,073,744	*71.2%*	14,517,360	*-101.9%*	2,419,504	*66.3%*	2,419,296	*66.3%*	7,189,453
DCT	2,424	*52.4%*	*19,008*	*-273.1%*	2,420	*52.5%*	304	*94.0%*	1,320	*74.1%*	5,095
FFT	99,797	*12.0%*	*171,163*	*-50.9%*	95,117	*16.1%*	118,560	*-4.5%*	*82,408*	*27.3%*	113,409
IDCT	1,640	*-60.6%*	*682*	*33.2%*	1,640	*-60.6%*	320	*68.7%*	824	*19.3%*	1,021
Max Value	852	*20.6%*	782	*27.1%*	852	*20.6%*	2,098	*-95.5%*	783	*27.1%*	1,073
Mix Stereo	823	*-17.5%*	*503*	*28.2%*	823	*-17.5%*	505	*27.9%*	849	*-21.2%*	701
Quantize	4,733	*45.6%*	*7,391*	*15.1%*	4,675	*46.3%*	21,098	*-142.5%*	*5,609*	*35.5%*	8,701
Short Term Anal. Filter	8,584	*-105.0%*	*3,056*	*27.0%*	8,584	*-105.0%*	272	*93.5%*	440	*89.5%*	4,187
Short Term Synth. Filter	7,100	*-93.3%*	*3,448*	*6.1%*	7,100	*-93.3%*	274	*92.5%*	441	*88.0%*	3,673
Subsample Horizontal	13,685,804	*-67.3%*	*2,945,448*	*64.0%*	13,685,892	*-67.3%*	5,642,508	*31.0%*	4,930,600	*39.7%*	8,178,050
Subsample Vertical	8,300,932	*-88.5%*	1,123,328	*74.5%*	8,301,052	*-88.5%*	2,137,092	*51.5%*	2,160,040	*51.0%*	4,404,489
Synthesis Filter	4,080	*-2.2%*	*4,136*	*-3.6%*	4,144	*-3.8%*	3,273	*18.0%*	*4,328*	*-8.4%*	3,992
Xform & Normalize	3,037	*-34.6%*	*1,976*	*12.5%*	2,162	*4.2%*	1,659	*26.5%*	*2,451*	*-8.6%*	2,257
Average	**2,290,192**	***-26.3%***	**397,640**	***-0.6%***	**2,289,707**	***-20.4%***	**646,923**	***18.3%***	**601,052**	***28.9%***	

Table 3: **SIMD Kernel Memory Traffic** - data bytes transferred per call as seen by the L1 cache (in other words, uncached memory traffic) are listed in the left subcolumns, with the percent deviation from the mean values (%DM) given in the right subcolumns. Values in *italics* indicate kernels which are implemented in C due to lacking SIMD instruction set functionality.

	AMD Athlon		DEC 21264A		Intel Pentium III		Motorola G4		Sun UltraSPARC IIi	
Int Load/Store	2.41E+08	*(5.4%)*	1.51E+09	*(20.3%)*	2.19E+08	*(6.2%)*	1.09E+08	*(2.4%)*	1.47E+08	*(2.6%)*
Int Arithmetic	5.43E+08	*(12.1%)*	2.18E+09	*(29.2%)*	4.34E+08	*(12.3%)*	8.65E+08	*(19.1%)*	1.49E+09	*(26.6%)*
Int Control Flow	5.75E+08	*(12.8%)*	8.41E+08	*(11.3%)*	4.73E+08	*(13.4%)*	6.26E+08	*(13.8%)*	5.20E+08	*(9.3%)*
Int Data Communication	1.03E+08	*(2.3%)*	4.98E+08	*(6.7%)*	7.50E+07	*(2.1%)*	4.75E+07	*(1.1%)*	0.00E+00	*(0.0%)*
FP Load/Store	1.42E+08	*(3.2%)*	8.36E+08	*(11.2%)*	1.40E+08	*(4.0%)*	3.35E+08	*(7.4%)*	4.48E+08	*(8.0%)*
FP Arithmetic	0.00E+00	*(0.0%)*	1.34E+09	*(18.1%)*	0.00E+00	*(0.0%)*	1.04E+08	*(2.3%)*	4.69E+08	*(8.4%)*
FP Control Flow	0.00E+00	*(0.0%)*	1.05E+07	*(0.1%)*	0.00E+00	*(0.0%)*	0.00E+00	*(0.0%)*	3.50E+08	*(6.2%)*
FP Data Communication	0.00E+00	*(0.0%)*	0.00E+00	*(0.0%)*	0.00E+00	*(0.0%)*	8.73E+05	*(0.0%)*	0.00E+00	*(0.0%)*
SIMD Load/Store	8.70E+08	*(19.4%)*	0.00E+00	*(0.0%)*	7.40E+08	*(21.0%)*	8.12E+08	*(17.9%)*	6.56E+08	*(11.7%)*
SIMD Data Communication	5.50E+08	*(12.3%)*	0.00E+00	*(0.0%)*	4.43E+08	*(12.6%)*	7.67E+08	*(17.0%)*	5.44E+08	*(9.7%)*
SIMD Int Arithmetic	5.61E+08	*(12.5%)*	2.28E+08	*(3.1%)*	6.62E+08	*(18.8%)*	5.36E+08	*(11.8%)*	7.61E+08	*(13.6%)*
SIMD Int Control Flow	0.00E+00	*(0.0%)*	0.00E+00	*(0.0%)*	0.00E+00	*(0.0%)*	5.23E+03	*(0.0%)*	1.97E+08	*(3.5%)*
SIMD FP Arithmetic	8.95E+08	*(19.9%)*	0.00E+00	*(0.0%)*	3.27E+08	*(9.3%)*	3.12E+08	*(6.9%)*	0.00E+00	*(0.0%)*
SIMD FP Control Flow	2.91E+05	*(0.0%)*	0.00E+00	*(0.0%)*	1.46E+05	*(0.0%)*	8.12E+04	*(0.0%)*	0.00E+00	*(0.0%)*
Control State	8.24E+06	*(0.2%)*	0.00E+00	*(0.0%)*	8.35E+06	*(0.2%)*	1.25E+07	*(0.3%)*	2.16E+07	*(0.4%)*
Total	**4.49E+09**	***100%***	**7.45E+09**	***100%***	**3.52E+09**	***100%***	**4.53E+09**	***100%***	**5.60E+09**	***100%***

Table 4: **Overall Instruction Mix** - counts are listed with percentages in parentheses. Control state instructions are those which interact with special purpose registers. Control flow instructions include branches as was as conditional moves and comparisons.

a particular kernel. This metric indicates how much better or worse than average, and provides a normalized result for computing the average improvement or degradation in performance. Table 6 lists the average %DM (the arithmetic average of the %DM values for all of the kernels on a given platform).

The algorithm employed by the original MPEG-2 encoder DCT code (Algorithm 3) is not very efficient - it uses double precision floating point where fixed point integer should provide sufficient precision (and is typically faster on most architectures) [IEEE91]. Because the DCT and IDCT algorithms are of the same computational order of magnitude it might seem strange that they demonstrate such different performance improvements (Table 5). Unlike the original forward DCT code which was computed in floating point, the scalar inverse DCT source code (Algorithm 4) is written in fixed point integer. Thus it is much more directly comparable to our fixed point integer SIMD implementations.

The fact that the original C DCT algorithm is floating point and the SIMD implementations are fixed-point integer meant that it was not appropriate to use the original code as the "solution" for the DEC Alpha architecture, which did not provide

	AMD Athlon			DEC Alpha 21264			Intel Pentium III			Motorola G4			Sun UltraSPARC IIi			Arithmetic Mean		
Add Block	**(9.6x)**	1,087	***114***	**(1.9x)**	669	***350***	**(6.4x)**	969	***152***	**(2.5x)**	1,054	***423***	**(4.9x)**	1,662	***342***	**(5.0x)**	1,088	276
Block Match	**(7.0x)**	1,801	***257***	**(2.6x)**	979	***379***	**(4.9x)**	1,855	***381***	**(3.4x)**	1,466	***437***	**(2.8x)**	2,408	***852***	**(4.1x)**	1,702	461
Clip Test & Project	**(1.6x)**	7,374	***4,559***	**(1.0x)**	8,591	***8,591***	**(1.3x)**	7,938	***6,336***	**(2.6x)**	8,906	***3,427***	**(1.0x)**	12,222	***12,222***	**(1.5x)**	9,006	7,027
Color Space	**(9.4x)**	122,103,855	***12,924,258***	**(1.7x)**	22,399,926	***13,043,849***	**(5.6x)**	60,141,946	***10,803,618***	**(4.5x)**	58,043,968	***12,959,530***	**(2.7x)**	62,015,071	***22,994,253***	**(4.8x)**	64,940,953	14,545,102
DCT**	**(16.9x)**	24,332	***1,438***	**(19.7x)**	12,475	***632***	**(31.1x)**	38,192	***1,228***	**(36.6x)**	33,176	***907***	**(9.2x)**	30,922	***3,353***	**(22.7x)**	27,819	1,512
FFT	**(2.0x)**	125,484	***63,446***	**(1.0x)**	67,321	***67,321***	**(1.7x)**	147,047	***84,661***	**(2.0x)**	232,272	***116,828***	**(1.0x)**	111,258	***111,258***	**(1.5x)**	136,677	88,703
IDCT	**(3.6x)**	2,999	***827***	**(1.0x)**	1,276	***1,276***	**(2.8x)**	2,772	***993***	**(3.9x)**	3,326	***847***	**(1.5x)**	3,782	***2,508***	**(2.6x)**	2,831	1,290
Max Value	**(4.0x)**	2,407	***604***	**(1.0x)**	1,143	***1,168***	**(3.8x)**	2,536	***669***	**(6.6x)**	4,110	***618***	**(0.7x)**	6,165	***8,715***	**(3.2x)**	3,272	2,355
Mix Stereo	**(2.8x)**	956	***341***	**(1.0x)**	616	***616***	**(2.0x)**	1,065	***528***	**(3.8x)**	1,710	***446***	**(2.1x)**	3,550	***1,716***	**(2.3x)**	1,579	729
Quantize	**(1.8x)**	34,233	***18,557***	**(1.0x)**	19,549	***19,549***	**(2.5x)**	37,100	***15,010***	**(2.1x)**	29,488	***14,335***	**(1.0x)**	34,928	***34,928***	**(1.7x)**	31,059	20,476
Short Term Analysis Filter	**(1.5x)**	15,268	***9,887***	**(1.0x)**	11,120	***11,120***	**(1.5x)**	16,129	***11,003***	**(6.5x)**	24,156	***3,729***	**(2.5x)**	36,773	***15,009***	**(2.6x)**	20,689	10,150
Short Term Synthesis Filter	**(2.9x)**	21,849	***7,425***	**(1.0x)**	19,208	***19,208***	**(7.6x)**	43,582	***5,759***	**(6.5x)**	23,721	***3,672***	**(3.2x)**	48,660	***15,202***	**(4.2x)**	31,404	10,253
Subsample Horizontal	**(1.4x)**	15,977,835	***11,337,253***	**(1.0x)**	9,210,928	***9,264,521***	**(1.3x)**	15,777,956	***11,714,792***	**(1.6x)**	17,472,826	***10,849,858***	**(0.9x)**	17,707,829	***19,521,092***	**(1.3x)**	15,229,475	12,537,503
Subsample Vertical	**(2.4x)**	25,517,638	***10,791,669***	**(3.3x)**	14,680,887	***4,487,870***	**(2.2x)**	21,969,585	***10,049,618***	**(9.6x)**	29,349,708	***3,070,046***	**(3.8x)**	39,872,378	***10,457,565***	**(4.2x)**	26,278,039	7,771,354
Synthesis Filter	**(2.8x)**	7,308	***2,585***	**(1.0x)**	4,900	***4,900***	**(1.8x)**	8,349	***4,718***	**(1.8x)**	4,917	***2,727***	**(1.0x)**	7,138	***7,138***	**(1.7x)**	6,522	4,414
Transform & Normalize	**(2.5x)**	11,527	***4,597***	**(1.0x)**	7,990	***7,990***	**(4.7x)**	20,491	***4,338***	**(27.4x)**	81,985	***2,998***	**(1.0x)**	14,455	***14,455***	**(7.3x)**	27,290	6,876
Arithmetic Mean	**(4.5x)**	10,240,997	***2,197,989***	**(2.5x)**	2,902,974	***1,683,709***	**(5.1x)**	6,138,595	***2,043,988***	**(7.6x)**	6,582,299	***1,689,427***	**(2.5x)**	7,494,325	***3,325,038***			
Geometric Mean	**(3.1x)**	47,328	***15,063***	**(1.5x)**	27,110	***18,626***	**(3.1x)**	52,161	***16,694***	**(4.8x)**	60,056	***12,416***	**(1.7x)**	66,825	***38,312***			

Table 5: **Average Time per Call (ns)** - C times listed in normal font, SIMD assembly times listed in *italics*, speedup shown to the left inside of (parentheses). Kernels with a grey background were not implemented in SIMD due to insufficient instruction set functionality. (**) DCT kernel originally coded in floating point, but implemented in fixed point integer for SIMD codes.

	AMD Athlon			DEC Alpha 21264			Intel Pentium III			Motorola G4			Sun UltraSPARC IIi		
Add Block	**90.0%**	0.1%	***58.9%***	**-62.1%**	38.6%	***-26.8%***	**26.2%**	10.9%	***44.8%***	**-50.6%**	3.1%	***-53.2%***	**-3.5%**	-52.7%	***-23.7%***
Block Match	**69.5%**	-5.8%	***44.2%***	**-37.4%**	42.5%	***17.8%***	**18.2%**	-9.0%	***17.5%***	**-18.8%**	13.9%	***5.2%***	**-31.5%**	-41.5%	***-84.7%***
Clip Test & Project	**8.3%**	18.1%	***35.1%***	**-33.1%**	4.6%	***-22.3%***	**-16.1%**	11.9%	***9.8%***	**74.0%**	1.1%	***51.2%***	**-33.1%**	-35.7%	***-73.9%***
Color Space	**97.6%**	-88.0%	***11.1%***	**-64.1%**	65.5%	***10.3%***	**16.4%**	7.4%	***25.7%***	**-6.3%**	10.6%	***10.9%***	**-43.6%**	4.5%	***-58.1%***
DCT**	**-25.5%**	12.5%	***4.9%***	**-13.1%**	55.2%	***58.2%***	**37.0%**	-37.3%	***18.8%***	**61.0%**	-19.3%	***40.0%***	**-59.4%**	-11.2%	***-121.8%***
FFT	**28.4%**	8.2%	***28.5%***	**-35.1%**	50.7%	***24.1%***	**12.7%**	-7.6%	***4.6%***	**29.1%**	-69.9%	***-31.7%***	**-35.1%**	18.6%	***-25.4%***
IDCT	**41.1%**	-5.9%	***35.9%***	**-61.1%**	54.9%	***1.1%***	**8.6%**	2.1%	***23.0%***	**52.7%**	-17.5%	***34.3%***	**-41.3%**	-33.6%	***-94.4%***
Max Value	**23.6%**	26.4%	***74.3%***	**-69.6%**	65.1%	***50.4%***	**17.7%**	22.5%	***71.6%***	**106.3%**	-25.6%	***73.7%***	**-78.0%**	-88.4%	***-270.1%***
Mix Stereo	**19.6%**	39.5%	***53.3%***	**-57.4%**	61.0%	***15.6%***	**-14.1%**	32.6%	***27.5%***	**63.7%**	-8.3%	***38.9%***	**-11.8%**	-124.8%	***-135.3%***
Quantize	**10.2%**	-10.2%	***9.4%***	**-40.3%**	37.1%	***4.5%***	**47.6%**	-19.4%	***26.7%***	**22.8%**	5.1%	***30.0%***	**-40.3%**	-12.5%	***-70.6%***
Short Term Analysis Filter	**-40.3%**	26.2%	***2.6%***	**-61.4%**	46.3%	***-9.6%***	**-43.3%**	22.0%	***-8.4%***	**150.3%**	-16.8%	***63.3%***	**-5.3%**	-77.7%	***-47.9%***
Short Term Synthesis Filter	**-30.5%**	30.4%	***27.6%***	**-76.4%**	38.8%	***-87.3%***	**78.7%**	-38.8%	***43.8%***	**52.6%**	24.5%	***64.2%***	**-24.4%**	-54.9%	***-48.3%***
Subsample Horizontal	**12.4%**	-4.9%	***9.6%***	**-20.7%**	39.5%	***26.1%***	**7.4%**	-3.6%	***6.6%***	**28.5%**	-14.7%	***13.5%***	**-27.6%**	-16.3%	***-55.7%***
Subsample Vertical	**-44.2%**	2.9%	***-38.9%***	**-22.8%**	44.1%	***42.3%***	**-48.4%**	16.4%	***-29.3%***	**125.5%**	-11.7%	***60.5%***	**-10.1%**	-51.7%	***-34.6%***
Synthesis Filter	**68.3%**	-12.0%	***41.4%***	**-40.5%**	24.9%	***-11.0%***	**5.3%**	-28.0%	***-6.9%***	**7.3%**	24.6%	***38.2%***	**-40.5%**	-9.4%	***-61.7%***
Transform & Normalize	**-65.7%**	57.8%	***33.1%***	**-86.3%**	70.7%	***-16.2%***	**-35.4%**	24.9%	***36.9%***	**273.8%**	-200.4%	***56.4%***	**-86.3%**	47.0%	***-110.2%***
Arithmetic Average	**16.4%**	6.0%	***26.9%***	**-48.8%**	46.2%	***4.8%***	**7.4%**	0.4%	***19.5%***	**60.7%**	-18.8%	***31.0%***	**-35.7%**	-33.8%	***-82.3%***

Table 6: **Percent Deviation from the Mean (%DM)** - for data in Table 5, which is defined as $100 \cdot \left(\frac{\overline{t}-t_i}{\overline{t}}\right)$, where t_i is the time taken on platform i, and $\overline{t}$ is the average performance (time) across all of the platforms examined for a particular kernel.

sufficient SIMD instruction set functionality to implement a SIMD version of the DCT. A C fixed-point integer DCT was substituted, which is why a 19.7x speedup is shown in Table 5, even though we did not recode it.

AMD Athlon, Intel Pentium III The AMD Athlon and Intel Pentium III processors in our study at first glance appear to be very similar; both run at 500 MHz, and, as we noted in Section 4, both share Intel's MMX SIMD integer extension. In fact, because we implemented the Intel kernel set first, it was possible to simply reuse the same code for the AMD SIMD version of many of the integer kernels in the BMKL. The SIMD integer kernels include all but the clip test, FFT, quantize, synthesis filter and transform kernels. It is interesting to observe the differences in performance between the two processors on what is often identical code. A few salient architectural differences to note [Stil99]:

- Athlon has a 64 KB instruction, 64 KB data cache, while Pentium III's L1 caches are 16 KB/16 KB respectively
- Athlon has three instruction decoders working in parallel, while Pentium III has only two
- Athlon's pipeline is 10-cycles long, while Pentium III's is 12-17 cycles; the cost of branches in the Pentium is exacerbated by Pentium III's weaker branch prediction unit
- Intel's MMX instruction latency is lower (1 cycle SIMD add/sub, 1 cycle shuffle, 3 cycle SIMD mult) compared to Athlon (2 cycle SIMD add/sub, 2 cycle shuffle, 3 cycle SIMD mult)

So, although the SIMD integer instruction set is the same in both cases, AMD's Athlon SIMD integer is a more powerful implementation; its only deficiency is the higher simple SIMD integer instruction latency when compared to the Pentium III. We can see this very clearly in Table 5, although where each architectural difference comes into play depends on the kernel.

One interesting case is that of the DCT and IDCT kernels - the Pentium III is faster on the DCT, while the Athlon is faster on the IDCT. This is surprising given that the two kernels are algorithmically quite similar - each is basically a mirror image of the other. Upon further investigation, we found that the DCT code has shuffle instructions (`pshufw`) in several places that are not scheduled well for the Athlon instruction's higher latency; a circumstance which was not duplicated in the IDCT code.

In terms of multimedia, the greatest difference between the two processors are their SIMD floating point extensions. AMD's 3DNow! is quite similar to MMX, reusing the eight x87 floating point registers as 64-bit wide SIMD registers. Intel's SSE has new 128-bit wide registers and instructions, although in the Pentium III implementation, each 128-bit instruction is actually decoded into two 64-bit wide micro-ops. This does, however, give the Pentium III more overall register space to work with. Other important architectural features of the two SIMD floating point instruction sets:

- AMD's 3DNow! FP latency (4 cycle add/sub, 4 cycle multiply) is lower than Intel's SSE (4 cycle add/sub, 5 cycle multiply, throughput of 64-bits per cycle like 3DNow!)

In the arena of floating point operations, the AMD Athlon architecture eliminates the only deficiency it had compared to the Pentium III in SIMD integer - namely, slightly higher instruction latency. From Table 6, we can see that overall SIMD AMD Athlon code does 26.9% better than average, while the Pentium III is 19.7% better than the average of all of the multimedia instruction sets. Both chips perform well, but the AMD Athlon matches or outperforms the Intel chip on all but one SIMD floating point kernel: quantize.

Quantization is a process by which a real value is converted to a discrete integer representation. An array of real double precision floating point values, $xr[]$, is converted to an array of 32-bit integers, $xi[]$, according to the function $xi[i] = \sqrt{\sqrt{xr[i]*} + 0.4054}$. Note that original C implementation utilizes a lookup table for some values (not shown in Algorithm 20).

Algorithm 20 Quantize

```
static INT32 lutab[10000];
INT32 l_end; FLOAT64 xr[]; INT32 ix[]; FLOAT64 *istep_p;
FLOAT32 temp;
for (i=0;i<l_end;i++) {
  temp=(*istep)*fabs(xr[i]);
  ix[i] = (int)( sqrt(sqrt(temp)*temp) + 0.4054);
}
```

Although SIMD floating point instruction sets include approximations for $\frac{1}{\sqrt{a}}$, rather than $\sqrt{a}$, the equivalence $\sqrt{a} = a \cdot \frac{1}{\sqrt{a}}$ can be used. The reason for AMD's lackluster performance on the quantize kernel is clear based on our earlier discussion - AMD's reciprocal square root instructions are actually scalar - they only produce one result value, based on the lower packed element. This costs 3DNow! performance for this highly square-root intensive kernel.

DEC Alpha 21264 From Tables 5 and 6 we see that the DEC Alpha platform far outstrips the other systems in terms of compiled C performance. It has been claimed that a broader multimedia instruction set would not be useful on Alpha, as an extension like Intel's MMX only fixes x86 legacy-architecture performance deficiencies which are not present in the Alpha architecture [Rubi96]. Our performance comparison makes this sound rather dubious, as the kernels programmed with the extensions from AMD, Intel and Motorola were able to not only match the performance of those on the Alpha 21264, but often exceeded it.

Motorola G4 Motorola's AltiVec was the only multimedia extension which was architected from the ground up - all of the others in some way leverage existing resources. AltiVec in many ways agrees with our design suggestions (e.g. a large number of 128-bit wide registers), although in fact the instructions included in AltiVec are far more general than those required by multimedia applications. Almost every possible operation and data type is supported, which should allow it to be applied to other application domains as well. A 128-bit register width combined with latencies that are at least as good as those found on the other (64-bit) architectures, allows AltiVec to come in with the best overall performance (31% better than average on SIMD accelerated code, according to Table 6). However, performance on a few of the kernels is still poor, especially on add block and the FFT. The add block kernel's problem has already been discussed - the 128-bit register width is actually too long for this kernel, causing unnecessary data to be loaded from memory. The reason for the poor FFT performance is not entirely clear, although a review of our code revealed that the way in which we coded data to be stored to unaligned memory could be improved.

Sun UltraSPARC IIi The performance of the VIS multimedia extension was mediocre at best, although we should note that the UltraSPARC IIi processor examined is the only one in our study running at 360 MHz (the other processors all have a 500 MHz clock). It is also the oldest multimedia instruction set we have looked at, the second to be released after HP's MAX-1 (1996). As we have pointed out, this instruction set suffers from some odd instruction choices (e.g. multiplication primitives), missing functionality (no partitioned shift operations) and a highly utilized control register that creates a bottleneck (the graphics status register).

7 New Directions

In this section we describe two new ideas for future multimedia extensions based on features we found lacking during our coding experience.

7.1 Strided Memory Access

Consider the difference between how memory is loaded into a SIMD register in the horizontal and vertical subsampling kernels. The original C sources for the horizontal and vertical subsampling kernels are in Algorithms 21 and 22 respectively. In the horizontal subsampling case, a vector load can only directly retrieve the data from one iteration of the loop into a register. For vertical subsampling, the n^{th} element from each of M loop iterations is loaded (M is the number of packed element in a register) into a register without any data rearrangement. Because SIMD applies the same operation to all of the elements of a vector, the vertical subsampling kernel can be computed more efficiently.

Algorithm 21 Subsample Horizontal

```
 UINT8 *src; UINT8 *dst; INT32 width; INT32 height;
  for (j=0; j<height; j++) {
    for (i=0; i<width; i+=2) {
      ltmp = (22*(src[i-5]   + src[i+5])-52*(src[i-3]  + src[i+3])
              +159*(src[i-1] + src[i-1])+256*src[i]    + 256)>>9;
      /* clip result to UINT8 range 0..255 */
      dst[i>>1] = ltmp>255 ?  255 :  (ltmp<0 ?  0 :  ltmp);
    }
    src+= width; dst+= width>>1;
  }
```

Algorithm 22 Subsample Vertical

```
  UINT8 *src; UINT8 *dst; INT32 width; INT32 height;
  INT32 w, ltmp;
  w = width>>1;
  for (i=0; i<w; i++) {
   for (j=0; j<height; j+=2) {
    /* FIR filter with 0.5 sample interval phase shift */
    ltmp = (228*(src[w*j] + src[w*j+1])+70*(src[w*(j-1)] + src[w*j+2])
      -37*(src[w*j-2] + src[w*j+3])-21*(src[w*j-3] + src[w*j+4])
      +11*(src[w*j-4] + src[w*j+5])+5*(src[w*j-5] +
src[w*j+6])+256)>>9;
    /* clip result to UINT8 range 0..255 */
    dst[w*(j>>1)]=ltmp>255 ?  255 :  (ltmp<0 ?  0 :  ltmp);
   }
   src++;
   dst++;
 }
```

With current SIMD architectures, when registers contain the data for a single loop iteration, either operations on some of the packed elements must be nullified, or significant overhead must go into transposing the data, wasting computation. Although this worked acceptably well in the DCT and IDCT kernels, there are some cases, such as image processing, when it is not feasible to transpose an image - the overhead is far too great.

We propose that SIMD architectures implement strided load and store instructions to make the gathering of non-adjacent data elements more efficient. This is similar to the prefetch mechanism in AltiVec, except that the data elements would be assembled together by the hardware into a single register, rather than simply loaded into the cache. Of course such a memory operation would necessarily be slower than a traditional one, but it would cut down immensely on the overhead that would have to go into reorganizing data as loaded from memory. Strided loads and stores would have three operands:

Instruction	Syntax
Load Strided	`lvxstrd  vD, rA, rB`
Store Strided	`stvstrd  vD, rA, rB`

where in each case `rA` is the base address and `rB` contains a description of the memory access pattern:

width — width of the data elements to be loaded [2 bits], 00 = 8-bits, 01 = 16-bits, 10 = 32-bits, 11= 64-bits

offset — number of bytes offset from the base address from which to begin loading [6 bits], interpreted as an signed value: -32..+31

stride — number of bytes between the effective address of one element in the sequence and the next [24 bits], interpreted as a signed value: -8388608..+8388607

The color space conversion kernel is an excellent example of where strided load and store instructions could be used. Pixel data consists of one or more channels or bands, with each channel representing some independent value associated with a given pixel's (x, y) position. A single channel, for example, represents greyscale, while a three (or more) channel image is typically color. The band data may be interleaved (each pixel's red, green, and blue data are adjacent in memory) or separated (e.g. the red data for adjacent pixels are adjacent in memory). In image processing algorithms such as color space conversion we operate on each channel in a different way, so band separated format is the most convenient for SIMD processing. Converting from the RGB to YC_BC_R color space is done through the conversion coefficients shown in Algorithm 23.

Algorithm 23 Color Space Conversion

```
UINT8 *rowp, *y_p, *u_p, v_p;
INT32 red = *rowp++, green = *rowp++, blue = *rowp++;
*y_p++ = +0.2558*red + 0.5022*green + 0.0975*blue + 16.5;
*u_p++ = -0.1476*red - 0.2899*green + 0.4375*blue + 128.5;
*v_p++ = +0.4375*red - 0.3664*green - 0.0711*blue + 128.5;
```

Algorithm 24 replaces thirty-eight instructions in the original AltiVec color space conversion kernel (the corresponding code fragment is listed in the Appendix). In the original AltiVec code, it was necessary to load six permute control vectors (each 128-bits wide) before executing the six `vperm` instructions required to rearrange the data into band separated format.

Algorithm 24 Modified Color Space Conversion

```
rgb_to_yuv:
oris    r11,r11 RED_PATTERN
oris    r12,r12,GREEN_PATTERN
oris    r13,r13,BLUE_PATTERN
lvxstrd v28,0,r3,r11
;; v28: |r0|r1|r2|r3|r4|r5|r6|r7|r8|r9|rA|rB|rC|rD|rE|rF|
lvxstrd v29,0,r3,r12
;; v29: |g0|g1|g2|g3|g4|g5|g6|g7|g8|g9|gA|gB|gC|gD|gE|gF|
lvxstrd v30,0,r3,r13
;; v30: |b0|b1|b2|b3|b4|b5|b6|b7|b8|b9|bA|bB|bC|bD|bE|bF|
```

Traditional SIMD data communication operations have trouble with data which is not aligned on boundaries which are powers of two - in the case of color space conversion, visually adjacent pixels from each band are spaced 3 bytes apart. Strided loads and stores are by definition unaligned, so this would need to be handled by the load/store hardware in the CPU. It would also make sense to have additional versions of these instructions which would be a hint to circumvent the cache (on a load) or to not do write-allocation (on a store) if the cache lines containing the strided data elements would not be of near-term utility.

7.2 Superwide Registers

Generally, multimedia data is stored in a packed format and is loaded into registers into the same format. Frequently, unpacking is required before operations are performed, and the unpacked data, of course, no longer fits within a single register. We therefore propose:

- registers that are wider than the data loaded into them
- implicit unpack with load
- implicit pack with store

Our design is in some ways similar to the as yet unimplemented MIPS MDMX instruction set [MIPS97]. The MIPS MDMX extension has a 192-bit accumulator register as its architectural cornerstone, with the more usual style of register to register SIMD operations also included; non-accumulator SIMD operations share the 64-bit floating point datapath. The destination of normal SIMD instructions can be either another SIMD register or the accumulator (to be loaded or accumulated). When accumulating packed unsigned bytes, the accumulator is partitioned into eight 24-bit unsigned slices. Packed 16-bit operations cause the accumulator to be split into four 48-bit sections. This extra width allows for multiple accumulations to occur without overflow.

What is good about the MDMX accumulator approach is that implicit width promotion provides an elegant solution to overflow and other issues caused by packing data as tightly as possible. For example, multiplication is semantically difficult to deal with on SIMD architectures because the result of a multiply is longer than either operand [Lee97c]. This problem is avoided by the MDMX accumulator, because there is enough extra space to prevent overflow.

Fixed point arithmetic is also more precise because full precision results can be accumulated, and the total rounded only once at the end of the loop. Similarly, most scalar multimedia algorithms only specify saturation at the end of computation. This is because it is more precise to saturate once rather than at every step of the algorithm. For example, if we are adding three signed 16-bit values:

Saturation at Every Step: $32760 + 50 - 20 = 32747$

Saturation at Last Step: $32760 + 50 - 20 = 32767$

Unfortunately, in SIMD architectures where the packed elements maximally fill the available register space, the choice is either to be imprecise (compute with saturation at every step) or loose parallelism (explicitly promote the input data to be wider). Saturating arithmetic can also produce unexpected results since, unlike normal addition, the order of operations matters.

While the MIPS MDMX solution may seem elegant in principle, it ignores the architectural side of actually making such a design fast. The accumulator is a singular (unique) shared resource, and as such has a tendency to limit instruction level parallelism. We found that the existence of only a single accumulator was a severe handicap to avoiding data dependencies. On a non-accumulator but otherwise superscalar architecture it is usually possible to perform some other useful, non data dependent operation in parallel so that the processing can proceed at the greatest degree of instruction level parallelism possible. On MDMX all computations which need to use the accumulator must proceed serially.

Low Precision		High Precision	
Storage	**Computation**	**Storage**	**Computation**
8U	12Sx16	8U	24Sx8
16S	24Sx8	16S	48Sx4
32S	48Sx4	-	-
32FP	32FPx4	-	-

Table 7: **Supported Data Types** - data types are divided into storage (how it is stored in memory) and computation (the width of arithmetic and other instruction elements)

Supported Data Types In order to avoid the problems with MDMX, we suggest that a normal register file architecture be used, with the entire SIMD register file made wide (for example, 192-bits). Supported memory (load and store) data types include those that we have seen to be of importance in multimedia for intermediate and storage formats: 8-bit unsigned, 16-bit signed, 32-bit signed and single precision floating point. The data types actually supported by packed arithmetic operations are different: 12-bit signed, 24-bit signed, 48-bit signed and single precision floating point. Depending on the algorithm it may be desirable to utilize the extra bits of a superwide register for either accumulation or data parallelism. We suggest supporting two memory widths: 64-bits (high-precision) and 128-bits (low-precision), which are unpacked to different computational widths. This also allows for better matching with algorithms that have different natural widths (e.g. add block, which works best with a 64-bit vector length).

Packed floating point data types present an interesting design choice - we can either operate on:

1. four single precision values in parallel (not using 16 of the bits in each register element)
2. four single precision values which have been expanded to a 48-bit extended precision format
3. six single precision values, exactly filling a 192-bit register

The downside of the second solution is that SIMD results may not exactly match scalar results. In addition, the latency of many floating point operations depends heavily on the precision being computed, so a more precise operation is a higher latency one. The third solution, although attractive for its additional data parallelism, is problematic because it would require its own set of data rearrangement instructions based on a six element (rather than four or eight element) vector. For our sample design, we chose the first option.

Supported Operations Because we have fundamentally changed the treatment of multimedia data types, we also need to reexamine which operations are still valuable in this new light. Several instructions are no longer useful:

- average - the only useful average instruction data type within our workload was for 8-bit unsigned values. Average is only really useful on existing multimedia architectures because it allows for computation without width

promotion. On a superwide register architecture average instructions are unnecessary, since the requisite functionality can be synthesized through shift and add (operations which are useful in and of themselves), and there is already sufficient precision.

- saturating arithmetic - saturation is done implicitly during packing. Of course, max and min instructions can always be used if an exotic type of clamping (e.g. 9-bit signed in the IDCT) need to be supported.

- pack/unpack - performed implicitly with loads and stores

- truncating multiplication - *truncation* predefines a set number of result bits to be thrown away. This primarily has application when multiplying n-bit fixed point values with fractional components which together take up a total of n-bits of precision. Unfortunately, this plays havoc with precision since it is usually desirable to truncate once, at the end of a fixed-point computation, rather than at every step. Because all of the high precision computational data types in our design are more than wide enough to hold the product of two storage data types, overflow is never a problem.

In the Appendix we present our proposed instruction set (97 instructions in total) for a superwide register SIMD multimedia extension. Unlike existing instruction sets which are fundamentally byte-based, this instruction set is centered around quantities which are multiples of 12-bits wide. This can be though of as four extra bits of precision for every byte of actual storage data loaded.

Example A small example of how to code for the proposed architecture is shown in Figure 1. Note that load and store operations specify both the computational and storage data type.

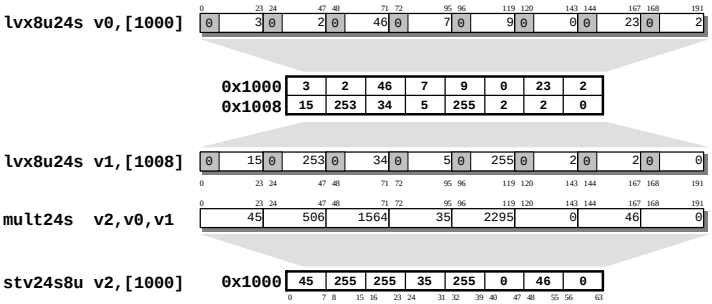


Figure 1: **Superwide Registers Example**

8 Summary

8.1 Useful Features

Many of the architectural features of existing multimedia instruction sets attempt to get around the limitations of tightly packed SIMD registers. With a superwide register architecture, many of the reasons for these features are eliminated, creating a simpler overall design. Our summary distills our conclusions about standard tightly packed SIMD, although we note were there are differences introduced by a superwide register approach.

8.1.1 Register File

- Sharing the floating point datapath is usually preferable to the integer data path because there is no contention with pointer and loop variables, and less chance of affecting the critical path of the processor. If no existing data path is to be shared by SIMD instructions, a 128-bit wide data path is optimal for most multimedia algorithms, 192-bits in the case of a superwide architecture.

- Multimedia algorithms can take advantage of large register files - we suggest *at least* 16 128-bit registers, or 32 64-bit registers.

8.1.2 Data Types

- 8-bit signed data types are not useful. 8-bit unsigned data types are most often used for storage, rather than computation.

- 16-bit signed data types are the most common; they are the intermediate (computational) data type for video and the storage (and sometimes computational) data type for audio and speech algorithms. Unsigned 16-bit values are not useful.

- 32-bit signed values are most often used for accumulation. Unsigned 32-bit values are not useful.

- Single precision floating point (32-bit) is found in the audio and 3D graphics (geometry) kernels. We did not come across a multimedia algorithm which required double precision (64-bit) floating point.

8.1.3 Integer Arithmetic

- Saturation prevents overflow in a fast, numerically acceptable way for SIMD operations, although with our proposed superwide register architecture, saturation is really not needed except when packing.

- Max and min operations are an efficient way to perform SIMD comparisons as well as clamping to arbitrary ranges. They are useful at all computational data widths.

- Average instructions we found only to be useful in the MPEG encode block match kernel for interpolation (8-bit unsigned data type). They are less useful on a superwide register architecture, because there averaging can be done through an add and subsequent shift right by one bit without unpacking to a wider width.

- Shift operations of all types are useful at all data widths - they are critical for fixed point arithmetic, and also provide an efficient means for data realignment and division and multiplication by powers of two.

- Sum of absolute difference instructions are only useful for MPEG encoding and other video encoding algorithms which utilize motion compensation (block match kernel - 8-bit unsigned data type).

8.1.4 Floating Point Arithmetic

- $\frac{1}{\sqrt{x}}$approximation instructions are useful; through multiplication they can also estimate $\sqrt{x}$ and $\frac{1}{x}$. A full precision version of this instruction is not necessary, as the Newton-Raphson method can always be used to improve the precision of the approximation.

- Exceptions and sophisticated rounding modes (as specified by the IEEE floating point standard) are not necessary for multimedia; in any instance of where these might be used it is possible to substitute a reasonable value that will allow computation to continue unhindered, and still produce an acceptable result.

8.1.5 Data Rearrangement

- A full permute operation (as is found in AltiVec) is very flexible, but is probably overkill for most multimedia applications where data rearrangement patterns can be handled by simpler data rearrangement operations. However, it should be noted that in AltiVec the `vperm` instruction serves double duty as a means for aligning unaligned data loads, so its capabilities are basically free.

- [Lee00] presents a novel set of simple data communication primitives which can perform all 24 permutations of a 2x2 matrix in a single cycle on a processor with dual data communication functional units. We endorse this technique because any larger data communication problem can be decomposed into 2x2 matrices, and because most multimedia data rearrangement patters are simple; they can be done in a single cycle. [Lee00]'s instructions are preferable to `vperm` because they do not require a permutation control vector to first be loaded into memory, as their data communication patterns are statically defined.

8.1.6 Memory Operations

- Hardware support to efficiently handle memory accesses that are not aligned are expensive in both area and timing [Thak99]. Ideally, data would always be aligned by software (e.g. the compiler or run-time architecture). In some situations it is impossible to guarantee alignment. The strided load and store operations which we have proposed would be in many cases inherently unaligned, making hardware support a requirement. Also, for example, in the motion compensation step of MPEG video coding, unaligned memory access is needed depending on the motion vector [Kuro98], as the addresses of the reference macroblock can be random depending on the type of motion search being performed.

- Allowing only aligned memory accesses (and synthesizing unaligned accesses in software) can potentially perform better than unaligned access implemented in hardware. However, silently accepting an unaligned address and forcing it to be aligned (as in AltiVec) is a bad idea as it can allow alignment errors (typically very difficult to track down because they are intermittent) to go unnoticed. Instead, an exception should be raised when an unaligned access occurs, or hardware should support unaligned memory access directly.

8.2 Bottlenecks and Unnecessary Features

- Instruction primitives (such as the multiplication instruction primitives found in Sun's VIS) are a bad idea, as they decrease instruction decoding bandwidth, increase register pressure, and are not useful in and of themselves. Even if the atomic version of an operation may be slow, it is preferable because it is much easier to upgrade an instruction's latency in the next revision of an architecture than it is to implement entirely new instructions, rendering the previous instructions and any related ones useless.

- Motorola's AltiVec extension includes pixel pack and unpack instructions for converting between 32-bit true color and 16-bit color representations which we did not find useful in the BMKL. Similarly, AltiVec includes approximations for log_2 and exp_2, which also went without application in our workload; they are used in lighting algorithms for 3D rendering.

- Sun's VIS includes edge instructions for dealing with boundaries in 2D image processing, as well as array instructions for volumetric imaging. Neither type of instruction was found to be useful to the Berkeley multimedia workload.

- In general, a singular (unique) resource (such as a control register, or accumulator) is a potential bottleneck if it will be highly utilized. In the case of Sun's VIS graphics status register (GSR), their bottleneck could have been avoided if SIMD shift instructions and better data communication primitives had been included. As it was, the GSR ended up being used to synthesize this missing functionality, beyond its original designed purpose. The MIPS MDMX accumulator register which we briefly discussed is another example of this type of problem.

8.3 New Directions

In addition to analyzing how well current multimedia instruction set features map to multimedia workloads, we also pro-

posed two new directions for multimedia instruction sets.

- Because SIMD architectures apply the same operation to all of the elements in a packed register, there are many cases where data is not optimally organized as loaded from memory. This occurs when working with 2D data types, such as video frames; either row or column processing will not be natively arranged in a way that is amenable to SIMD style processing. Typically, we would like to load M data elements into a vector register, with each element being loaded starting at some base address and separated from each other by a constant byte offset. Similar to scatter-gather operations from traditional vector architectures, we proposed implementing strided loads and stores for packed registers. These are specified in a way that is similar to how prefetch streams are specified in AltiVec.
- Based on the observation that storage and computational data types are almost always different, we proposed a superwide register architecture, which eliminates much of the explicit packing and unpacking overhead that typically makes SIMD processing progress at less than its maximal degree of data parallelism. We found that this fundamental change in how data types are handled had significant implications for instruction set design.

References

[Allen99] Gregory E. Allen, Brian L. Evans, Lizy K. John, "Real-Time High-Throughput Sonar Beamforming Kernels Using Native Signal Processing and Memory Latency Hiding Techniques," *Proc. of the 33rd IEEE Asilomar Conf. on Signals, Systems and Computers*, Pacific Grove, California, October 24-27, 1999, pp. 137-141

[AMDOpt] Advanced Micro Devices, *AMD Athlon Processor x86 Code Optimization Guide*, Publication 22007G/0, April 2000, *http://www.amd.com/products/cpg/athlon/techdocs/index.html*, retrieved April 24, 2000

[AMD00] Advanced Micro Devices, Inc., "AMD Athlon Processor x86 Code Optimization Guide," Publication #22007, Rev. G, April 2000, *http://www.amd.com/products/cpg/athlon/techdocs/pdf/22007.pdf*, retrieved April 24, 2000

[AMD99] Advanced Micro Devices, Inc., "AMD Athlon Processor Technical Brief," Publication #22054, Rev. D, December 1999, *http://www.amd.com/products/cpg/athlon/techdocs/pdf/22054.pdf*, retrieved April 24, 2000

[AMDWP] Advanced Micro Devices, "3DNow! Technology vs. KNI," White Paper, *http://www.amd.com/products/cpg/3dnow/vskni.html*, retrieved April 24, 2000

[Bhar98] R. Bhargava, L. K. John, B. L. Evans, R. Radhakrishnan, "Evaluating MMX Technology Using DSP and Multimedia Applications," *Proc. of the 31st IEEE Intl. Symp. on Microarchitecture (MICRO-31)*, Dallas, Texas, November 30-December 2, 1998, pp. 37-46

[Burd] Tom Burd, "CPU Info Center: General Processor Info," *http://bwrc.eecs.berkeley.edu/CIC/summary/local/summary.pdf*, retrieved April 24, 2000

[Carl97] David A. Carlson, Ruben W. Castelino, Robert O. Mueller, "Multimedia Extensions for a 550-MHz RISC Microprocessor," *IEEE Journal of Solid-State Circuits*, Vol. 32, No. 11, November 1997, pp. 1618-1624

[Chen96] Wilam Chen, H. John Reekie, Sunil Bhave, Edward A. Lee, "Native Signal Processing on the UltraSparc in the Ptolemy Environment," *Proc. of the 30th Annual Asilomar Conf. on Signals, Systems, and Computers*, Pacific Grove, California, November 3-6, 1996, Vol. 2, pp. 1368-1372

[Comp00] Compaq Computer Corporation, "Alpha 21264 Microprocessor Hardware Reference Manual," Part No. DS-0027A-TE, February 2000, *http://www.support.compaq.com/alpha-tools/documentation/current/21264_EV67/ds-0027a-te_21264_hrm.pdf*, retrieved April 24, 2000

[Hans96] Craig Hansen, "MicroUnity's MediaProcessor Architecture," *IEEE Micro*, Vol. 16, No. 4, August 1996, pp. 34-41

[IAOpt] Intel Corporation, *Intel Architecture Optimization Reference Manual*, *http://developer.intel.com/design/pentiumii/manuals*, retrieved April 24, 2000

[IEEE91] IEEE, "IEEE Standard Specifications for the Implementation of 8x8 Inverse Discrete Cosine Transform", *IEEE Standard* 1180-1990, M. T. Sun ed., IEEE, 1991

[Intel97] Intel Corporation, "Intel Introduces The Pentium Processor With MMX Technology," January 8, 1997 Press Release, *http://www.intel.com/pressroom/archive/releases/dp010897.htm*, retrieved April 24, 2000

[Intel99a] Intel Corporation, "Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture," Publication 243190, 1999, *http://developer.intel.com/design/pentiumii /manuals/24319002.PDF*, retrieved April 24, 2000

[Intel00a] Intel Corporation, "IA32 Intel Architecture Software Developer's Manual with Preliminary Intel Pentium 4 Processor Information Volume 1: Basic Architecture," *http://developer.intel.com/design/processor/future/ manuals/24547001.pdf*, retrieved September 26, 2000

[Intel00b] Intel Corporation, "Intel Announces New NetBurst Micro-Architecture for Pentium IV Processor," *http://www.intel.com/pressroom/archive/releases/dp082200.htm*, retrieved September 27, 2000

[Kesh99] Jagannath Keshava, Vladimir Pentkovski, "Pentium III Processor Implementation Tradeoffs," Intel Technology Journal, Quarter 2, 1999, *http://developer.intel.com/technology/itj/q21999/pdf/impliment.pdf*, retrieved April 24, 2000

[Kien99] Tim Kientzle, "Implementing Fast DCTs," *Dr. Dobb's Journal*, Vol. 24, No. 3, March 1999, pp. 115-119

[Kohn95] L. Kohn, G. Maturana, M. Tremblay, A. Prabhu, G. Zyner, "The Visual Instruction Set (VIS) in UltraSPARC," *Proc. of Compcon '95*, San Francisco, California, March 5-9, 1995, pp. 462-469

[Kuro98] Ichiro Kuroda, Takao Nishitani, "Multimedia Processors," *Proc. of the IEEE*, Vol. 86 No. 6, June 1998, pp. 1203-1221

[Lawl92] Patricia K. Lawlis, "Ada Outperforms Assembly: A Case Study," *Proceedings of TRI-Ada '92*, Orlando, Florida, November 16-20, 1992, *http://www.acm.org/sigs/sigada/education/pages/lawlis.html*, retrieved October 20, 2000

[Lee00] Ruby B. Lee, "Subword Permutation Instructions for Two-Dimensional Multimedia Processing in MicroSIMD Architectures," *Proc. of IEEE Intl. Conf. on Application-Specific Systems, Architectures, and Processors*, July 10-12, 2000, Boston, Massachusetts, pg. 3-14

[Lee97c] Ruby B. Lee, "Multimedia Extensions for General Purpose Processors," *Proc. of the IEEE Workshop on VLSI Signal Processing*, Leicester, UK, November 3-5, 1997, pp. 1-15

[MIPS97] MIPS Technologies, Inc., "MIPS Extension for Digital Media with 3D," WhitePaper, March 12, 1997 *http://www.mips.com/Documentation/ isa5_tech_brf.pdf*, retrieved April 24, 2000

[Moto00] Motorola Inc., "MPC7400 RISC Microprocessor User's Manual, Rev. 0," Document MPC7400UM/D, March 27, 2000, *http://www.mot.com/SPS/PowerPC/teksupport /teklibrary/manuals/MPC7400UM.pdf*, retrieved April 24, 2000

[Naka96] Jill Nakashima, Ken Tallman, "The VIS Advantage: Benchmark Results Chart VIS Performance," White Paper, October 1996, *http://www.sun.com/microelectronics/vis/*, retrieved April 24, 2000

[Nguy99] Huy Nguyen, Lizy Kurian John, "Exploiting SIMD Parallelism in DSP and Multimedia Algorithms Using the AltiVec Technology," *Proc. of the 1999 Interational Conf. on Supercomputing*, Rhodes, Greece, June 20-25, 1999, pp. 11-20

[Noer] Kim Noer, "Heat Dissipation Per Square Millimeter Die Size Specifications," *http://home.worldonline.dk/~noer/*, retrieved April 24, 2000

[Norm98] Kevin B. Normoyle, Michael A. Csoppenszky, Allan Tzeng, Timothy P. Johnson, Christopher D. Furman, Jamshid Mostoufi, "UltraSPARC-IIi: Expanding the Boundaries of a System on a Chip," *IEEE Micro*, Vol. 18, No. 2, March/April 1998, pp. 14-24

[Patt96] David A. Patterson, John L. Hennessy, *Computer Architecture: A Quantitative Approach*, Second Edition, 1996 Morgan Kaufman Publishers, Inc.

[Rang99] Parthasarathy Ranganathan, Sarita Adve, Norman P. Jouppi, "Performance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions," *Proc. of the 26th Annual Intl. Symp. on Computer Architecture*, May 2-4, 1999, Atlanta, Georgia, pp. 124-135

[Rath96] Selliah Rathnam, Gert Slavenberg, "An Architectural Overview of the Programmable Multimedia Processor, TM-1," *Proc. of COMPCON '96*, February 25-28, 1996, Santa Clara, California, pp. 319-326

[Rice96] Daniel S. Rice, "High-Performance Image Processing Using Special-Purpose CPU Instructions: The UltraSPARC Visual Instruction Set," University of California at Berkeley, Master's Report, March 19, 1996

[Rubi96] Paul Rubinfeld, Bob Rose, Michael McCallig, "Motion Video Instruction Extensions for Alpha," White Paper, October 18, 1996 *http://www.digital.com/alphaoem/papers/pmvi-abstract.htm*, retrieved April 24, 2000

[Sling00a] Nathan T. Slingerland, Alan Jay Smith, "Design and Characterization of the Berkeley Multimedia Workload," *University of California at Berkeley Technical Report CSD-00-1122*, December, 2000

[Sling00c] Nathan T. Slingerland, Alan Jay Smith, "Multimedia Instruction Sets for General Purpose Microprocessors: A Survey," *University of California at Berkeley Technical Report CSD-00-1125*, December, 2000

[Stil99] Andreas Stiller, "Architecture Contest," *c't Magazine*, Vol. 16/99, *http://www.heise.de/ct/english/99/16/092/*, retrieved December 2, 2000

[Sun97] Sun Microsystems Inc., "UltraSPARC-IIi User's Manual," 1997, Part No. 805-0087-01, *http://www.sun.com/microelectronics/UltraSPARC-IIi/docs/805-0087.pdf*, retrieved April 24, 2000

[Thak99] Shreekant (Ticky) Thakkar, Tom Huff, "The Internet Streaming SIMD Extensions," *Intel Technology Journal*, Q2 1999, *http://developer.intel.com/technology/itj/q21999.htm*, retrieved April 24, 2000

[Trem96b] Marc Tremblay, J.Michael O'Connor, Venkatesh Narayanan, Liang He, "VIS Speeds New Media Processing," *IEEE Micro*, Vol. 16, No. 4, August 1996, pp. 10-20

1 Appendix A

1.1 Workload

Figure 1 depicts a typical 3D rendering pipeline. Rasterization was not part of the kernels studied due to the ubiquity of 3D accelerator cards which offload this from the CPU, and will continue to do so in the foreseeable future. Although we expect that when enough CPU cycles become available, much of the 3D rendering workload will be moved back onto the CPU and done in software (for cost savings), the current trend is moving in the opposite direction. First generation 3D accelerator cards took care of the rasterization stage, but not 3D geometry computations. Current 3D accelerator cards have also taken on the burden of geometry computations, indicating that the growth in complexity of 3D environments is outpacing that of CPU performance, despite the best efforts of multimedia extensions.

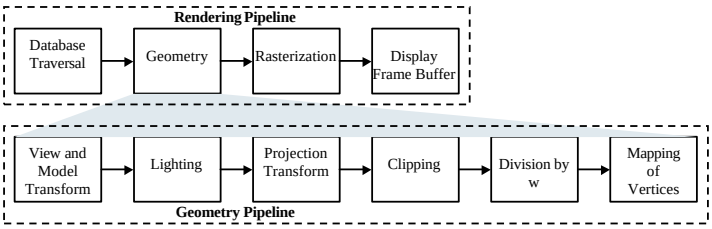


Figure 1: **3D Rendering Pipeline**

1.2 Methodology

1.2.1 Programming with SIMD

Shared Libraries One of the simplest ways to improve application performance through SIMD instructions is to rewrite shared system libraries to utilize them. Existing applications can immediately take advantage of the new instructions without recompilation. Many vendors also make highly optimized SIMD multimedia libraries freely available for incorporation into new applications. Although a library based approach is simple, the restriction of media processing hardware enhancements to system libraries also limits potential performance benefits. In the case of rewriting existing libraries, an application's performance will not improve unless it calls the appropriate system functions. Even if the appropriate functions are used or an application is rewritten to use them, data must be formatted as specified by the API rather than as it might be most efficiently organized for a particular application. [Lee96] The authors of [Bhar98] found that enhancing existing applications with predefined libraries of multimedia procedures was not the optimal way to utilize the MMX instruction set. Applications which are truly able to exploit multimedia instructions often require significant restructuring, as well as hand coding of the precise functions needed. It was found that too often there was a mismatch between the functions available in a library and what the target application required.

Macros *Macros* are high level language "wrappers" which programmers utilize as function calls within their C or C++ code. Each multimedia instruction can be used as if it were a C function call. The primary potential advantage to this approach is that the compiler rather than the developer performs machine specific optimizations such as instruction scheduling and register allocation. The added level of abstraction also has the benefit that if the code needs to be compiled for a platform without the media extensions, the macros can simply be replaced with their high level language equivalents. Despite the fact that macros are used within a high level language, [Chen96] found that programming C applications with multimedia instruction macros was difficult, likening it to typical DSP (assembly) coding. Additionally, contrary to the belief that macros are superior to other programming methods because the compiler can schedule instructions and register allocation, [Allen99] found that upon examining output of the SPARCompiler (v5.0), that instruction scheduling of the expanded macro code was poor. In addition, macros were found to inhibit the compiler's more aggressive optimizations.

Compilers Ideally, high level language compilers would be able to systematically and automatically identify parallelizable sections of code and generate the appropriate SIMD instructions. SIMD optimizations would then not just be limited to multimedia applications, but could be more generally applied to any application exhibiting the appropriate type of data parallelism. Although compilers which can automatically identify and take advantage of situtations where SIMD instructions can be used were until recently mostly limited to experimental and research systems [SUIF], commercial compilers which claim to provide this functionality are now available [Code00],[Intel00].

Subword operations typically support saturating arithmetic, where as standard declarations of integers in high level languages implicitly specify modulo addition. The lack of languages which allow programmers to specify data types and overflow semantics at variable declaration time has hindered the development of compiler support for multimedia instruction sets. Compilers should be able to determine subword groupings on the basis of data parallelism and dependency analysis, but modern dependence analysis techniques are not suited to having multiple quantities in a single register [Cont97]. Nearly all traditional compiler optimizations are based on tracing which values are available and when. [Fish98] suggests that symbolic tracking of arbitrary masked bit patterns within a register is appropriate for partitioned operations.

Optimizations that have been devised for parallel programming often apply in a natural way to SIMD programming. This is demonstrated in [Cheo97], which applies the SUIF vectorizer to generate vector operations. The SUIF (Stanford University Intermediate Format) compiler, developed by the Stanford Compiler Group, is a free infrastructure designed to support collaborative research in optimizing and parallelizing compilers [SUIF]. Because the SUIF vectorizer only performs vectorization on parallel loops, generating infinite length vec-

tor operations, Sun VIS code is generated by strip mining these vector operations into loop iterations of fixed sizes. In order to strip mine a loop for SIMD processing, the loop must be *strictly parallel*, and therefore without any iteration-borne dependencies. In addition, it must be determined how densely the data can be packed. In the realm of multimedia this is not an easy thing to do, as there are often soft boundaries on precision, with the amount of tolerable noise being both difficult to quantify and highly dependent on the type of application. To be coded efficiently, all of the requisite operations within the body of a strip mined loop must be supported for packed data. Data must often be properly arranged and aligned to allow for data parallel processing. Current SIMD instruction set support is largely limited to those widths and operations that the designers expected would be used in targeted multimedia applications. [Luml97]

Assembly Language The efficient programming of microprocessors with multimedia extensions can only be attained if experts tune their software using assembly language, just as in DSP approaches [Kuro98]. Although this method is more tedious and error prone than the other methods that we have looked at, it is available on every platform, and allows for great flexibility and precision when coding. Although many previous studies of multimedia instruction set performance have measured the optimized libraries provided by instruction set vendors, we chose instead to code each kernel for every platform in pure assembly language ourselves. We felt that differences in programmer ability and time spent coding between the vendors' libraries could potentially skew results, and so a single programmer coded all of our kernels for a roughly equal length of time. It was our goal to measure instruction sets, and not the intermediate programmers or tools, leading us to chose direct assembly language coding for the optimized codes examined in our study.

1.2.2 Programming Experience

Numerical Methods In order to program with SIMD instructions it is useful to review low level concepts that are easily overlooked or forgotten when programming in a high level language like C or Java, but are of central importance to a correct implementation in assembly language. There are many heuristics for performing operations quickly in assembly language. Additionally, if a needed operation is required by an algorithm but not supported directly by an instruction set it is unusually possible to synthesize the needed operation. The only costs are a larger number of instructions and potentially greater register pressure and execution dependencies.

The meaning of the 2^N states of an N-bit binary word depends entirely on its interpretation. For an unsigned N-bit $U(a.b)$ format fixed point number, the variable, a, is the number of significant bits to the left of the binary point in an N-bit value, and b is the number of bits to the right. Signed two's complement format numbers are denoted $S(a.b)$, where $a = N - b - 1$ and b is the number bits to the right of the binary point. We discuss the properties of fixed point numbers are discussed in greater detail later. Floating point representations offer greater dynamic range in the same number of bits as natural binary or fixed point representations. This is accomplished by using a format similar to scientific notation in decimal. Bits of precision are traded to extend the range of representable values. Block floating point is a scaling technique in which a single exponent is used for all the data in a block of fixed point integer data. This originated as a software technique (as did floating point), but some digital signal processors (DSPs) support this mode directly in hardware. It has the advantage of being less expensive in terms of hardware than floating point, as well as faster.

Integer Values The meaning assigned to any of the 2^N states of an N-bit binary word depends entirely on its interpretation. Common binary representations include unsigned integers, unsigned fixed-point rationals, signed two's complement integers and signed two's complement fixed-point rationals. Rational numbers are those numbers expressible as a/b where a and b are both members of the set of integers, Z. Like decimal (base 10) representations of rational numbers, binary (base 2) representations assign a weighted power of the respective base to each position of a number. Although we do not usually think of a decimal number as having a limited number of positions, binary numbers on computers have a limited width. Consider an unsigned 8-bit binary fixed point number format, which we will designate $U(6.2)$:

b_5	b_4	b_3	b_2	b_1	b_0	b_{-1}	b_{-2}
1	0	0	0	1	0	1	0

As is conventional with decimal numbers, the above binary number has been written with its most significant bits to the left, and its least significant bits to right. The value of an unsigned N-bit $U(a.b)$ format fixed point number, x, is given by:

$$x = \left(\frac{1}{2}\right)^b \sum_{n=0}^{N-1} 2^n x_n \qquad (1)$$

The variable, a, is the number of significant bits to the left of the binary point in an N-bit value, and b is the number of bits to the right. Like decimal (base 10) arithmetic, binary (base 2) arithmetic contains an implicit binary point indicating the boundary between positions representing powers of the base greater than zero and those signifying powers of the base less than zero. Unsigned integer (or "natural binary") representations are a special case of $U(a.b)$ format fixed point numbers where $b = 0$. Each bit, b_k, has a weight of 2^k, so the value of the above example binary number is 34.5.

Two's complement is a method for representing signed numbers which simplifies the underlying hardware implementation on digital computers. The two's complement of a binary number, x, is given by taking the *one's complement* (negating all the bits) and adding one. We will denote signed two's complement format $S(a.b)$, where $a = N - b - 1$ and b is the number bits to the right of the binary point. The value of an N-bit $S(a.b)$ format number, x, is given by:

$$x = \left(\frac{1}{2}\right)^b \left[-2^{N-1}x_{N-1} + \sum_{n=0}^{N-2} 2^n x_n\right] \quad (2)$$

We will use the notation $X(a.b)$ to note when a rule is applicable to either $U(a.b)$ or $S(a.b)$ format numbers. Fixed point arithmetic has the following fundamental rules [Yates]:

1. Unsigned Wordlength - the number of bits required to represent $U(a.b)$ is $a + b$
2. Signed Wordlength - the number of bits required to represent $S(a.b)$ is $a + b + 1$
3. Unsigned Range - the range of a $U(a.b)$ fixed-point number is $0 \leq x \leq 2^a - 2^{-b}$
4. Signed Range - the range of an $S(a.b)$ fixed-point number is $-2^a \leq x \leq 2^a - 2^{-b}$
5. Addition Operands - the binary points of two numbers must be aligned in order for addition or subtraction to be performed. $X(c.d) + X(e.f)$ is only valid if $c = e$, and $d = f$.
6. Addition Result - the sum of two N-bit binary numbers requires $N + 1$ bits
7. Unsigned Multiplication - $U(a_1.b_1) \times U(a_2.b_2) = U(a_1 + a_2.b_1 + b_2)$
8. Signed Multiplication - $S(a_1.b_1) \times S(a_2.b_2) = S(a_1 + a_2 + 1.b_1 + b_2)$
9. Shifting - a shift can either be considered a scaling operation, moving an entire binary value along with its binary point:

$$X(a.b) \gg n = X(a + n.b - n)$$
$$X(a.b) \ll n = X(a - n.b + n)$$

or a multiplication/division by a power of two:

$$X(a.b) \gg n = X(a - n.b + n)$$
$$X(a.b) \ll n = X(a + n, b - n)$$

Floating Point Floating point representations offer greater dynamic range in the same number of bits as natural binary or fixed point representations. This is accomplished by using a format similar to scientific notation in decimal. Bits of precision are traded to extended the range of representable values. The IEEE 754 floating point standard is used by almost all modern floating point hardware. The formats associated with it are defined in Figure 2.

The IEEE standard assigns the largest and smallest numbers supported by the standard to be $\pm 3.4 \times 10^{38}$, and $\pm 1.2 \times 10^{-38}$ respectively for single precision, leaving some bit patterns free for special values:

1. ± 0 - all of the mantissa bits and exponent bits being 0s
2. $\pm\infty$ - all of the mantissa bits 0s and all of the exponent bits 1s
3. NaN - not a number
4. group of small un-normalized numbers $\pm 1.2 \times 10^{-38}$ to $\pm 1.4 \times 10^{-45}$

Double precision IEEE floating point extends single precision by adding extra bits to the exponent and mantissa extends the range: $\pm 1.8 \times 10^{+308}$ to $\pm 2.2 \times 10^{-308}$.

Block Floating Point Block floating point is a scaling procedure in which a single exponent is used for all the data in a block of data. This originated as a software technique (as did floating point), but some digital signal processors (DSPs) support this mode directly in hardware. It has the advantage of being less expensive in terms of hardware than floating point, as well as faster.

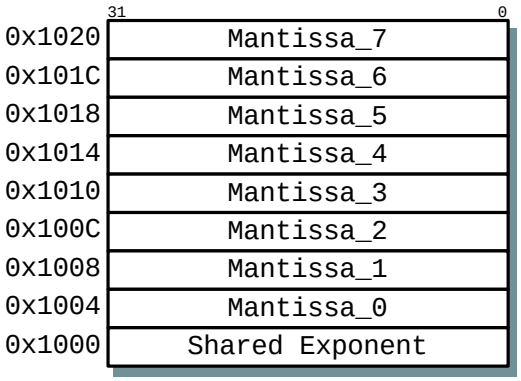
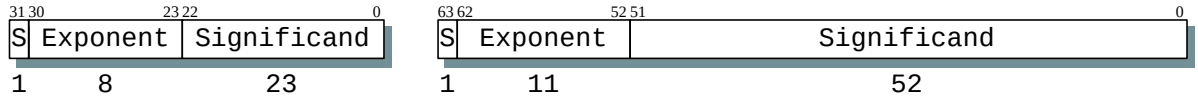


Figure 3: **Example Block Floating Point Memory Map**

To visualize how this works, consider the memory map shown in Figure 3, which diagrams how eight block floating point number and their shared exponent might be laid out in memory. Floating point numbers have a greater dynamic range because the distance between numbers gets larger as the magnitude of the numbers get larger. The disadvantage of block floating point is that all numbers must share the greatest exponent of all the actual values. The mantissas must be scaled to match this shared exponent, but because there is a finite amount of precision in the mantissa, precision can be lost. Block floating point only works well if a block of data can have a wide possible range of values, but values are clustered for a particular computation.

Endianness The memory order of bytes or *endianness* of an architecture is an important consideration for assembly language programming, especially when dealing with packed data. Consider what happens when loading the first eight pixels of an array of 8-bit greyscale image data into a 64-bit multimedia register (Figure 4). The arrows in Figure 4 indicate increasing memory addresses. Image data is normally laid out as a 2D array, with spatially adjacent row elements adjacent in memory. The first element in the array is in the



(a) Single Precision

(b) Double Precision

Figure 2: **IEEE 754 Floating Point Formats**

upper left corner of the image. Machine endianness is important because this determines the order in which packed bytes are placed into registers.

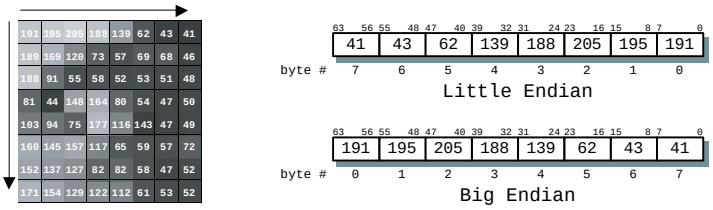


Figure 4: **Effect of Byte Order with Image Data**

Synthesizing Multiply/Shift Multiplication of fixed-point values by integer and fractional constants can be simulated with right/left shift and add operations. Likewise, shifts can be simulated with multiplication and division. None of the multimedia extensions offer partitioned integer divide instructions, so only left shifts can be simulated with current instruction sets. It is also useful to keep in mind that multiplication by a power of 2 on a platform without left shift operations can be decomposed into a series of additions. If the desired power of two is small this can be a performance win if other operations can be done at the same time to combat the added data dependencies. We found this useful on Sun's VIS when bit-wise left shifts were needed.

Synthesizing Absolute Value The absolute value of an $S(a.b)$ integer, X, can be synthesized as follows:

1. $X \longrightarrow X_{pos}$, $0 - X \longrightarrow X_{neg}$
2. $max(X_{neg}, X_{pos}) \longrightarrow X_{pos}$

Floating point absolute value can be performed by AND'ing a single precision value with `0x7FFFFFFF`. This clears the sign bit.

Synthesizing Floating Point Sign Negation Floating point negation can be performed by XOR'ing a single precision value with `0x80000000`. This negates the sign bit.

Newton-Raphson Method The Newton-Raphson formula for finding the root of an equation is defined:

$$x_{i+1} = x_i - f(x_i)/f'(x_i) \tag{3}$$

Utilizing the above method, it is possible to generate equations for increasing the precision of approximations to $1/a$:

$$x_1 = x_0 - (a \cdot x_0^2 - x_0) \tag{4}$$

as well as 1/sqrt(a):

$$x_1 = x_0 - (0.5 \cdot a \cdot x_0^3 - 0.5 \cdot x_0) = 0.5 \cdot x_0 \cdot (3.0 - a \cdot x_0^2) \tag{5}$$

Both the reciprocal and reciprocal operations are common in floating point 3D geometry routines. In the above equations, x_0 represents the first approximation, and x_1 is the iteratively higher precision result. This method approximately doubles the number of significant digits for each iteration if the initial guess is close. [Intel99b]

1.2.3 Performance Counters

Many architectures include a small number (2 - 4) of performance counters. These are generally wide (up to 64-bits) registers which are incremented when user selectable architectural events occur. The types of events vary greatly, and are inherently architecture specific, but can be roughly categorized as follows:

branch prediction - predicted/mispredicted conditional branches

clock cycles - measures a duration in clock cycles

floating point - types/counts of floating point instructions executed

instructions - executed, retired

integer - types/counts of integer instructions executed

interrupts - number of external (hardware) interrupts

L1 I cache - misses, hits, lines in, lines out

L1 D cache - misses, hits, lines in, lines out

L2 cache - misses, hits

memory bus - snoop statistics, bus transaction statistics,

memory controller - memory requests, unaligned data references

memory ordering - memory ordering instructions

SIMD - types/counts of SIMD instructions executed

	AMD Athlon	DEC Alpha 21264A	Intel Pentium III	Motorola G4	Sun UltraSPARC IIi
Counters #(width)	4 (48-bit)	2 (20-bit)	2 (40-bit)	4 (32-bit)	2 (32-bit)
branch predicition	√		√	√	
clock cycles	√	√	√	√	√
floating point			√	√	
instructions	√	√	√	√	√
integer					
interrupts	√		√		
L1 I cache	√		√	√	√
L1 I TLB	√		√	√	
L1 D cache	√		√	√	√
L1 D TLB	√			√	
L2 cache	√	√	√	√	
memory bus	√		√	√	√
memory controller	√		√		
memory ordering	√		√	√	
SIMD			√	√	
stalls	√		√	√	√

Table 1: **Performance Counter Comparison**

stalls - reservation stations full, pipeline stalled, instruction decoder empty

TLB - misses, hits

Table 1 compares the types of events that can be counted on the platforms in this study. Any counter may be able to count any event, or each may cover a subset of those available. Some architectures allow for counting to be toggled based on the system state (user, supervisor) or when a marked process is executing. Counter values indicate the number of times events occur within a section of code, but are not sufficiently accurate to determine precisely which instruction causes what events. They give a general idea of where stalls or other delays are coming from, but because they measure all events the statistics from any profiling code are also necessarily included. Careful measurement of such overheads sometimes makes it possible to at least partially mask such effects. In addition, measurements on processors which feature out of order execution, may include count events caused by instructions preceding the counter read or miss events caused before it, in program order.

All of the microprocessors studied include performance monitoring counters. Although performance counters were sometimes used to guide our optimizations, their primary purpose was to be nearly cycle-accurate timers with which to measure the very short execution times of the kernels in the BMKL.

1.2.4 C Compiler Flags

Table 2 details the specific compiler flags used on each platform to compile the C reference version of the BMKL. Note that in the case of the AMD Athlon, Athlon specific instruction scheduling was not supported by v1.1.3 of PGCC.

1.3 Processors and Instruction Sets

Table 3 contrasts the operations provided by each instruction set. Table 4 compares the SIMD functional units available on each architecture studied. Any overlap with other scalar integer or floating point functionality is noted.

1.4 Analysis

1.4.1 Overall Instruction Mix

Figure 5 depicts the overall instruction mix for each architecture. This data is only for the code within the BMKL kernels, and does not include instructions executed by the rest of each application.

1.4.2 SIMD Operation Types per Kernel

Table 5 lists the SIMD instruction set coverage of each type of functionality for the kernels and architectures studied. As an example of how to read the table, the accelerated (SIMD) Intel Pentium III kernel consists of 19.2% add/subtract integer SIMD instructions (of the total dynamic instruction count for that kernel).

1.4.3 Multimedia Data Types

Table 7 breaks down the distribution of data widths for SIMD instruction on a per-kernel basis. Table 6 lists the overall distribution of SIMD instructions executed based on the data type used by each instruction and categorized according to the type of operation. These counts are for the entire workload, although SIMD instructions are only employed by the optimized kernels in the BMKL. Data types are specified as either sign-independent (e.g. 2's complement addition), (U)nsigned, or (S)igned N-bit numbers.

Processor	Compiler	Flags
AMD Athlon	PGCC v1.1.3	-O6 -march=pentiumpro -finline-functions -ffast-math -mpentiumpro -fomit-frame-pointer
DEC Alpha 21264	Compaq C v6.1-011	-arch ev6 -fast
Intel Pentium III	PGCC v1.1.3	-O6 -march=pentiumpro -finline-functions -ffast-math -mpentiumpro -fomit-frame-pointer
Motorola 7400 (G4)	MacOS X DP3 gcc	-O4 -finline-functions -ffast-math -fomit-frame-pointer
Sun UltraSPARC IIi	Workshop Compilers v5.0	-fast

Table 2: **C Compilers and Optimization Flags**

	Modulo Add/Sub	Saturating Add/Sub	Average	Max/Min	Multiply	Shift	SAD	Compare	1/x	1/sqrt(x)	Shuffle	Merge	Pack	Unpack
AMD Athlon	8,16 32,FP	U8,U16 S8,S16	U8,U16	U8,U16 FP	U16,S16 FP	16,32,64	U8	S8,S16 S32,FP	FP	FP	16	8,16,32	S16,S32 16,32,FP	U8,U16 S16,S32
DEC 21264A				U8,S16			U8						16,32	U8
Intel Pentium III	8,16 32,FP	U8,U16 S8,S16	U8,U16	U8,U16 FP	U16,S16 FP	16,32,64	U8	S8,S16, S32,FP	FP	FP	16 32	8,16,32	S16,S32 16,32,FP	U8,U16 S16,S32
Motorola G4	8,16 32,FP	U8,U16,U32 S8,S16,S32	U8,U16,U32 S8,S16,S32	U8,U16,U32 S8,S16,S32 FP	U8,U16 S8,S16 FP	8,16,32		U8,U16,U32 S8,S16,S32	FP	FP	8	8,16,32	S16,S32 16,32,FP	U8,U16 U32,S32
Sun UltraSPARC IIi	16,32				U8,S16		U8	S16,S32				8	S32	U8,U16

Table 3: **Data Types and Operations Supported** - Data types are specified as ({**U**nsigned, **S**igned}) N-bit integers or single precision (32-bit) floating point (FP). SAD refers to the sum of absolute differences operation.

Processor	SIMD Functional Units - Instruction Classes Executed
AMD Athlon [AMD99][AMD00]	FADD Unit - x87 FP add/sub, MMX Integer add/sub/logic/shift, 3DNow! add/sub/logic FMUL Unit - x87 FP multiply, MMX Integer add/sub/logic/shift/multiply, 3DNow! multiply/$\sim \frac{1}{x}$/$\sim \frac{1}{\sqrt{x}}$ FSTORE Unit - x87 FP load/stores, 3DNow! loads/stores
DEC 21264 [Comp00]	Integer Unit 0 - any integer or MVI instructions Integer Unit 1 - any integer or MVI instructions
Intel Pentium III [Intel99a][Kesh99]	Unit 0 - Integer add/sub/logic/multiply, x87 FP add/sub, MMX add/sub/logic/multiply, SSE multiply/$\sqrt{x}$/divide Unit 1 - MMX add/sub/logic/shift, Intger add/sub/logic/multiply/jump, SSE shuffle/add/sub/logic/$\sim \frac{1}{x}$/$\sim \frac{1}{\sqrt{x}}$ Unit 2 - loads (all types) Unit 3 - stores (all types) Unit 4 - stores (all types)
Motorola G4 [Moto00]	Load/Store Unit (LSU) - all load/store instructions, data transfer between register files Vector Permute Unit (VPU) - pack/unpack/merge/splat/permute/select Vector ALU (VALU) - three subunits, only one new sub-unit instruction can start each cycle: Vector Simple Integer Unit (VSIU) - add/sub/max/min/compare/avg/rotate/shift/logical Vector Complex Integer Unit (VCIU) - multiply/divide/sum across (reduce) Vector Floating Point Unit (VFPU) - all vector floating point operations
Sun UltraSPARC IIi [Norm98][Sun97]	Graphics Multiplier Unit - VIS multiply/compare/pack/pixel distance Graphics Adder Unit - VIS add/sub/data align/merge/expand/logic

Table 4: **Multimedia Functional Units**

		SIMD Integer								SIMD Floating Point							SIMD					Scalar		
		Add/Subtract	Average	Control Flow	Conversion	Max/Min	Multiply	Sum of Absolute Differences	Shift	Add/Subtract	Approximate 1/sqrt(x)	Approximate 1/x	Control Flow	Conversion	Max/Min	Multiply	Control State	Data Communication	Load	Logical	Store	Integer	Floating Point	Other
Add Block	AMD Athlon	14.1%			7.1%												0.9%	18.1%	20.2%	0.9%	7.1%	30.8%		0.9%
	DEC 21264A				9.8%	10.6%																79.2%		0.3%
	Intel Pentium III	14.2%			7.1%												0.9%	18.1%	20.3%	0.9%	7.1%	26.9%		4.5%
	Motorola G4	6.5%			6.5%													30.5%	18.6%	5.5%	12.9%	18.7%		0.8%
	Sun UltraSPARC IIi	17.9%			17.9%													15.2%	25.6%	1.0%	8.9%	12.4%		1.1%
Block Match	AMD Athlon	12.8%	0.4%		0.4%			10.3%	0.7%								0.6%	9.3%	22.7%	1.2%		41.8%		
	DEC 21264A				2.9%			5.1%														92.0%		
	Intel Pentium III	11.9%	0.3%		0.3%			9.6%	0.7%								0.6%	8.7%	21.2%	1.1%		35.8%		9.8%
	Motorola G4	13.7%	0.2%		0.1%	8.5%			0.3%									19.9%	17.9%	1.0%	4.0%	33.9%		0.5%
	Sun UltraSPARC IIi	3.0%		6.6%	4.0%			7.2%										23.0%	23.2%	0.4%	0.4%	31.8%		0.4%
Clip Test	AMD Athlon									4.0%			8.1%			8.1%	0.0%	18.2%	16.3%	18.1%	4.1%	23.1%		
	DEC 21264A																					26.8%	73.2%	
	Intel Pentium III									2.7%			5.4%			5.4%		27.2%	3.2%	16.3%	5.7%	33.9%		
	Motorola G4			0.3%									3.6%			7.3%		22.7%	4.1%	0.8%	14.7%	42.0%	0.7%	3.8%
	Sun UltraSPARC IIi																					29.8%	70.2%	
Color Space	AMD Athlon	12.0%			2.0%		12.0%		14.0%								0.0%	26.0%	22.0%	1.3%	6.0%	4.0%		0.7%
	DEC 21264A				1.7%																	97.9%		0.4%
	Intel Pentium III	12.1%			2.0%		12.1%		13.4%								0.0%	26.2%	22.1%	1.3%	6.0%	4.0%		0.7%
	Motorola G4	21.4%			5.4%		21.4%		14.3%									25.6%	2.4%	0.6%	1.8%	4.8%		2.4%
	Sun UltraSPARC IIi	28.3%			9.4%		28.3%											15.0%	3.1%		2.4%	4.7%		8.7%
DCT	AMD Athlon	19.2%			3.7%		9.6%		9.6%								0.1%	24.1%	19.6%	1.5%	8.0%	4.5%		
	DEC 21264A																					100.0%		
	Intel Pentium III	19.2%			3.7%		9.6%		9.6%								0.1%	24.1%	19.6%	1.5%	8.0%	4.5%		
	Motorola G4	24.5%			3.8%		13.2%		9.4%									30.4%	2.6%		3.5%	12.2%		0.5%
	Sun UltraSPARC IIi	30.1%			5.7%		23.0%											19.0%	9.0%		5.7%	6.9%		0.5%
FFT	AMD Athlon									14.4%						8.9%	0.0%	12.6%	13.1%		12.0%	39.0%		
	DEC 21264A																					42.6%	57.4%	
	Intel Pentium III									11.7%						7.2%	0.0%	16.7%	13.1%		11.4%	40.0%		
	Motorola G4									4.0%						3.2%		17.5%	8.1%	0.0%	12.7%	38.9%	15.6%	0.0%
	Sun UltraSPARC IIi																					53.9%	46.1%	
IDCT	AMD Athlon	19.7%				4.1%	10.0%		12.0%								0.1%	22.0%	19.2%	1.0%	6.6%	5.2%		
	DEC 21264A																					100.0%		
	Intel Pentium III	19.7%				4.1%	10.0%		12.0%								0.1%	22.0%	19.2%	1.0%	6.6%	5.2%		
	Motorola G4	23.9%			3.5%	3.5%	9.6%		8.7%									32.8%	2.6%	0.4%	3.3%	11.3%		0.4%
	Sun UltraSPARC IIi	26.8%		3.3%			12.5%											19.2%	11.7%		10.0%	15.8%		0.6%
Max Val	AMD Athlon	7.9%			7.9%	15.9%											0.2%	0.5%	15.8%	7.9%		36.1%		7.9%
	DEC 21264A					4.2%																91.8%		4.0%
	Intel Pentium III	7.9%			7.9%	16.0%											0.2%	0.5%	15.8%	7.9%		36.0%		7.9%
	Motorola G4					11.6%												23.3%	11.6%		0.2%	53.1%		0.2%
	Sun UltraSPARC IIi	5.8%		11.7%														0.0%	11.9%	0.0%	0.0%	47.2%	23.3%	
Mix Stereo	AMD Athlon	13.9%					13.9%		0.3%								0.3%	14.0%	17.8%		13.9%	25.7%		
	DEC 21264A																					100.0%		
	Intel Pentium III	13.5%					13.5%		0.3%								0.3%	13.5%	17.2%		13.5%	24.9%		3.4%
	Motorola G4	6.1%					6.1%											28.6%	8.6%		6.1%	44.1%		0.3%
	Sun UltraSPARC IIi	14.5%					14.5%											21.9%	18.7%		15.1%	13.5%		1.8%
Quantize	AMD Athlon									3.3%	13.1%			3.3%		22.9%	0.0%	13.1%	3.3%	3.3%	3.3%	21.5%	13.1%	0.0%
	DEC 21264A																					26.6%	71.2%	2.2%
	Intel Pentium III									3.0%	6.0%			6.0%	3.0%	12.0%	0.0%	6.1%	3.1%	3.0%	6.0%	24.7%	24.1%	3.0%
	Motorola G4										6.0%			3.0%		12.0%		3.1%	3.1%	3.0%	12.0%	31.9%	25.4%	0.4%
	Sun UltraSPARC IIi																					55.7%	44.3%	
Short Term Analysis	AMD Athlon	18.9%			3.4%		6.9%		13.7%								0.0%	24.0%	15.5%	0.9%	6.9%	9.8%		
	DEC 21264A																					100.0%		
	Intel Pentium III	18.7%			3.4%		6.8%		13.6%								0.0%	23.8%	15.3%	0.9%	6.8%	9.7%		0.9%
	Motorola G4	23.7%			3.6%		7.3%		7.3%									47.7%	2.1%	0.0%	2.2%	6.0%		0.0%
	Sun UltraSPARC IIi	43.5%			12.2%		32.1%											3.3%	0.9%	0.0%	0.9%	4.0%		3.1%
Short Term Synthesis	AMD Athlon	20.5%			3.7%		7.4%		13.0%								0.0%	24.2%	14.0%	0.0%	6.5%	9.7%		0.9%
	DEC 21264A																					100.0%		
	Intel Pentium III	20.5%			3.7%		7.4%		13.0%								0.0%	24.2%	14.0%	0.0%	6.5%	9.7%		0.9%
	Motorola G4	24.1%			3.7%		7.4%		7.4%									44.9%	2.1%	0.0%	2.3%	8.0%		0.0%
	Sun UltraSPARC IIi	43.2%			12.9%		31.8%											2.5%	0.9%		0.9%	4.7%		3.1%
Subsample Horizontal	AMD Athlon	8.4%			2.4%		8.4%		1.2%								0.0%	9.6%	17.6%	0.0%		51.2%		1.2%
	DEC 21264A																					100.0%		
	Intel Pentium III	8.4%			2.4%		8.4%		1.2%								0.0%	9.6%	17.6%	0.0%	0.0%	51.2%		1.2%
	Motorola G4	10.1%			6.7%		6.7%		3.4%									16.8%	6.7%	0.0%	3.4%	42.8%		3.4%
	Sun UltraSPARC IIi	17.4%			1.2%		17.4%											14.3%	8.7%	4.4%	2.5%	33.5%		0.6%
Subsample Vertical	AMD Athlon	7.8%			1.3%		7.8%		0.6%								0.0%	16.2%	16.2%	0.0%		49.4%		0.6%
	DEC 21264A				14.8%																	84.9%		0.3%
	Intel Pentium III	7.7%			1.3%		7.7%		0.6%								0.0%	16.0%	16.0%	0.0%	0.0%	48.1%		2.6%
	Motorola G4	17.3%			1.1%		8.7%		1.4%									32.5%	8.7%	0.0%	5.8%	24.5%		0.0%
	Sun UltraSPARC IIi	29.8%			1.2%		28.6%											15.5%	7.1%	1.2%	2.4%	14.3%		0.0%
Synthesis Filter	AMD Athlon									18.4%				2.2%		17.4%	0.1%	11.0%	35.3%	1.0%		14.5%		
	DEC 21264A																					10.2%	89.8%	
	Intel Pentium III				2.3%					12.4%				2.3%		9.0%	0.1%	30.8%	18.3%	1.1%		15.0%		8.9%
	Motorola G4				2.6%					8.1%				2.6%		10.5%		30.5%	26.4%	0.5%	2.6%	15.4%		0.7%
	Sun UltraSPARC IIi																					13.8%	86.2%	
Transform	AMD Athlon									9.6%	2.4%		1.2%			20.3%	0.0%	35.9%	16.7%	3.6%	3.6%	6.7%		
	DEC 21264A																					15.1%	84.9%	
	Intel Pentium III									9.5%	1.2%		1.2%			19.1%		37.9%	16.7%	3.6%	3.6%	7.4%		
	Motorola G4										1.4%		1.4%			23.1%		33.0%	5.6%	4.4%	16.7%	14.2%		0.1%
	Sun UltraSPARC IIi																					14.9%	85.1%	
Overall	**AMD Athlon**	**6.8%**	**0.1%**	**0.0%**	**1.1%**	**1.2%**	**2.3%**	**2.6%**	**1.4%**	**4.4%**	**3.3%**	**0.0%**	**0.0%**	**0.9%**	**0.0%**	**2.3%**	**0.2%**	**12.9%**	**15.8%**	**1.8%**	**4.6%**	**34.3%**	**3.3%**	**0.7%**
	DEC 21264A				**1.2%**	**0.4%**		**1.5%**														**67.0%**	**29.2%**	**0.8%**
	Intel Pentium III	**7.8%**	**0.1%**	**0.0%**	**1.3%**	**1.4%**	**2.6%**	**3.0%**	**1.6%**	**2.6%**	**0.9%**	**0.0%**	**0.0%**	**1.0%**	**0.5%**	**3.2%**	**0.2%**	**11.8%**	**15.8%**	**1.6%**	**4.0%**	**32.1%**	**3.7%**	**4.8%**
	Motorola G4	**5.6%**	**0.0%**	**0.0%**	**0.5%**	**3.8%**	**1.1%**		**0.7%**	**1.1%**	**1.4%**	**0.0%**	**0.0%**	**0.7%**	**0.0%**	**3.7%**	**0.0%**	**16.8%**	**10.2%**	**1.0%**	**7.6%**	**36.0%**	**9.6%**	**0.4%**
	Sun UltraSPARC I	**6.0%**		**3.5%**	**1.7%**		**3.6%**	**2.0%**										**9.7%**	**10.2%**	**0.3%**	**1.5%**	**38.5%**	**22.6%**	**0.4%**

Table 5: **Multimedia Functionality** - each square lists the percentage of dynamic instructions a given type of SIMD operation represented for each kernel. Kernel names are given in the left most column, which each kernel subdivided into rows of results for each of the five platforms studied. A grey square indicates functionality not avalible with a given instruction set, while a white square indicates an available operation which was not utilized.

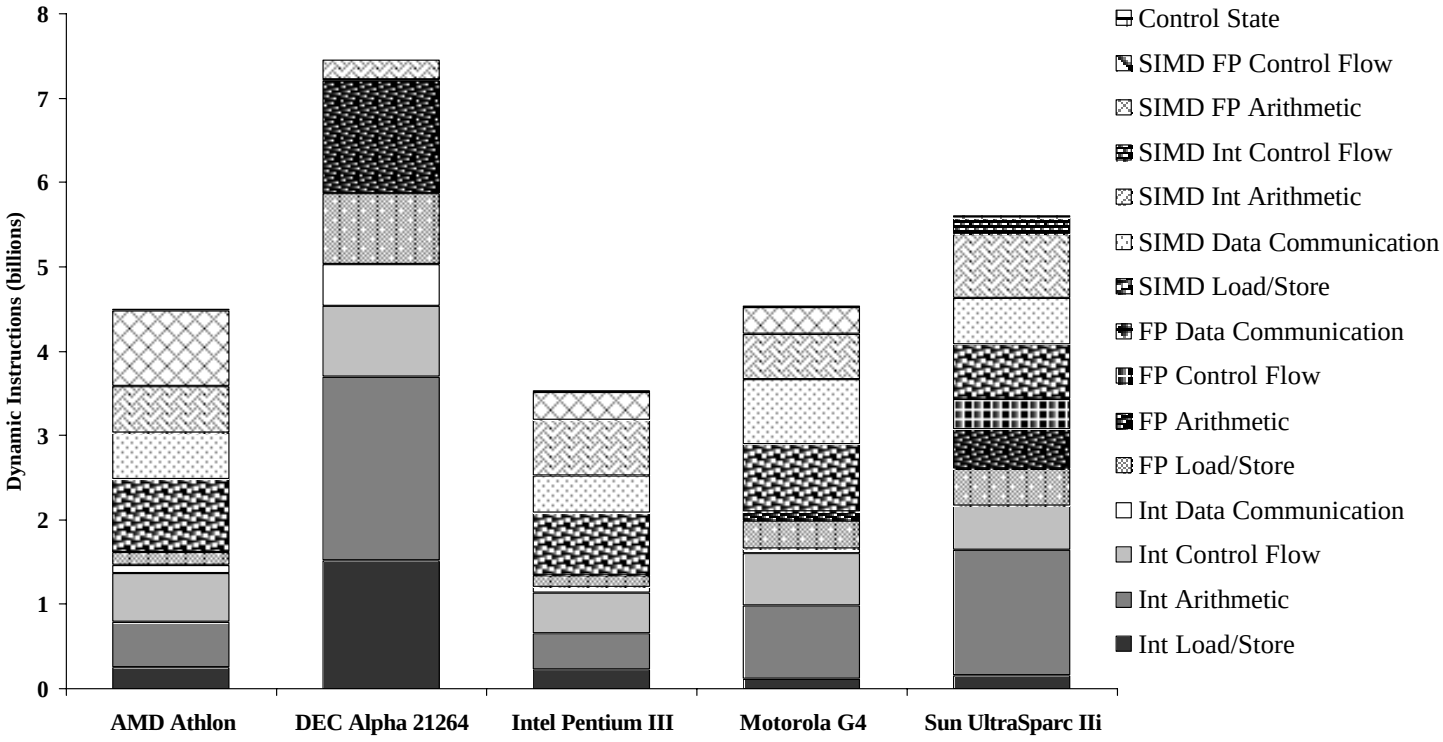


Figure 5: **Overall Instruction Mix**

1.4.4 Instruction Set Coverage

Figure 6 compares the coverage of the SIMD instructions executed by each architecture within the BMKL code.

1.5 More Kernels

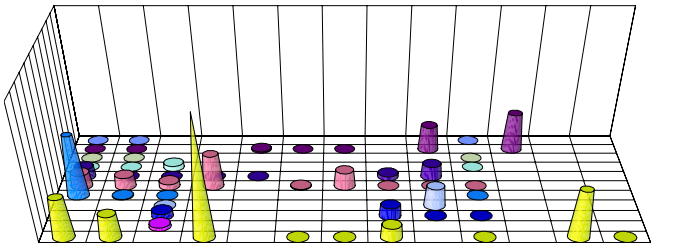
In the next section we use our experience coding each of the sixteen kernels with five different multimedia extensions to determine: 1) existing architectural features that are useful, 2) features that have been implemented, but don't appear to be useful, and 3) significant bottlenecks in current multimedia architectures. These kernels are only those which are not discussed in the main body of the paper. Illustrating our discussion are code fragments both from the original C source code of each kernel algorithm, as well as the different SIMD implementations. The code fragments consist of a few of the key central lines of code from a given kernel. This gives an idea about the types of operations and data types used. The data types of all of the variables in our sample C code are specified in a platform independent way such that the prefix indicates the type: INT: signed integer, UINT: unsigned integer, FP: floating point, followed by N, the number of bits. The complete original C source code for each kernel can be found in Appendix B. Source code for the SIMD implementations of the BMKL are available on the web at *http://www.cs.berkeley.edu/~slingn/research/*.

Add Block The add block kernel boils down to adding a signed 16-bit value with an unsigned 8-bit value, and then clipping the result to an 8-bit unsigned value. With the exceptions of DEC's MVI and Sun's VIS, the other multimedia extensions include saturating addition operations. In the case of VIS, the only way to saturate values is during packing. Fortunately, the add block kernel needs to pack the intermediate 16-bit result to 8-bits again, so this limitation of VIS ends up not costing any additional instructions relative to the other instruction sets.

Algorithm 1 VIS Add Block Fragment

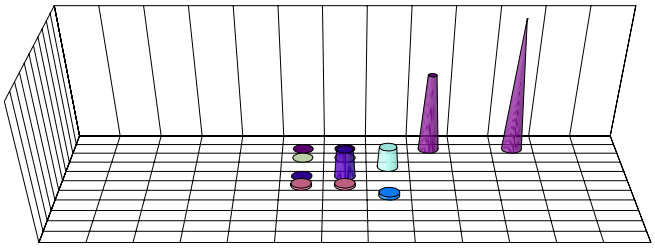
```
wr      %g0, 0x38, %gsr  !  set scale factor for packing (8-bit, no
fraction)
fzero   %f20
ldd     [%o1], %f4       !  %f4:  |r0|r1|r2|r3|r4|r5|r6|r7|
ldd     [%o0 + 0], %f0   !  %f0:  | bp0 | bp1 | bp2 | bp3 |
ldd     [%o0 + 8], %f2   !  %f2:  | bp4 | bp5 | bp6 | bp7 |
fpmerge %f20, %f4, %f8
fpmerge %f20, %f5, %f6
fpadd16 %f0, %f8, %f0    !  %f0:  | r0' | r1' | r2' | r3' |
fpadd16 %f2, %f6, %f2    !  %f2:  | r4' | r5' | r6' | r7' |
fpack16 %f0, %f0         !  %f0:  |r0|r1|r2|r3|     X     |
fpack16 %f2, %f1         !  %f0:  |r0|r1|r2|r3|r4|r5|r6|r7|
std     %f0, [%o1]       !  store new rfp values back to memory
```

Color Space Conversion Although the original color space conversion code is implemented in single precision floating point, the calculation of the final 8-bit pixels values can be done with sufficient precision with fixed point integers. Fixed point multiplication is always done in three steps: 0) pre-shift operands as necessary so that they have the same binary point 1) multiply 2) round by adding fixed point value representing 0.5 3) arithmetically shift result right the number of fraction bits. On many platforms, especially where partitioned (SIMD) floating point operations are not available, being able to efficiently perform fixed point operations



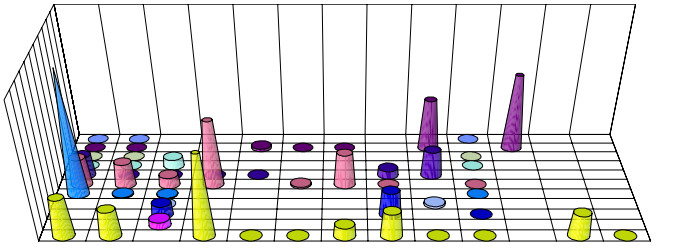
	add	subtract	shift	multiply	average	minimum	maximum	demote	promote	compare	sad	rsqrt	recip
8	0.00%	0.00%								0.00%			
U8	0.00%	0.00%			0.26%	0.00%	0.00%		4.88%		7.25%		
S8	0.00%	0.00%								0.00%			
16	0.00%	0.00%	0.92%							0.00%			
U16	2.03%	0.00%	0.00%	0.00%	0.00%			0.73%	2.52%				
S16	2.69%	2.24%	1.04%	6.28%		0.13%	3.14%	0.00%	0.00%	0.00%			
32	11.71%	0.10%	0.03%							0.00%			
U32			0.00%						3.83%				
S32			1.14%					2.20%	0.00%	0.00%			
64			0.65%										
FP	7.61%	4.55%		22.50%		0.00%	0.00%	2.43%		0.02%		9.11%	0.00%

(a) AMD Athlon



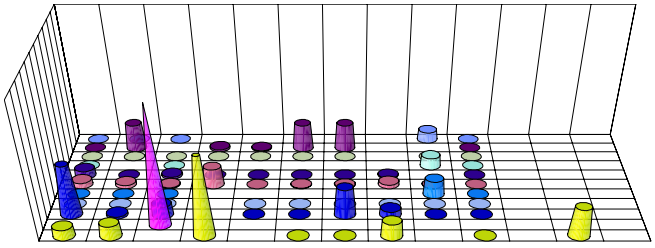
	add	subtract	shift	multiply	average	minimum	maximum	demote	promote	compare	sad	rsqrt	recip
8													
U8						0.00%	0.00%		29.05%		49.39%		
S8						0.00%	0.00%						
16								7.93%					
U16						0.00%	10.58%						
S16						0.91%	0.91%						
32								1.22%					
U32													
S32													
64													
FP													

(b) DEC Alpha 21264



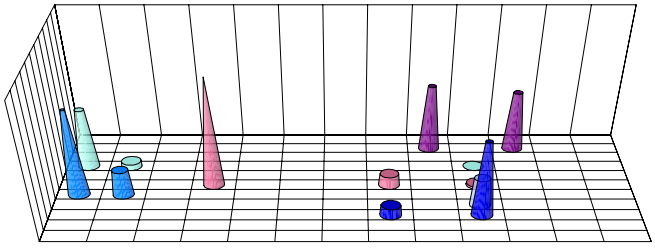
	add	subtract	shift	multiply	average	minimum	maximum	demote	promote	compare	sad	rsqrt	recip
8	0.00%	0.00%								0.00%			
U8	0.00%	0.00%			0.37%	0.00%	0.00%		6.96%		10.32%		
S8	0.00%	0.00%								0.00%			
16	0.00%	0.00%	1.31%							0.00%			
U16	2.90%	0.00%	0.00%	0.00%	0.00%			1.04%	3.60%				
S16	3.84%	3.20%	1.42%	8.96%		0.19%	4.49%	0.00%		0.00%			
32	16.70%	0.14%	0.05%							0.00%			
U32			0.00%						0.29%				
S32			1.63%					3.36%		0.00%			
64			0.92%										
FP	5.24%	3.70%		11.10%	0.00%	0.00%	1.60%	3.43%	0.00%	0.01%		3.21%	0.00%

(c) Intel Pentium III



	add	subtract	shift	multiply	average	minimum	maximum	demote	promote	compare	sad	rsqrt	recip
8	0.00%	0.00%	0.00%						2.01%	0.00%			
U8	0.00%	5.03%		0.36%	0.18%	5.03%	5.03%			0.00%			
S8	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%		0.00%	0.00%			
16	0.00%		0.00%						2.15%	0.00%			
U16	1.41%	0.00%	0.35%	0.00%	0.00%	0.00%	0.00%	0.18%		0.00%			
S16	0.99%	0.60%	0.19%	3.58%	0.00%	0.09%	0.09%	0.69%	0.02%	0.00%			
32	0.00%	0.00%	0.00%						3.12%	0.00%			
U32	0.00%	0.00%	0.00%		0.00%	0.00%	0.00%	0.00%	0.00%	0.00%			
S32	9.61%	0.38%	2.17%		0.00%	0.00%	5.34%	1.28%	0.00%	0.00%			
64			22.14%										
FP	1.91%	2.55%		14.99%		0.00%	0.00%	2.93%		0.01%		5.59%	

(d) Motorola G4



	add	subtract	shift	multiply	average	minimum	maximum	demote	promote	compare	sad	rsqrt	recip
8													
U8									12.33%		11.10%		
S8													
16	11.05%	1.02%								0.00%			
U16													
S16				20.18%				2.19%		0.41%			
32	16.04%	4.81%											
U32										5.15%			
S32								1.86%		13.86%			
64													
FP													

(e) Sun UltraSPARC IIi

Figure 6: **Instruction Set Coverage** - each square lists the percentage of dynamic instructions a given type of SIMD operation represented for the BMKL code on each platform. Data types are listed in the loft most column, as N-bit {sign independent, (U)nsigned, (S)igned, (F)loating (P)oint}. A white square indicates functionality not avalible with a given instruction set.

		Sign Independent				Unsigned			Signed			
		8	16	32	64	U8	U16	U32	S8	S16	S32	FP
Add	AMD Athlon			11.7%			2.0%			2.7%		7.6%
	DEC 21264A											
	Intel Pentium III			16.7%			2.9%			3.8%		5.2%
	Motorola G4						1.4%			1.0%	9.6%	1.9%
	Sun UltraSPARC IIi		11.0%	16.0%								
Subtract	AMD Athlon			0.1%						2.2%		4.5%
	DEC 21264A											
	Intel Pentium III			0.1%						3.2%		3.7%
	Motorola G4					5.0%				0.6%	0.4%	2.5%
	Sun UltraSPARC IIi		1.0%	4.8%								
Shift	AMD Athlon		0.9%	0.0%	0.6%					1.0%	1.1%	
	DEC 21264A											
	Intel Pentium III		1.3%	0.0%	0.9%					1.4%	1.6%	
	Motorola G4				22.1%		0.4%			0.2%	2.2%	
	Sun UltraSPARC IIi											
Multiply	AMD Athlon									6.3%		22.5%
	DEC 21264A											
	Intel Pentium III									9.0%		11.1%
	Motorola G4					0.4%				3.6%		15.0%
	Sun UltraSPARC IIi									20.2%		
Average	AMD Athlon					0.3%						
	DEC 21264A											
	Intel Pentium III					0.4%						
	Motorola G4					0.2%						
	Sun UltraSPARC IIi											
Minimum	AMD Athlon									0.1%		
	DEC 21264A									0.9%		
	Intel Pentium III									0.2%		
	Motorola G4					5.0%				0.1%		
	Sun UltraSPARC IIi											
Maximum	AMD Athlon									3.1%		
	DEC 21264A						10.6%			0.9%		
	Intel Pentium III									4.5%		1.6%
	Motorola G4					5.0%				0.1%	5.3%	
	Sun UltraSPARC IIi											
Demote	AMD Athlon						0.7%				2.2%	2.4%
	DEC 21264A		7.9%	1.2%								
	Intel Pentium III						1.0%				3.4%	3.4%
	Motorola G4						0.2%			0.7%	1.3%	2.9%
	Sun UltraSPARC IIi									2.2%	1.9%	
Promote	AMD Athlon					4.9%	2.5%	3.8%				
	DEC 21264A					29.1%						
	Intel Pentium III					7.0%	3.6%	0.3%				
	Motorola G4	2.0%	2.1%	3.1%						0.0%		
	Sun UltraSPARC IIi					12.3%						
Compare	AMD Athlon											0.0%
	DEC 21264A											
	Intel Pentium III											0.0%
	Motorola G4										0.0%	0.0%
	Sun UltraSPARC IIi							5.2%		0.4%	13.9%	
SAD	AMD Athlon					7.2%						
	DEC 21264A					49.4%						
	Intel Pentium III					10.3%						
	Motorola G4											
	Sun UltraSPARC IIi					11.1%						
~1/sqrt(x)	AMD Athlon											9.1%
	DEC 21264A											
	Intel Pentium III											3.2%
	Motorola G4											5.6%
	Sun UltraSPARC IIi											
~1/x	AMD Athlon											
	DEC 21264A											
	Intel Pentium III											
	Motorola G4											
	Sun UltraSPARC IIi											

Table 6: **Multimedia Data Types: Operations** - each square lists the percentage of dynamic SIMD instructions executed in the Berkeley multimedia kernel library (executed as part of the Berkeley multimedia workload) for each data type. A grey square indicates functionality not available with a given instruction set, while a white square indicates an available operation which was not utilized

		8	16	32	64	FP
Add Block	AMD Athlon	36.3%	63.7%			
	DEC 21264A	22.1%	77.9%			
	Intel Pentium III	36.3%	63.7%			
	Motorola G4	22.2%	51.9%		25.9%	
	Sun UltraSPARC IIi	29.9%	70.1%			
Block Match	AMD Athlon	49.4%	14.4%	36.2%		
	DEC 21264A	90.9%	9.1%			
	Intel Pentium III	77.3%	22.7%			
	Motorola G4	48.0%	5.6%	13.7%	32.7%	
	Sun UltraSPARC IIi	48.8%	19.4%	31.8%		
Clip Test	AMD Athlon			9.4%		90.6%
	DEC 21264A					
	Intel Pentium III					100.0%
	Motorola G4			2.5%	4.1%	93.4%
	Sun UltraSPARC IIi					
Color Space	AMD Athlon	13.0%	71.4%	3.9%	11.7%	
	DEC 21264A			100.0%		
	Intel Pentium III	12.7%	68.4%	7.6%	11.4%	
	Motorola G4	5.4%	24.1%	69.6%	0.9%	
	Sun UltraSPARC IIi	15.2%	84.8%			
DCT	AMD Athlon		56.7%	41.8%	1.5%	
	DEC 21264A					
	Intel Pentium III		70.4%	27.8%	1.9%	
	Motorola G4		52.3%	44.2%	3.6%	
	Sun UltraSPARC IIi	12.8%	48.9%	38.3%		
FFT	AMD Athlon			8.4%		91.6%
	DEC 21264A					
	Intel Pentium III					100.0%
	Motorola G4			7.8%	48.9%	43.3%
	Sun UltraSPARC IIi					
IDCT	AMD Athlon		97.9%	2.1%		
	DEC 21264A					
	Intel Pentium III		96.0%	4.0%		
	Motorola G4	0.6%	53.4%	42.5%	3.4%	
	Sun UltraSPARC IIi	19.0%	81.0%			
Max Val	AMD Athlon		75.1%	24.9%		
	DEC 21264A		100.0%			
	Intel Pentium III		75.1%	24.9%		
	Motorola G4			94.0%	6.0%	
	Sun UltraSPARC IIi			100.0%		
Mix Stereo	AMD Athlon		49.4%	50.6%		
	DEC 21264A					
	Intel Pentium III		100.0%			
	Motorola G4		31.2%	41.6%	27.1%	
	Sun UltraSPARC IIi	27.3%	36.4%	36.4%		
Quantize	AMD Athlon			7.2%		92.8%
	DEC 21264A					
	Intel Pentium III					100.0%
	Motorola G4				0.2%	99.8%
	Sun UltraSPARC IIi					
Short Term A	AMD Athlon		57.6%	30.3%	12.1%	
	DEC 21264A					
	Intel Pentium III		65.5%	20.7%	13.8%	
	Motorola G4		34.1%	36.7%	29.1%	
	Sun UltraSPARC IIi		36.5%	63.5%		
Short Term S	AMD Athlon		59.4%	31.3%	9.4%	
	DEC 21264A					
	Intel Pentium III		67.9%	21.4%	10.7%	
	Motorola G4		36.0%	38.8%	25.2%	
	Sun UltraSPARC IIi		36.2%	63.8%		
Subsample H	AMD Athlon	28.3%	33.8%	38.0%		
	DEC 21264A					
	Intel Pentium III	40.1%	47.9%	12.0%		
	Motorola G4	10.0%	30.0%	50.0%	10.0%	
	Sun UltraSPARC IIi		48.3%	51.7%		
Subsample V	AMD Athlon	23.5%	49.0%	27.5%		
	DEC 21264A	100.0%				
	Intel Pentium III	30.8%	64.1%	5.1%		
	Motorola G4	16.8%	35.7%	37.8%	9.8%	
	Sun UltraSPARC IIi	10.7%	42.9%	46.4%		
Synthesis Filt	AMD Athlon			1.4%		98.6%
	DEC 21264A					
	Intel Pentium III			8.8%		91.2%
	Motorola G4			9.6%	12.9%	77.5%
	Sun UltraSPARC IIi					
Transform	AMD Athlon			40.4%		59.6%
	DEC 21264A					
	Intel Pentium III					100.0%
	Motorola G4				1.5%	98.5%
	Sun UltraSPARC IIi					

Table 7: **Multimedia Data Types: Kernels** - each square lists the percentage of dynamic SIMD instructions in each kernel executed for each data type. A grey square indicates a kernel which was not implemented due to lacking instruction set functionality, while a white square indicates an available operation which was not utilized

is critical.

Algorithm 2 lists the code fragment from the original AltiVec color space conversion kernel which corresponds to a modified version using strided load and store instructions in the main body of this paper.

Algorithm 2 AltiVec Color Space Conversion

```
PERM_0: .byte   0x00, 0x03, 0x06, 0x09, 0x0C, 0x0F, 0x12, 0x15,
                0x18, 0x1B, 0x1E, 0x00, 0x00, 0x00, 0x00, 0x00
PERM_1: .byte   0x01, 0x04, 0x07, 0x0A, 0x0D, 0x10, 0x13, 0x16,
                0x19, 0x1C, 0x1F, 0x00, 0x00, 0x00, 0x00, 0x00
PERM_2: .byte   0x02, 0x05, 0x08, 0x0B, 0x0E, 0x11, 0x14, 0x17,
                0x1A, 0x1D, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
PERM_3: .byte   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                0x08, 0x09, 0x0A, 0x11, 0x14, 0x17, 0x1A, 0x1D
PERM_4: .byte   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                0x08, 0x09, 0x0A, 0x12, 0x15, 0x18, 0x1B, 0x1E
PERM_5: .byte   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                0x08, 0x09, 0x10, 0x13, 0x16, 0x19, 0x1C, 0x1F
rgb_to_yuv:
li      r11, 16         ; immediate for unaligned vector loads
;; Set up Vector Permutations
li32  r12, PERM_0
lvsl  v31, 0, r12     ; v31: vector alignment mask for vperm
lvx   v19, 0, r12     ; v19: vector MSQ
lvx   v20, r11, r12   ; v20: vector LSQ
addi  r12, r12, 16
lvx   v21, r11, r12   ; v21: vector LSQ
addi  r12, r12, 16
lvx   v22, r11, r12   ; v22: vector LSQ
addi  r12, r12, 16
lvx   v23, r11, r12   ; v23: vector LSQ
addi  r12, r12, 16
lvx   v24, r11, r12   ; v24: vector LSQ
addi  r12, r12, 16
lvx   v16, r11, r12   ; v16: vector LSQ
addi  r12, r12, 16
vperm v19, v19, v20, v31 ; v19: red permute vector #1
vperm v20, v20, v21, v31 ; v20: green permute vector #1
vperm v21, v21, v22, v31 ; v21: blue permute vector #1
vperm v22, v22, v23, v31 ; v22: red permute vector #2
vperm v23, v23, v24, v31 ; v23: green permute vector #2
vperm v24, v24, v16, v31 ; v24: blue permute vector #2
;; Convert from Band-Interleaved to Band-Separated Format
lvsl  v31, 0, r3          ; v31: vector alignment mask for vperm
lvx   v25, 0, r3          ; v25: vector MSQ
lvx   v26, r11, r3        ; v26: vector LSQ
addi  r3, r3, 16
lvx   v27, r11, r3        ; v27: vector LSQ
addi  r3, r3, 16
lvx   v28, r11, r3        ; v28: vector LSQ
addi  r3, r3, 16
vperm v25, v25, v26, v31
;; v25: |r0|g0|b0|r1|g1|b1|r2|g2|b2|r3|g3|b3|r4|g4|b4|r5|
vperm v26, v26, v27, v31
;; v26: |g5|b5|r6|g6|b6|r7|g7|b7|r8|g8|b8|r9|g9|b9|rA|gA|
vperm v27, v27, v28, v31
;; v27: |bA|rB|gB|bB|rC|gC|bC|rD|gD|bD|rE|gE|bE|rF|gF|bF|
vperm v28, v25, v26, v19
;; v28: |r0|r1|r2|r3|r4|r5|r6|r7|r8|r9|rA|XX|XX|XX|XX|XX|
vperm v29, v25, v26, v20
;; v29: |g0|g1|g2|g3|g4|g5|g6|g7|g8|g9|gA|XX|XX|XX|XX|XX|
vperm v30, v25, v26, v21
;; v30: |b0|b1|b2|b3|b4|b5|b6|b7|b8|b9|XX|XX|XX|XX|XX|XX|
vperm v28, v28, v27, v22
;; v28: |r0|r1|r2|r3|r4|r5|r6|r7|r8|r9|rA|rB|rC|rD|rE|rF|
vperm v29, v29, v27, v23
;; v29: |g0|g1|g2|g3|g4|g5|g6|g7|g8|g9|gA|gB|gC|gD|gE|gF|
vperm v30, v30, v27, v24
;; v30: |b0|b1|b2|b3|b4|b5|b6|b7|b8|b9|bA|bB|bC|bD|bE|bF|
```

FFT The fast Fourier transform (FFT) is an efficient means for computing the discrete Fourier transform (DFT) on sampled data. The FFT algorithm utilized by the LAME mp3 encoder application is a split-radix type which is computed recursively in single precision floating point. A few sample lines of the central portion of the recursive code are listed in Algorithm 3.

Algorithm 3 Fast Fourier Transform (FFT) - all variables are FLOAT32

```
  xr1 = xr + m2;  xr2 = xr1 + m4;
  xi1 = xi + m2;  xi2 = xi1+ m4;
  for (n = 0; n < m4; n++){
    tmp1 = *xr1 + *xi2;
    tmp2 = *xi1 + *xr2;
    *xi1 = *xi1 - *xr2;
    *xr2 = *xr1 - *xi2;
    *xr1 = tmp1;
    *xi2 = tmp2;
    xr1++; xr2++; xi1++; xi2++;
  }
```

Unlike the original FFT kernel, which is a split radix real FFT (only real input data is provided, although the output is complex), we implemented SIMD versions of the FFT with a more traditional complex radix-2 decimation in frequency algorithm. Although this algorithm would be slower if implemented in scalar code, it is much more amenable to SIMD processing due to its more regular data accesses and much more straight forward parallelization. The original real-value split radix algorithm is highly data dependent, taking advantage of known symmetries when operating on only real input data. SIMD processing is most effective when there are no instruction stream data dependencies.

One thing that became apparent when programming SIMD versions of the FFT on different architectures is that longer vectors require more sophisticated data communication operations to support them. Consider the code snippets listed in Algorithm 4 and Algorithm 5, both of which rearrange the data as read from memory (typically in an FFT algorithm, the real and imaginary components of a given data element are stored adjacent in memory) such that one register contains all of the real components and another all of the corresponding imaginary values. The longer register width of Intel's SSE meant that a simple data communication operation such as merge (`punpckldq`/`punpckhdq`) which are used in AMD's version of the kernel, are not sufficient - full shuffles (`shufps`) are required instead.

Algorithm 4 AMD FFT

```
mov       edx, [w_ptr]
movq      mm6, [edx]     ; mm6:  |  Q0.i  |  Q0.r  |
movq      mm5, [edx]     ; mm5:  |  Q1.i  |  Q1.r  |
add       edx, [w_index]
movq      mm7, mm6
punpckldq mm6, mm5       ; mm6:  |  Q1.r  |  Q0.r  |
punpckhdq mm7, mm5       ; mm7:  |  Q1.i  |  Q0.i  |
```

Algorithm 5 Intel FFT

```
mov       edx, [w_ptr]
movlps    xmm6, [edx]           ; xmm6: |XXXXXX|XXXXXX| Q0.i | Q0.r |
add       edx, [w_index]
movhps    xmm6, [edx]           ; xmm6: | Q1.i | Q1.r | Q0.i | Q0.r |
add       edx, [w_index]
movlps    xmm5, [edx]           ; xmm5: |XXXXXX|XXXXXX| Q2.i | Q2.r |
add       edx, [w_index]
movhps    xmm5, [edx]           ; xmm5: | Q3.i | Q3.r | Q2.i | Q2.r |
shufps    xmm6, xmm6, 11011000b ; xmm6: | Q1.i | Q0.i | Q1.r | Q0.r |
shufps    xmm5, xmm5, 11011000b ; xmm5: | Q3.i | Q2.i | Q3.r | Q2.r |
movaps    xmm7, xmm6
movlhps   xmm6, xmm5            ; xmm6: | Q3.r | Q2.r | Q1.r | Q0.r |
shufps    xmm7, xmm5, 11101110b ; xmm7: | Q3.i | Q2.i | Q1.i | Q0.i |
```

DCT/IDCT As an example of where data communication operations are useful, consider Algorithm 6, which is taken from the Motorola AltiVec DCT kernel. Simple merge operations are completely sufficient for computing an 8x8 matrix transpose.

Algorithm 6 AltiVec Matrix Transpose - matrix elements are 16-bits wide

```
;; v8:   |0_0|0_1|0_2|0_3|0_4|0_5|0_6|0_7|
;; v9:   |1_0|1_1|1_2|1_3|1_4|1_5|1_6|1_7|
;; v10:  |2_0|2_1|2_2|2_3|2_4|2_5|2_6|2_7|
;; v11:  |3_0|3_1|3_2|3_3|3_4|3_5|3_6|3_7|
;; v12:  |4_0|4_1|4_2|4_3|4_4|4_5|4_6|4_7|
;; v13:  |5_0|5_1|5_2|5_3|5_4|5_5|5_6|5_7|
;; v14:  |6_0|6_1|6_2|6_3|6_4|6_5|6_6|6_7|
;; v15:  |7_0|7_1|7_2|7_3|7_4|7_5|7_6|7_7|
vmrghh v0, v8, v12   ; v0:  |0_0|4_0|0_1|4_1|0_2|4_2|0_3|4_3|
vmrglh v1, v8, v12   ; v1:  |0_4|4_4|0_5|4_5|0_6|4_6|0_7|4_7|
vmrghh v2, v9, v13   ; v2:  |1_0|5_0|1_1|5_1|1_2|5_2|1_3|5_3|
vmrglh v3, v9, v13   ; v3:  |1_4|5_4|1_5|5_5|1_6|5_6|1_7|5_7|
vmrghh v4, v10, v14  ; v4:  |2_0|6_0|2_1|6_1|2_2|6_2|2_3|6_3|
vmrglh v5, v10, v14  ; v5:  |2_4|6_4|2_5|6_5|2_6|6_6|2_7|6_7|
vmrghh v6, v11, v15  ; v6:  |3_0|7_0|3_1|7_1|3_2|7_2|3_3|7_3|
vmrglh v7, v11, v15  ; v7:  |3_4|7_4|3_5|7_5|3_6|7_6|3_7|7_7|

vmrghh v8, v0, v4    ; v8:  |0_0|2_0|4_0|6_0|0_1|2_1|4_1|6_1|
vmrglh v9, v0, v4    ; v9:  |0_2|2_2|4_2|6_2|0_3|2_3|4_3|6_3|
vmrghh v10, v1, v5   ; v10: |0_4|2_4|4_4|6_4|0_5|2_5|4_5|6_5|
vmrglh v11, v1, v5   ; v11: |0_6|2_6|4_6|6_6|0_7|2_7|4_7|6_7|
vmrghh v12, v2, v6   ; v12: |1_0|3_0|5_0|7_0|1_1|3_1|5_1|7_1|
vmrglh v13, v2, v6   ; v13: |1_2|3_2|5_2|7_2|1_3|3_3|5_3|7_3|
vmrghh v14, v3, v7   ; v14: |1_4|3_4|5_4|7_4|1_5|3_5|5_1|7_5|
vmrglh v15, v3, v7   ; v15: |1_6|3_6|5_6|7_6|1_7|3_7|5_7|7_7|

vmrghh v0, v8, v12   ; v0:  |0_0|1_0|2_0|3_0|4_0|5_0|6_0|7_0|
vmrglh v1, v8, v12   ; v1:  |0_1|1_1|2_1|3_1|4_1|5_1|6_1|7_1|
vmrghh v2, v9, v13   ; v2:  |0_2|1_2|2_2|3_2|4_2|5_2|6_2|7_2|
vmrglh v3, v9, v13   ; v3:  |0_3|1_3|2_3|3_3|4_3|5_3|6_3|7_3|
vmrghh v4, v10, v14  ; v4:  |0_4|1_4|2_4|3_4|4_4|5_4|6_4|7_4|
vmrglh v5, v10, v14  ; v5:  |0_5|1_5|2_5|3_5|4_5|5_5|6_5|7_5|
vmrghh v6, v11, v15  ; v6:  |0_6|1_6|2_6|3_6|4_6|5_6|6_6|7_6|
vmrglh v7, v11, v15  ; v7:  |0_7|1_7|2_7|3_7|4_7|5_7|6_7|7_7|
```

Max Value The maximum value kernel searches for the maximum absolute valued element in an array of 32-bit signed integers (Algorithm 7). The most critical instruction for this kernel to proceed efficiently was, unsurprisingly, the max instruction. Although a maximum value search, as well as the computation of absolute value can be done with bitmask or partial store comparison results, these methods are far less efficient. The difference can be seen in looking at the SIMD code for the AMD Athlon (Algorithm 8), which includes max/min instructions, and Sun's VIS (Algorithm 9), which does not. Note that although the data type of the input array is 32-bit wide integers, the values of the data do not exceed signed 16-bit integers. Because of this, the AMD kernel is able to take advantage of the added parallelism of the smaller width. The Sun kernel is not able to do the same because it depends on 32-bit wide data communication operations (16-bit data communication operations are not available with VIS or the underlying SPARC v9 architecture).

Algorithm 7 Max Value Search

```
INT32 arr[], begin, end, i, max = 0;
  for ( i = begin; i < end; i++ ) {
    int x = abs( arr[i] );
    if ( x > max )
      max = x;
  }
```

Algorithm 8 AMD Max Value

```
movq     mm1, [esi]     ; mm1:  |     v1    |     v0    |
pxor     mm3, mm3       ; mm3:  |  0  |  0  |  0  |  0  |
movq     mm2, [esi + 8] ; mm2:  |     v3    |     v2    |
packssdw mm1, mm2       ; mm1:  |  v3 |  v2 |  v1 |  v0 |
psubsw   mm3, mm1       ; mm3:  | -v3 | -v2 | -v1 | -v0 |
pmaxsw   mm1, mm3       ; mm1:  |  v3 |  v2 |  v1 |  v0 | ABS VAL
pmaxsw   mm7, mm1       ; mm7:  new max absolute values
```

Algorithm 9 Sun Max Value

```
ld       [%g1], %f4     !  %f4:  |  arr[0]   |XXXXXXXXXXX|
ld       [%g1 + 4], %f5 !  %f4:  |  arr[0]   |  arr[1]   |
fpsub32  %f0, %f4, %f6
fcmpgt32 %f6, %f0, %o5  !  %o5:  mask of -arr[i] > 0
and      %o5, 0x1, %o4
and      %o5, 0x02, %o5
fmovrsgz %o4, %f7, %f5
fmovrsgz %o5, %f6, %f4  !  %f4:  |  arr[0]   |  arr[1]   | (abs val)
fcmpgt32 %f4, %f2, %o5  !  %o5:  mask of |arr[i]| > max[i]
and      %o5, 0x1, %o4
and      %o5, 0x2, %o5
fmovrsgz %o4, %f5, %f3
fmovrsgz %o5, %f4, %f2  !  %f2:  |   max0    |   max1    |
```

Quantize The original scalar version of the quantize kernel utilized a lookup table for certain ranges of input values in order to speed up common cases. Unfortunately, a lookup table approach is terribly inefficient for SIMD processing since each each element needs to be processed separately. Even so, full SIMD computations were faster than scalar code with look up tables on all of the platforms supporting SIMD floating point operations.

Subsample Horizontal/Vertical Subsampling arithmetically combines pixels together to form a composite value. The original MPEG-2 code accomplishes this through FIR filtering. In the horizontal subsampling kernel, seven pixels are weighted with coefficients and their products are summed together to create a composite pixel. The vertical subsampling kernel similarly weights twelve pixel values. In either kernel, each pixel value requires it's own index into the array representing the image, which must be kept within the horizontal and vertical boundaries of the image being operated on. Because of this, a large part of each iteration of these kernels is a scalar portion in which the needed indices are computed.

The organization of data is critical to SIMD performance. Consider the subsample horizontal (Algorithm 10) kernel on the Motorola G4. Because of the format of the data in registers it is inefficient to operate on more than one loop iteration at a time. However, in the case of vertical subsampling, the data for multiple loop iterations can be loaded into each register. In SIMD processing, the same operation is applied to multiple data elements, so vertical subsampling is a much more natural candidate for SIMD processing than the otherwise computationally similar horizontal subsample kernel. With a 64-bit wide SIMD extension, for example, eight vertical subsample iterations can be processed in parallel. This also means that all images must have horizontal dimensions evenly divisible by eight, or a scalar "clean up" loop must be coded to deal with any odd remaining widths. Many of our other kernels required such clean up loops as well.

Algorithm 10 Motorola Subsample Horizontal

```
lvsl    v31, r8, r3
lvx     v1, r8, r3  ; MSQ
lvx     v2, r9, r3  ; LSQ
; v1: |m5|XX|m3|XX|m1| i|p1|XX|p3|XX|p5|XX|XX|XX|XX|XX|
vperm   v1, v1, v2, v31
; v1: |m5|m3|m1| i|p1|p3|p5|XX|XX|XX|XX|XX|XX|XX|XX|XX|
vperm   v1, v1, v1, v22
vmrghb  v3, v0, v1  ; v3: | m5| m3| m1| i | p1| p3| p5|XXX|
vmulesh v5, v3, v21 ; v5: |   m5  |   m1  |   p1  |   p5  |<<SHMT
vmulosh v6, v3, v21 ; v6: |   m3  |    i  |   p3  |XXXXXXX|<<SHMT
vsumsws v5, v5, v0
vsumsws v5, v6, v5  ; v5: |   0   |   0   |   0   |t<<SHMT| ~
vaddsws v5, v5, v20 ; v5: += ROUND
vsraw   v5, v5, v23 ; v5: >> SHMT
vspltw  v5, v5, 3   ; v5: |  tmp  |  tmp  |  tmp  |  tmp  |
vpkswss v5, v5, v5  ; v5: |tmp|tmp|tmp|tmp|tmp|tmp|tmp|tmp|
vpkshus v5, v5, v5  ; v5: |t|t|t|t|t|t|t|t|t|t|t|t|t|t|t|t|
srawi   r15, r7, 1  ; r15: i>>1
stvebx  v5, r15, r4 ; dst[i>>1] = tmp
```

Algorithm 11 Short Term Analysis Filtering

```
INT16 *u; INT16 *rp; INT32 k_n; INT16 *s;
INT16 di, zzz, ui, sav, rpi; INT32 ltmp;
for (; k_n--; s++) {
 di = sav = *s;
 for (i = 0; i < 8; i++) {
  ui    = u[i]; rpi   = rp[i]; u[i]  = sav;
  ltmp=ui + (rpi*di + 16384)>>15;
  sav=ltmp>32767 ?  32767 :  (ltmp<-32768 ?  -32768 :  ltmp);
  ltmp=di + (rpi*ui + 16384)>>15;
  di=(ltmp)>32767 ?  32767 :  (ltmp<-32768 ?  -32768 :  ltmp);
 }
 *s = di;
}
```

Short Term Analysis Filtering The central operation of the short term analysis filtering kernel (Algorithm 11) is a fixed point multiply accumulate operation, followed by clipping (saturation) to a 16-bit signed value.

Algorithm 12 Short Term Synthesis Filtering

```
INT16 *v; INT16 *rrp; INT32 k; INT16 *wt; INT16 *sr;
INT16 sri, tmp1, tmp2; INT32 ltmp;
while (k--) {
  sri = *wt++;
  for (i = 8; i--;) {
   ltmp=(rrp[i]*v[i] + 16384)>>15;
   ltmp=ltmp>32767 ?  32767 :  (ltmp<-32768 ?  -32768 :  ltmp);
   ltmp=sri - ltmp;
   sri=ltmp>32767 ?  32767 :  (ltmp<-32768 ?  -32768 :  ltmp);
   ltmp=(rrp[i]*sri + 16384)>>15;
   ltmp=ltmp>32767 ?  32767 :  (ltmp<-32768 ?  -32768 :  ltmp);
   ltmp=v[i] + ltmp;
   v[i+1]=ltmp>32767 ?  32767 :  (ltmp<-32768 ?  -32768 :  ltmp);
  }
  *sr++ = v[0] = sri;
}
```

Short Term Synthesis Filtering Very similar to short term analysis filtering, the short term synthesis filtering kernel (Algorithm 12) is also based on a fixed point multiply accumulate operation, followed by clipping (saturation) to a 16-bit signed value.

Both the short term analysis filtering and short term synthesis filtering kernels were originally coded in a way that did not at first appear to be parallelizable in SIMD code. It was necessary to diagram by hand the computational path each element in the result followed in order to visualize what operations could be performed in parallel.

Synthesis Filtering Central to the synthesis filtering kernel from mp3 audio playback (Algorithm 13) are both positive and negative multiply-accumulate operations in single precision floating point. The final output audio samples must then be converted and clipped to signed 16-bit integers.

Algorithm 13 Synthesis Filtering

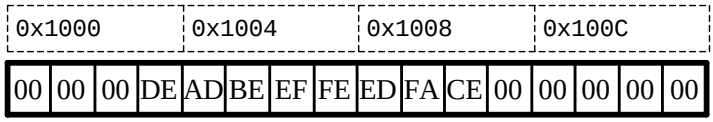
```
FLOAT32 *window; FLOAT32 *b0; INT16 *samples;
INT32 j, step = 2; FLOAT32 sum;
for (j=16; j>0; j--) {
 sum  = *window++ * *b0++;  sum -= *window++ * *b0++;
 sum += *window++ * *b0++;  sum -= *window++ * *b0++;
 sum += *window++ * *b0++;  sum -= *window++ * *b0++;
 sum += *window++ * *b0++;  sum -= *window++ * *b0++;
 sum += *window++ * *b0++;  sum -= *window++ * *b0++;
 sum += *window++ * *b0++;  sum -= *window++ * *b0++;
 sum += *window++ * *b0++;  sum -= *window++ * *b0++;
 sum += *window++ * *b0++;  sum -= *window++ * *b0++;
 /* clip output sample to 16-bit signed integer */
 *sample=sum>32767.0 ?  32767 :  (sum<-32768.0 ?  32768 :  sum);
 window += 0x10;
 samples += step;
}
```

Although not the only kernel to utilize multiply-accumulate operations, the synthesis filtering kernel benefits from them when they are available. Compare Algorithm 14 on the Motorola G4 which has the `vmaddfp` instruction with Algorithm 15 on the Intel Pentium III, which does not have a floating point multiply-accumulate operation. Both extensions are 128-bits wide.

A problem that often comes up in SIMD programming is the need to sum all of the packed elements in a register. This is often done at the end of an otherwise element independent computation. Motorola's AltiVec includes such sideways accumulation (also termed "reduction") operations for integer data types, but not for floating point. Intel's MMX and SSE instruction sets do not include reduction operations for any data types. This is regrettable because such an operation would find utility in this kernel (note the cumbersome way in which even and odd elements are summed together at the end of each code snippet).

1.6 Alignment

An aligned memory access is one that accesses a 2^N byte size data element at an address in which the lower N bits are all zeroes. Alignment is generally important on all architectures for performance reasons, as unaligned accesses typically suffer a significant performance penalty or cause a run time exception, depending on the behavior specified by an architecture. The Motorola and Sun instruction sets support only aligned memory access, so it is necessary to load two adjacent vectors of data and then rotate the data appropriately in order to simulate unaligned memory access. As an example, consider the sample Sun VIS code in Algorithm 16, loading the 64-bits data element 0xDEADBEEFFEEDFACE which is stored in memory at address 0x1003 as shown below:



Algorithm 14 Motorola Synthesis Filtering

```
vperm   v1, v1, v2, v20  ; v1: | win_0 | win_1 | win_2 | win_3 |
lvx     v4, r9, r3
vperm   v2, v2, v3, v20  ; v2: | win_4 | win_5 | win_6 | win_7 |
lvx     v5, r10, r3
vperm   v3, v3, v4, v20  ; v3: | win_8 | win_9 | win_A | win_B |
lvx     v6, 0, r4
vperm   v4, v4, v5, v20  ; v4: | win_C | win_D | win_E | win_F |
lvx     v7, r7, r4
lvx     v8, r8, r4
vperm   v6, v6, v7, v21  ; v6: |  b0_0 |  b0_1 |  b0_2 |  b0_3 |
lvx     v9, r9, r4
vperm   v7, v7, v8, v21  ; v7: |  b0_4 |  b0_5 |  b0_6 |  b0_7 |
lvx     v10, r10, r4
vperm   v8, v8, v9, v21  ; v8: |  b0_8 |  b0_9 |  b0_A |  b0_B |
vmaddfp v1, v1, v6, v0   ; v1: |sumevn0|sumodd0|sumevn1|sumodd1|
vperm   v9, v9, v10, v21 ; v9: |  b0_C |  b0_D |  b0_E |  b0_F |
vmaddfp v3, v3, v8, v0   ; v3: |sumevn2|sumodd2|sumevn3|sumodd3|
addi    r3, r3, 0x80
vmaddfp v1, v2, v7, v1   ; v1: |sumevn0|sumodd0|sumevn1|sumodd1|
addi    r4, r4, 0x40     ; r4: b0++ (x 16, pointer to FLOAT32)
vmaddfp v3, v4, v9, v3   ; v3: |sumevn2|sumodd2|sumevn3|sumodd3|
addic.  r11, r11, -1     ; j--
vaddfp  v1, v1, v3       ; v1: |sumevn0|sumodd0|sumevn1|sumodd1|
vsldoi  v2, v1, v1, 8    ; v2: |sumevn1|sumodd1|sumevn0|sumodd0|
vaddfp  v1, v1, v2       ; v1: |sum_evn|sum_odd|sum_evn|sum_odd|
vspltw  v2, v1, 0        ; v2: |sum_evn|sum_evn|sum_evn|sum_evn|
vspltw  v3, v1, 1        ; v3: |sum_odd|sum_odd|sum_odd|sum_odd|
vsubfp  v2, v2, v3       ; v2: SUM = sum_even - sum_odd
vctsxs  v1, v2, 0        ; v1: convert to integer
vpkswss v1, v1, v1       ; v1: |SUM|SUM|SUM|SUM|SUM|SUM|SUM|SUM|
```

Algorithm 15 Intel Synthesis Filtering

```
movups   xmm0, [esi + 0]       ; xmm0: |win_3|win_2|win_1|win_0|
movups   xmm4, [esi + 16]      ; xmm4: |win_7|win_6|win_5|win_4|
movups   xmm3, [ecx + 0]       ; xmm3: | b0_3| b0_2| b0_1| b0_0|
movups   xmm5, [ecx + 16]      ; xmm5: | b0_7| b0_6| b0_5| b0_4|
movaps   xmm7, xmm0
shufps   xmm0, xmm4, 10001000b ; xmm0: |win_6|win_4|win_2|win_0|
shufps   xmm7, xmm4, 11011101b ; xmm7: |win_7|win_5|win_3|win_1|
movaps   xmm6, xmm3
shufps   xmm3, xmm5, 10001000b ; xmm3: | b0_6| b0_4| b0_2| b0_0|
shufps   xmm6, xmm5, 11011101b ; xmm6: | b0_7| b0_5| b0_3| b0_1|
mulps    xmm0, xmm3            ; xmm0: sum += win_x*b0_x (even)
mulps    xmm6, xmm7
subps    xmm0, xmm6            ; xmm0: sum -= win_x*b0_x (odd)
movaps   xmm6, xmm3
shufps   xmm3, xmm5, 10001000b ; xmm3: |b0_14|b0_12|b0_10| b0_8|
shufps   xmm6, xmm5, 11011101b ; xmm6: |b0_15|b0_13|b0_11| b0_9|
mulps    xmm2, xmm3
mulps    xmm6, xmm7
addps    xmm0, xmm2            ; xmm0: sum += win_x*b0_x (even)
subps    xmm0, xmm6            ; xmm0: sum -= win_x*b0_x (odd)
;; merge sum values
movaps   xmm2, xmm0            ; xmm2: |sum_3|sum_2|sum_1|sum_0|
shufps   xmm0, xmm0, 00011011b ; xmm0: |sum_0|sum_1|sum_2|sum_3|
addps    xmm0, xmm2            ; xmm0: |s3+s0|s2+s1|s1+s2|s0+s3|
movaps   xmm2, xmm0
shufps   xmm2, xmm2, 10110001b ; xmm2: |s2+s1|s3+s0|s0+s3|s1+s2|
addps    xmm0, xmm2            ; xmm0: | SUM | SUM | SUM | SUM |
cvtps2pi mm0, xmm0             ; mm0:  |    SUM    |    SUM    |
packssdw mm0, mm0              ; mm0:  | SUM | SUM | SUM | SUM |
movd     ebx, mm0              ; ebx:  | SUM | SUM |
mov      [edi], bx             ; store 16-bit sample
```

Algorithm 16 Sun VIS Unaligned Memory Access

```
alignaddr    %i1, %g0, %l1    ! %l1: aligned address (0x1000)
                              ! set GSR align_addr field to 0x3
ldd          [%l1], %f2       ! %f2: |00|00|00|DE|AD|BE|EF|FE|
ldd          [%l1+8], %f4     ! %f4: |ED|FA|CE|00|00|00|00|00|
faligndata   %f2, %f4, %f2    ! %f2: |DE|AD|BE|EF|FE|ED|FA|CE|
```

In the code fragment from Algorithm 16, the address of the data to be loaded is in register `%i1`. The `alignaddr` instruction adds this address to the offset (in this case, register `%g0` is zero), clears the lower three address bits and places this 64-bit aligned address result in register `%l1`. The lower three bits of the intermediate result are placed in the graphics status register for later use by the `faligndata` instruction. Two consecutive register-width chunks of data must be loaded, and then shifted to the original alignment with the `faligndata` instruction. The DEC Alpha and x86 architectures (AMD and Intel) support unaligned access to memory, whereby the CPU utilizes special hardware to perform exactly the same steps (and thereby generates the same amount of memory traffic in reality).

The tradeoff between the two approaches is the additional register pressure and better awareness of alignment issues vs. ease of programming. However, there is a hidden benefit to explicitly aligned access: in the case where data is accessed sequentially, subsequent accesses cost a single additional load, as the previous upper half of the data to be shifted becomes the next load operation's lower half (at the cost of added register pressure). In this way, aligned memory accesses simulating unaligned access can outperform hardware which transparently performs unaligned access without recognizing the continuity. In addition, the hardware to support unaligned memory accesses is expensive both in terms of die area and latency [Thak99].

1.7 Other Useful Features

Splat Instructions Because SIMD instructions perform the same operation on each element, it is common to require the same constants in each element as well. Being able to avoid going to memory more often than necessary is a performance win. Compare the AltiVec approach (Algorithm 17) to Sun's VIS (Algorithm 18). Sun's solution is much more costly because it must access memory once for each vector constant, as opposed to once in the case of Motorola's AltiVec.

Algorithm 17 AltiVec DCT Constant Initialization

```
;; Load and Initialize Constants for DCT
lvsl    v31, 0, r5
lvx     v17, r11, r5
vperm   v16, v16, v17, v31 ; v16:  |K1|K2|K3|K4|K5|K6|K7|RD|
vsplth  v30, v16, 6        ; v30:  |K7|K7|K7|K7|K7|K7|K7|K7|
vsplth  v29, v16, 5        ; v29:  |K6|K6|K6|K6|K6|K6|K6|K6|
vsplth  v28, v16, 4        ; v28:  |K5|K5|K5|K5|K5|K5|K5|K5|
vsplth  v27, v16, 3        ; v27:  |K4|K4|K4|K4|K4|K4|K4|K4|
vsplth  v26, v16, 2        ; v26:  |K3|K3|K3|K3|K3|K3|K3|K3|
vsplth  v25, v16, 1        ; v25:  |K2|K2|K2|K2|K2|K2|K2|K2|
vsplth  v24, v16, 0        ; v24:  |K1|K1|K1|K1|K1|K1|K1|K1|
```

1.8 Bottlenecks

Highly Utilized Singular Resources The color space conversion kernel is an excellent example of the problem with singular, highly utilized resources. Consider the initial step of the color space conversion kernel which converts from band-interleaved to a more SIMD-friendly band-separated pixel form (Algorithm 19). The conclusion to be drawn is that

Algorithm 18 VIS DCT Constant Initialization

```
!  load constants
sethi  %hi(K1_VEC), %o2
ldd    [%o2 + %lo(K1_VEC)], %f32   !  %f32: K1
sethi  %hi(K2_VEC), %o2
ldd    [%o2 + %lo(K2_VEC)], %f34   !  %f34: K2
sethi  %hi(K3_VEC), %o2
ldd    [%o2 + %lo(K3_VEC)], %f36   !  %f36: K3
sethi  %hi(K4_VEC), %o2
ldd    [%o2 + %lo(K4_VEC)], %f38   !  %f38: K4
sethi  %hi(K5_VEC), %o2
ldd    [%o2 + %lo(K5_VEC)], %f40   !  %f40: K5
sethi  %hi(K6_VEC), %o2
ldd    [%o2 + %lo(K6_VEC)], %f42   !  %f42: K6
sethi  %hi(K7_VEC), %o2
ldd    [%o2 + %lo(K7_VEC)], %f44   !  %f44: K7
```

resources that will be highly utilized should be duplicated in order to allow for parallel instruction execution, otherwise a bottleneck is created which prevents the extraction of instruction level parallelism.

Algorithm 19 VIS Color Space Conversion Kernel

```
rgb_to_yuv_loop:
   !  load band interleaved input data
   alignaddr  %i0, %g0, %l0
   ldd        [%l0], %f2
   ldd        [%l0 + 8], %f4
   faligndata %f2, %f4, %f2    !  %f2:  |R0|G0|B0|R1|G1|B1|R2|G2|
   ldd        [%l0 + 16], %f6
   faligndata %f4, %f6, %f4    !  %f4:  |B2|R3|G3|B3|R4|G4|B4|R5|
   ldd        [%l0 + 24], %f8
   faligndata %f6, %f8, %f6    !  %f6:  |G5|B5|R6|G6|B6|R7|G7|B7|
   !  convert from band interleaved to band separated format
                               !  %f2:  |R0|G0|B0|R1|G1|B1|R2|G2|
   wr         %g0, 0x3, %gsr   !  set alignment for <<24
   faligndata %f2, %f4, %f8    !  %f8:  |R1|G1|B1|R2|G2|B2|R3|G3|
   wr         %g0, 0x6, %gsr   !  set alignment for <<48
   faligndata %f2, %f4, %f10   !  %f10:  |R2|G2|B2|R3|G3|B3|R4|G4|
   wr         %g0, 0x1, %gsr   !  set alignment for <<48
   faligndata %f4, %f6, %f12   !  %f12:  |R3|G3|B3|R4|G4|B4|R5|G5|
   wr         %g0, 0x4, %gsr   !  set alignment for <<32
   faligndata %f4, %f6, %f14   !  %f14:  |R4|G4|B4|R5|G5|B5|R6|G6|
   wr         %g0, 0x7, %gsr   !  set alignment for <<56
   faligndata %f4, %f6, %f16   !  %f16:  |R5|G5|B5|R6|G6|B6|R7|G7|
   wr         %g0, 0x2, %gsr   !  set alignment for <<16
   faligndata %f6, %f6, %f18   !  %f18:  |R6|G6|B6|R7|G7|B7|G5|B5|
   wr         %g0, 0x5, %gsr   !  set alignment for <<40
   faligndata %f6, %f6, %f20   !  %f20:  |R7|G7|B7|G5|B5|R6|G6|B6|
```

1.9 New Directions

1.9.1 Subsampling Memory Access

Figures 7a and 7b demonstrate the way in which the horizontal and vertical subsampling kernels access memory, respectively.

1.9.2 Sample Instruction Set

The instructions that comprise the proposed multimedia extension set are outlined Tables 8, 9, and 10.

Arithmetic Instructions	
Instruction	**Description**
vadd12s	16 x 12s modulo addition
vadd24s	8 x 24s modulo addition
vadd48s	4 x 48s modulo addition
vaddfp	4 x fp addition
vsub12s	16 x 12s modulo subtraction
vsub24s	8 x 24s modulo subtraction
vsub48s	4 x 48s modulo subtraction
vsubfp	4 x fp modulo subtraction
vmult24s	8 x 24s multiplication
vmult48s	4 x 48s multiplication
vmultfp	4 x fp multiplication
vxsum24s	8 x 24s cross-wise summation
vxsum48s	4 x 48s cross-wise summation
vxsumfp	4 x fp cross-wise summation
vrsqrtefp	4 x fp reciprocal sqrt estimation
vclipfp	4 x fp clip test comparison
vand	bitwise and
vor	bitwise or
vxor	bitwise exclusive or
vandc	bitwise and with complement
vnor	bitwise negated or
vsra12s	shift right arithmetic
vsrl12s	shift right logical
vsl12s	shift left
vsra24s	shift right arithmetic
vsrl24s	shift right logical
vsl24s	shift left
vsra48s	shift right arithmetic
vsrl48s	shift right logical
vsl48s	shift left
vmax12s	16 x 12s maximum
vmax24s	8 x 24s maximum
vmax48s	4 x 48s maximum
vmaxfp	4 x fp maximum
vmin12s	16 x 12s minimum
vmin24s	8 x 24s minimum
vmin48s	4 x 48s minimum
vminfp	4 x fp minimum
vsad12s	16 x 12s sum of absolute differences

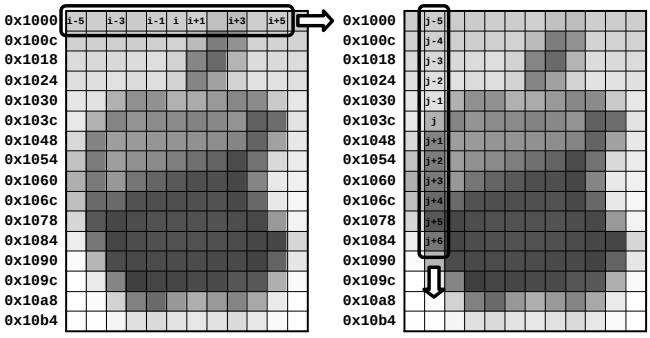
Table 8: **Arithmetic Instructions**

References

[Allen99] Gregory E. Allen, Brian L. Evans, Lizy K. John, "Real-Time High-Throughput Sonar Beamforming Kernels Using Native Signal Processing and Memory Latency Hiding Techniques," *Proc. of the 33rd IEEE Asilomar Conf. on Signals, Systems and Computers*, Pacific Grove, California, October 24-27, 1999, pp. 137-141

Memory Instructions	
Instruction	**Description**
vld8u12s	16 x 8u ->12s vector load
vld8u24s	8 x 8u->24s vector load
vld16s24s	8 x 16s->24s vector load
vld16s48s	4 x 16s->48s vector load
vld32s48s	4 x 32s->48s vector load
vldfpfp	4 x fp->fp vector load
vld192	1 x 192 vector load
vst12s8u	16 x 12s ->8u vector store
vst24s8u	8 x 24s->8u vector store
vst24s16s	8 x 24s->16s vector store
vst48s16s	4 x 48s->16s vector store
vst48s32s	4 x 48s->32s vector store
vstfpfp	4 x fp->fp vector store
vst192	1 x 192 vector store
vldstr64	strided load 64 total bits
vldstr128	strided load 128 total bits
vststr64	strided store 64 total bits
vststr128	strided store 128 total bits
vldsc8u12s	8u->12s load scalar
vldsc8u24s	8u->24s load scalar
vldsc16s24s	16s->24s load scalar
vldsc16s48s	16s->48s load scalar
vldsc32s48s	32s->48s load scalar
vldscfpfp	fp->fp load scalar
vstsc12s8u	12s->8u store scalar
vstsc24s8u	24s->8u store scalar
vstsc24s16s	24s->16s store scalar
vstsc48s16s	48s->16s store scalar
vstsc48s32s	48s->32s store scalar
vstscfpfp	fp->fp store scalar
dss	data stream stop
dst	data stream touch
dstst	data stream touch for store

Table 9: **Memory Instructions**



(a) Horizontal Subsampling (b) Vertical Subsampling

Figure 7: **Subsampling Memory Access** - the bytes required to compute one loop iteration are circled, with an arrow indicating the direction the original algorithm moves on subsequent iterations

Data Rearrangement Instructions	
Instruction	**Description**
vmrgh12	upper 8 x 12 merge
vmrgh24	upper 4 x 24 merge
vmrgh48	upper 2 x 48 merge
vmrgl12	lower 8 x 12 merge
vmrgl24	lower 4 x 24 merge
vmrgl48	lower 2 x 48 merge
vpermset12	16 x 12 permute subset
vpermset24	8 x 24 permute subset
vpermset48	4 x 48 permute subset
vcheck12	16 x 12 checkerboard
vcheck24	8 x 24 checkerboard
vcheck48	4 x 48 checkerboard
vexcheck12	16 x 12 reverse checkerboard
vexcheck24	8 x 24 reverse checkerboard
vexcheck48	4 x 48 reverse checkerboard
vsplt12	splat 12-bit element
vsplt24	splat 24-bit element
vsplt48	splat 48-bit element
vsplti12s	splat immediate as 12s
vsplti24s	splat immediate as 24s
vsplti48s	splat immediate as 48s
vldsl	load left permute control vector
vldsr	load right permute control vector
vperm	12-bit permutation
vrotli12	rotate left immediate by 12-bit multiples

Table 10: **Data Rearrangement Instructions**

[AMD00] Advanced Micro Devices, Inc., "AMD Athlon Processor x86 Code Optimization Guide," Publication #22007, Rev. G, April 2000, *http://www.amd.com/products /cpg/athlon/techdocs/pdf/22007.pdf*, retrieved April 24, 2000

[AMD99] Advanced Micro Devices, Inc., "AMD Athlon Processor Technical Brief," Publication #22054, Rev. D, December 1999, *http://www.amd.com/products /cpg/athlon/techdocs/pdf/22054.pdf*, retrieved April 24, 2000

[Bhar98] R. Bhargava, L. K. John, B. L. Evans, R. Radhakrishnan, "Evaluating MMX Technology Using DSP and Multimedia Applications," *Proc. of the 31st IEEE Intl. Symp. on Microarchitecture (MICRO-31)*, Dallas, Texas, November 30-December 2, 1998, pp. 37-46

[Chen96] Wilam Chen, H. John Reekie, Sunil Bhave, Edward A. Lee, "Native Signal Processing on the UltraSparc in the Ptolemy Environment," *Proc. of the 30th Annual Asilomar Conf. on Signals, Systems, and Computers*, Pacific Grove, California, November 3-6, 1996, Vol. 2, pp. 1368-1372

[Cheo97] Gerald Cheong, Monica Lam, "An Optimizer for Multimedia Instruction Sets," *Proc. of the 2nd SUIF Compiler Workshop, August 21-23, 1997, Stanford University, http://suif.stanford.edu/suifconf/suifconf2/papers/21.ps*, retrieved April 24, 2000

[Code00] CodePlay, "VectorC Compiler," *http://www.codeplay.com/vectorc.html*, retrieved November 26, 2000

[Comp00] Compaq Computer Corporation, "Alpha 21264 Microprocessor Hardware Reference Manual," Part No. DS-0027A-TE, February 2000, *http://www.support.compaq.com/alpha-tools/documentation/current/21264_EV67/ds-0027a-te_21264_hrm.pdf*, retrieved April 24, 2000

[Cont97] Thomas M. Conte, Pradeep K. Dubey, Matthew D. Jennings, Ruby B. Lee, Alex Peleg, Salliah Rathnam, Mike Schlansker, Peter Song, Andrew Wolfe, "Challenges to Combining General-Purpose and Multimedia Processors," *IEEE Computer*, Vol. 30, No. 12, December 1997, pp. 33-37

[Fish98] Randall J. Fisher, Henry G. Dietz, "Compiling for SIMD Within a Register," *Proc. of the 11th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC'98)*, Chapel Hill, North Carolina, August 7-9, 1998, pp. 290-304

[Intel99a] Intel Corporation, "Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture," Publication 243190, 1999, *http://developer.intel.com/design/pentiumii /manuals/24319002.PDF*, retrieved April 24, 2000

[Intel99b] Intel Corp., "Increasing the Accuracy of the Results from the Reciprocal and Reciprocal Square Root Instructions using the Newton-Raphson Method," *Intel Application Note AP-803*, February 4, 1999, *http://developer.intel.com /vtune/cbts/strmsimd/appnotes.htm*, retrieved April 24, 2000

[Intel00] Intel Corp., "Intel C++ Compiler," *http://developer.intel.com/software/products/compilers/c50/*, retrieved November 26, 2000.

[Kesh99] Jagannath Keshava, Vladimir Pentkovski, "Pentium III Processor Implementation Tradeoffs," Intel Technology Journal, Quarter 2, 1999, *http://developer.intel.com/technology/itj /q21999/pdf/impliment.pdf*, retrieved April 24, 2000

[Kuro98] Ichiro Kuroda, Takao Nishitani, "Multimedia Processors," *Proc. of the IEEE*, Vol. 86 No. 6, June 1998, pp. 1203-1221

[Lee96] Ruby B. Lee, Michael D. Smith, "Media Processing: A New Design Target," *IEEE Micro*, Vol. 16, No. 4, August 1996, pp. 6-9

[Luml97] John Lumley, "Packed Arithmetic - Architectural Influence on Compilation," *IEE Colloquium on Multimedia Instruction Sets: Design and Application*, Birmingham, United Kingdom, February 25, 1997, *http://www.hpl.hp.com/techreports/97/HPL-97-36.pdf*, retrieved April 24, 2000

[Moto00] Motorola Inc., "MPC7400 RISC Microprocessor User's Manual, Rev. 0," Document MPC7400UM/D, March 27, 2000, *http://www.mot.com/SPS/PowerPC/teksupport /teklibrary/manuals/MPC7400UM.pdf*, retrieved April 24, 2000

[Norm98] Kevin B. Normoyle, Michael A. Csoppenszky, Allan Tzeng, Timothy P. Johnson, Christopher D. Furman, Jamshid Mostoufi, "UltraSPARC-IIi: Expanding the Boundaries of a System on a Chip," *IEEE Micro*, Vol. 18, No. 2, March/April 1998, pp. 14-24

[SUIF] The Stanford SUIF Compiler Group, *http://suif.stanford.edu/*, retrieved April 24, 2000

[Sun97] Sun Microsystems Inc., "UltraSPARC-IIi User's Manual," 1997, Part No. 805-0087-01, *http://www.sun.com/microelectronics/UltraSPARC-IIi/docs/805-0087.pdf*, retrieved April 24, 2000

[Thak99] Shreekant (Ticky) Thakkar, Tom Huff, "The Internet Streaming SIMD Extensions," *Intel Technology Journal*, Q2 1999, *http://developer.intel.com/technology/itj/q21999.htm*, retrieved April 24, 2000

[Yates] Charles R. Yates, "Fixed-Point Arithmetic: An Introduction," *http://www.shadow.net/~yates/papers.htm*, retrieved April 24, 2000

add_block.c

```
#include<pyrogen.h>

/* move/add 8x8-Block from block[comp] to backward_reference_frame */
/* copy reconstructed 8x8 block from block[comp] to current_frame[]
 * ISO/IEC 13818-2 section 7.6.8: Adding prediction and coefficient data
 * This stage also embodies some of the operations implied by:
 *   - ISO/IEC 13818-2 section 7.6.7: Combining predictions
 *   - ISO/IEC 13818-2 section 6.1.3: Macroblock
*/


extern UINT8 *Clip;

void __pyrogen_add_block(INT16 *bp, UINT8 *rfp, INT32 iincr, INT32 addflag) {
  int i, j;

  if(addflag) {
    for (i=0; i<8; i++) {
      for (j=0; j<8; j++) {
        *rfp = Clip[*bp++ + *rfp];
        rfp++;
      }

      rfp+= iincr;
    }
  }
  else {
    for (i=0; i<8; i++) {
      for (j=0; j<8; j++) {
        *rfp++ = Clip[*bp++ + 128];
      }
      rfp+= iincr;
    }
  }
}
```

block_match.c

```
#include <pyrogen.h>

/* Notes:     *Extracted from mpeg2enc
 *            *Code has been modified for readability
 */

/*
 * total absolute difference between two (16*h) blocks
 * including optional half pel interpolation of blk1 (hx,hy)
 * blk1,blk2: addresses of top left pels of both blocks
 * lx:        distance (in bytes) of vertically adjacent pels
 * hx,hy:     flags for horizontal and/or vertical interpolation
 * h:         height of block (usually 8 or 16)
 * distlim:   bail out if sum exceeds this value
 */
INT32 __pyrogen_block_match(UINT8 *block_1, UINT8 *block_2, INT32 lx, INT32 hx, INT32
hy, INT32 h, INT32 distlim) {
  unsigned char *p1,*p1a,*p2;
  int i,j;
  int s,v;

  s = 0;
  p1 = block_1;
  p2 = block_2;

  if (!hx && !hy)
    for (j=0; j<h; j++)
    {
      if ((v = p1[0]  - p2[0])<0)  v = -v; s+= v;
      if ((v = p1[1]  - p2[1])<0)  v = -v; s+= v;
      if ((v = p1[2]  - p2[2])<0)  v = -v; s+= v;
      if ((v = p1[3]  - p2[3])<0)  v = -v; s+= v;
      if ((v = p1[4]  - p2[4])<0)  v = -v; s+= v;
      if ((v = p1[5]  - p2[5])<0)  v = -v; s+= v;
      if ((v = p1[6]  - p2[6])<0)  v = -v; s+= v;
      if ((v = p1[7]  - p2[7])<0)  v = -v; s+= v;
      if ((v = p1[8]  - p2[8])<0)  v = -v; s+= v;
      if ((v = p1[9]  - p2[9])<0)  v = -v; s+= v;
      if ((v = p1[10] - p2[10])<0) v = -v; s+= v;
      if ((v = p1[11] - p2[11])<0) v = -v; s+= v;
      if ((v = p1[12] - p2[12])<0) v = -v; s+= v;
      if ((v = p1[13] - p2[13])<0) v = -v; s+= v;
      if ((v = p1[14] - p2[14])<0) v = -v; s+= v;
      if ((v = p1[15] - p2[15])<0) v = -v; s+= v;

      if (s >= distlim)
        break;

      p1+= lx;
      p2+= lx;
    }
  else if (hx && !hy)
    for (j=0; j<h; j++)
    {
      for (i=0; i<16; i++)
      {
        v = ((unsigned int)(p1[i]+p1[i+1]+1)>>1) - p2[i];
        if (v>=0)
          s+= v;
        else
          s-= v;
      }
      p1+= lx;
      p2+= lx;
    }
  else if (!hx && hy)
  {
    p1a = p1 + lx;
    for (j=0; j<h; j++)
    {
      for (i=0; i<16; i++)
      {
        v = ((unsigned int)(p1[i]+p1a[i]+1)>>1
        if (v>=0)
          s+= v;
        else
          s-= v;
      }
      p1 = p1a;
      p1a+= lx;
      p2+= lx;
    }
  }
  else /* if (hx && hy) */
  {
    p1a = p1 + lx;
    for (j=0; j<h; j++)
    {
      for (i=0; i<16; i++)
      {
        v = ((unsigned int)(p1[i]+p1[i+1]+p1a[
        if (v>=0)
          s+= v;
        else
          s-= v;
      }
      p1 = p1a;
      p1a+= lx;
      p2+= lx;
    }
  }

  return s;
}
```

clip.c

```
#include <pyrogen.h>

/* Vertex buffer clipping flags */
#define CLIP_RIGHT_BIT   0x01
#define CLIP_LEFT_BIT    0x02
#define CLIP_TOP_BIT     0x04
#define CLIP_BOTTOM_BIT  0x08
#define CLIP_NEAR_BIT    0x10
#define CLIP_FAR_BIT     0x20

void __pyrogen_project_and_cliptest_perspective(UINT32 n, FLOAT32 vClip[][4], FLOAT32
*m, const FLOAT32 vEye[][4], UINT8 clipMask[], UINT8 *orMask, UINT8 *andMask) {
  FLOAT32 m0 = m[0], m5 = m[5], m8 = m[8], m9 = m[9];
  FLOAT32 m10 = m[10], m14 = m[14];
  UINT32 i;
  UINT8 tmpOrMask = *orMask;
  UINT8 tmpAndMask = *andMask;

  for (i=0;i<n;i++) {
    FLOAT32 ex = vEye[i][0], ey = vEye[i][1];
    FLOAT32 ez = vEye[i][2], ew = vEye[i][3];
    volatile FLOAT32 cx = m0 * ex           + m8  * ez           ;
    volatile FLOAT32 cy =           m5 * ey + m9  * ez           ;
    volatile FLOAT32 cz =                     m10 * ez + m14 * ew;
    volatile FLOAT32 cw =                          -ez           ;
    UINT8 mask = 0;
    vClip[i][0] = cx;
    vClip[i][1] = cy;
    vClip[i][2] = cz;
    vClip[i][3] = cw;
    if (cx >  cw)       mask |= CLIP_RIGHT_BIT;
    else if (cx < -cw)  mask |= CLIP_LEFT_BIT;
    if (cy >  cw)       mask |= CLIP_TOP_BIT;
    else if (cy < -cw)  mask |= CLIP_BOTTOM_BIT;
    if (cz >  cw)       mask |= CLIP_FAR_BIT;
    else if (cz < -cw)  mask |= CLIP_NEAR_BIT;
    if (mask) {
      clipMask[i] |= mask;
      tmpOrMask |= mask;
    }
    tmpAndMask &= mask;
  }
  *orMask = tmpOrMask;
  *andMask = tmpAndMask;

}
```

color_space.c

```
#include <pyrogen.h>

/* Notes: Code extracted and cleaned up from mpeg2enc read_ppm(), */
/*        conv444to422() and conv422to420                         */
/*                                                                */
/* Input: band interleaved RGB format                             */
/* Ouput: band separated YUV format                               */
/*                                                                */

static INT32 matrix_coefficients = 3;

void __pyrogen_rgb_to_yuv(UINT8 *input, UINT8 *y_base, UINT8 *u_base, UINT8 *v_base, I
NT32 width, INT32 height) {
  INT32 i, j;
  FLOAT64 cr, cg, cb, cu, cv;
  UINT8 *yp, *up, *vp, *rowp;
  INT32 r, g, b;
  FLOAT64 y, u, v;

  static FLOAT64 coef[7][3] = {
    {0.2125,0.7154,0.0721}, /* ITU-R Rec. 709 (1990) */
    {0.299, 0.587, 0.114},  /* unspecified */
    {0.299, 0.587, 0.114},  /* reserved */
    {0.30,  0.59,  0.11},   /* FCC */
    {0.299, 0.587, 0.114},  /* ITU-R Rec. 624-4 System B, G */
    {0.299, 0.587, 0.114},  /* SMPTE 170M */
    {0.212, 0.701, 0.087}}; /* SMPTE 240M (1987) */

  yp = y_base;
  up = u_base;
  vp = v_base;

  rowp = input;

  i = matrix_coefficients;
  if (i>8)
    i = 3;

  cr = coef[i-1][0];
  cg = coef[i-1][1];
  cb = coef[i-1][2];
  cu = 0.5/(1.0-cb);
  cv = 0.5/(1.0-cr);

  for (i=0; i<height; i++) {
    for (j=0; j<width; j++) {
      r = *rowp++;
      g = *rowp++;
      b = *rowp++;
      /* convert to YUV */
      y = cr*r + cg*g + cb*b;
      u = cu*(b-y);
      v = cv*(r-y);
      *yp++ = (219.0/256.0)*y + 16.5;  /* nominal range: 16..235 */
      *up++ = (224.0/256.0)*u + 128.5; /* 16..240 */
      *vp++ = (224.0/256.0)*v + 128.5; /* 16..240 */
    }
  }
}
```

dct.c

```
#include <pyrogen.h>

/* fdctref.c, forward discrete cosine transform, double precision          */

/* Copyright (C) 1996, MPEG Software Simulation Group. All Rights Reserved. */

/*
 * Disclaimer of Warranty
 *
 * These software programs are available to the user without any license fee or
 * royalty on an "as is" basis.  The MPEG Software Simulation Group disclaims
 * any and all warranties, whether express, implied, or statuary, including any
 * implied warranties or merchantability or of fitness for a particular
 * purpose.  In no event shall the copyright-holder be liable for any
 * incidental, punitive, or consequential damages of any kind whatsoever
 * arising from the use of these programs.
 *
 * This disclaimer of warranty extends to the user of these programs and user's
 * customers, employees, agents, transferees, successors, and assigns.
 *
 * The MPEG Software Simulation Group does not represent or warrant that the
 * programs furnished hereunder are free of infringement of any third-party
 * patents.
 *
 * Commercial implementations of MPEG-1 and MPEG-2 video, including shareware,
 * are subject to royalty fees to patent holders.  Many of these patents are
 * general enough such that they are unavoidable regardless of implementation
 * design.
 *
 */

/* private data */
extern FLOAT64 c[8][8]; /* transform coefficients */

void __pyrogen_dct(INT16 *block) {
  INT32 i, j, k;
  FLOAT64 s;
  FLOAT64 tmp[64];

  for (i=0; i<8; i++) {
    for (j=0; j<8; j++) {
      s = 0.0;

      for (k=0; k<8; k++) {
        s += c[j][k] * block[8*i+k];
      }

      tmp[8*i+j] = s;
    }
  }

  for (j=0; j<8; j++) {
    for (i=0; i<8; i++) {
      s = 0.0;

      for (k=0; k<8; k++) {
        s += c[i][k] * tmp[8*k+j];
      }

      block[8*i+j] = (int)floor(s+0.499999);
      /*
       * reason for adding 0.499999 instead of 0.5:
       * s is quite often x.5 (at least for i and/or j = 0 or 4)
       * and setting the rounding threshold exactly to 0.5 leads to an
       * extremely high arithmetic implementat
       * s being between x.5 and x.500001 (whi
       * downwards instead of upwards) is assu
       * (if at all)
       */
    }
  }
}
```

fft.c

```
#include <pyrogen.h>

/* Replacement FFT routine for mp3 encoding.
   Wedged in by Mike Cheng  :  http://www.cryogen.com/mikecheng
   Based upon split-radix fft by Malvar
       from book:  "Signal Processing with Lapped Transforms". Malvar, HS.

   The reason this fft is so damn fast, is because mp3 encoding only uses
   *real* FFTs.  The previous fft routine included in the iso source is for
   a complex->complex transform.  But if the imaginary part of the signal
   is zero (as it is for encoding sound), then you only have to do a
   real->complex fft.  This makes this new fft routine about twice the
   speed of the old one.

   There is another real->complex fft that is about another 30% than the
   split-radix (in theory), but it may take me a lot longer to put in.

   later
   mike
*/

#define BLKSIZE 1024
#define MAXLOGM  25
#define TWOPI  6.28318530717958647692
#define SQHALF   0.70710678118654524401

void __pyrogen_pow_phase(FLOAT32 x_real[BLKSIZE],FLOAT32 energy[BLKSIZE], FLOAT32 phi[
BLKSIZE], INT32 N) {
  INT32 i;
  FLOAT32 *ep, *pp, *epn, *ppn;
  FLOAT32 *xp1, *xpn;

  ep = energy;
  pp = phi;
  xp1 = x_real;
  xpn = &x_real[N-1];

  *ep++ = *xp1 * *xp1;
  *pp++ = atan2( 0.0, (double)(*xp1++));

  if (N==1024) {
    for (i=1;i<512;i++) {
      *ep = *xp1 * *xp1 + *xpn * *xpn;
       if (*ep < 0.0005) {
         *ep++ = 0.0005;
         *pp++ = 0.0;
         xpn--;
         xp1++;
       }
       else {
         ep++;
         *pp++ = atan2( -(double)(*xpn--), (double)(*xp1++) );
       }
    }

    ep = &energy[513];
    pp = &phi[513];
    epn = &energy[511];
    ppn = &phi[511];
    for (i=1;i<512;i++) {
      *ep++ = *epn--;
      *pp++ = - *ppn--;
    }

    energy[512] = x_real[512] * x_real[512];
    phi[512] = atan2 (0.0, (double)x_real[512]
  }
  else {
    for (i=1;i<N/2;i++) {
      *ep = *xp1 * *xp1 + *xpn * *xpn;
      if (*ep < 0.0005) {
        *ep++ = 0.0005;
        *pp++ = 0.0;
        xpn--;
        xp1++;
      }
      else {
        ep++;
        *pp++ = atan2( -(double)(*xpn--), (dou
      }
    }

    ep = &energy[N/2+1];
    pp = &phi[N/2+1];
    epn = &energy[N/2-1];
    ppn = &phi[N/2-1];
    for (i=1;i<N/2;i++) {
      *ep++ = *epn--;
      *pp++ = - *ppn--;
    }

   energy[N/2] = x_real[N/2] * x_real[N/2];
   phi[N/2] = atan2 (0.0, (double)x_real[N/2])
   }
}
/*****************************************
* Data unshuffling according to bit-reversed i
*
* Bit reversal is done using Evans' algorithm.
* "An Improved Digit-Reversal Permutation Algo
*  IEEE Trans. ASSP, Aug 1987, pp. 1120-25.
*
*****************************************

static INT32 brseed[256]; /* Evans' seed table
static INT32 brsflg;   /* flag for table build

void __pyrogen_BR_permute(FLOAT32 *x, INT32 lo
  register INT32 lg2, n;
  INT32 i,j,imax;
  INT32 off,fj, gno, *brp;
  FLOAT32 tmp, *xp, *xq;

  lg2 = logm >> 1;
  n = 1 << lg2;

  if (logm & 1) lg2++;

  /* create seed table if not yet built */
  if (brsflg != logm) {
    brsflg = logm;
    brseed[0] = 0;
    brseed[1] = 1;
    for(j=2; j <= lg2; j++){
      imax =  1 << (j - 1);
      for (i=0; i < imax; i++){
        brseed[i] <<= 1;
        brseed[i + imax] = brseed[i] + 1;
```

fft.c

```
      }
    }
  }

  /* unshuffling loop */
  for(off = 1; off < n; off++){
    fj = n*brseed[off]; i = off; j = fj;
    tmp = x[i]; x[i] = x[j]; x[j] = tmp;
    xp = &x[i];
    brp = &brseed[1];
    for(gno = 1; gno < brseed[off]; gno++){
      xp += n;
      j = fj + *brp++;
      xq = x + j ;
      tmp = *xp; *xp = *xq; *xq = tmp;
    }
  }
}

/*************************************************************
*                                                            *
* Recursive part of split radix FFT algorithm
*
*************************************************************/

void __pyrogen_srrec(FLOAT32 *xr, FLOAT32 *xi, INT32 logm) {
  static INT32 m, m2, m4, m8, nel, n;
  static FLOAT32 *xr1, *xr2, *xi1, *xi2;
  static FLOAT32 *cn, *spcn, *smcn, *c3n, *spc3n, *smc3n;
  static FLOAT32 tmp1, tmp2, ang, c, s;
  static FLOAT32 *tab[MAXLOGM];

  /* check range of logm */

  if ((logm < 0) || (logm > MAXLOGM)){
    fprintf(stderr, "Error: SRFFT logm = %d is out of bounds [%d %d]\n", logm, 0, MAXL
OGM);
    exit(1);
  }

  /* compute trivial cases */
  if (logm < 3){
    if (logm == 2){
      xr2 = xr + 2;
      xi2= xi + 2;
      tmp1 = *xr + *xr2;
      *xr2 = *xr - *xr2;
      *xr = tmp1;
      tmp1 = *xi + *xi2;
      *xi2 = *xi - *xi2;
      *xi = tmp1;
      xr1 = xr + 1;
      xi1 = xi + 1;
      xr2++;
      xi2++;
      tmp1 = *xr1 + *xr2;
      *xr2 = *xr1 - *xr2;
      *xr1 = tmp1;
      tmp1 = *xi1 +  *xi2;
      *xi2 = *xi1 - *xi2;
      *xi1 = tmp1;
      xr2 = xr + 1;
      xi2 = xi + 1;
      tmp1 = *xr + *xr2;
      *xr2 = *xr - *xr2;
      *xr = tmp1;
      tmp1 = *xi + *xi2;
      *xi2 = *xi - *xi2;
      *xi = tmp1;
      xr1 = xr +2;
      xi1 = xi+2;
      xr2 = xr+3;
      xi2 = xi + 3;
      tmp1 = *xr1 + *xi2;
      tmp2 = *xi1 + *xr2;
      *xi1 = *xi1 - *xr2;
      *xr2 = *xr1 - *xi2;
      *xr1 = tmp1;
      *xi2 = tmp2;
      return;
    }
    else if (logm == 1){
      xr2 = xr + 1;
      xi2 = xi + 1;
      tmp1 = *xr + *xr2;
      *xr2 = *xr - *xr2;
      *xr = tmp1;
      tmp1 = *xi + *xi2;
      *xi2 = *xi - *xi2;
      *xi = tmp1;
      return;
    }
    else if (logm == 0) return;
  }

  /* compute a few constants */

  m = 1 << logm; m2 = m / 2; m4 = m2 / 2; m8 =


  /* build tables of butterfly coefficients if
  if ((logm >=4) && (tab[logm-4] == NULL)){
    /* allocate memory for tables */
    nel = m4 - 2;
    if ((tab[logm-4] = (FLOAT32 *) calloc(6 *
        == NULL){
      exit(1);
    }

    /* initialize pointers */
    cn = tab[logm-4]; spcn = cn + nel; smcn =
    c3n = smcn + nel; spc3n = c3n + nel; smc3n

    /* compute tables */
    for (n = 1; n < m4; n++){
      if (n == m8) continue;
      ang = n * TWOPI / m;
      c = cos(ang); s = sin(ang);
      *cn++ = c; *spcn++ = -(s + c); *smcn++ =
      ang = 3*n*TWOPI/m;
      c = cos(ang); s = sin(ang);
      *c3n++ = c; *spc3n++ = -(s + c); *smc3n+
    }
  }

  /* step 1 */
  xr1 = xr; xr2 = xr1 + m2;
  xi1 = xi; xi2 = xi1 + m2;
```

fft.c

```
  for (n =0; n < m2; n++){
    tmp1 = *xr1 + *xr2;
    *xr2 = *xr1 - *xr2;
    *xr1 = tmp1;
    tmp2 = *xi1 + *xi2;
    *xi2 = *xi1 - *xi2;
    *xi1 = tmp2;
    xr1++; xr2++; xi1++; xi2++;
  }

  /* Step 2 */
  xr1 = xr + m2;  xr2 = xr1 + m4;
  xi1 = xi + m2;  xi2 = xi1+ m4;
  for (n = 0; n < m4; n++){
    tmp1 = *xr1 + *xi2;
    tmp2 = *xi1 + *xr2;
    *xi1 = *xi1 - *xr2;
    *xr2 = *xr1 - *xi2;
    *xr1 = tmp1;
    *xi2 = tmp2;
    xr1++; xr2++; xi1++; xi2++;
  }

  /* Steps 3&4 */
  xr1 = xr + m2; xr2 = xr1 + m4;
  xi1 = xi + m2; xi2 = xi1 + m4;

  if (logm >= 4) {
    nel = m4 -2;
    cn = tab[logm-4]; spcn = cn + nel; smcn = spcn + nel;
    c3n = smcn + nel; spc3n = c3n + nel; smc3n = spc3n + nel;
  }

  xr1++; xr2++; xi1++; xi2++;

  for(n=1; n < m4; n++){
    if (n == m8){
      tmp1 = SQHALF*(*xr1 + *xi1);
      *xi1 = SQHALF*(*xi1 - *xr1);
      *xr1 = tmp1;
      tmp2 = SQHALF*(*xi2 - *xr2);
      *xi2 = -SQHALF*(*xr2 + *xi2);
      *xr2 = tmp2;
    }
    else {
      tmp2 = *cn++ *(*xr1 + *xi1);
      tmp1 = *spcn++ * *xr1 + tmp2;
      *xr1 = *smcn++ * *xi1 + tmp2;
      *xi1 = tmp1;
      tmp2 = *c3n++ * (*xr2 + *xi2);
      tmp1 = *spc3n++ * *xr2 + tmp2;
      *xr2 = *smc3n++ * *xi2 + tmp2;
      *xi2 = tmp1;
    }
    xr1++; xr2++; xi1++; xi2++;
  }

  /* call ssrec again with half DFT length */
  __pyrogen_srrec(xr, xi, logm -1);
  /* call ssrec again twice with one quarter DFT length.
     Constants have to be recomputed because they are static! */

  m = 1 << logm; m2 = m/2;
    __pyrogen_srrec(xr+m2, xi+m2, logm - 2);

    m = 1 << logm; m4 = 3*(m/4);
    __pyrogen_srrec(xr+m4, xi+m4, logm - 2);
  }

  /*****************************************
  *  Direct transform
  *
  *****************************************

  void __pyrogen_srfft(FLOAT32 *xr, FLOAT32 *xi,
    /* call recursive routine */
    __pyrogen_srrec(xr, xi, logm);

    /* output array unshuffling using bit-reverse
    if (logm > 1){
      __pyrogen_BR_permute(xr, logm);
      __pyrogen_BR_permute(xi, logm);
    }
  }

  /* recursive part of rsfft algo. not externally
  static void __pyrogen_rsrec(FLOAT32 *x, INT32
    static INT32 m,m2,m4,m8,nel, n;
    static FLOAT32 *xr1, *xr2, *xi1;
    static FLOAT32 *cn, *spcn, *smcn;
    static FLOAT32 tmp1, tmp2, ang, c, s;
    static FLOAT32 *tab[MAXLOGM];

    /* check range of logm */
    if ((logm <0)|| (logm>MAXLOGM)) {
      fprintf(stderr,"log out of range: %i\n",log
      exit(-1);
    }

    /* compute trivial cases */
    if (logm <2) {
      if (logm ==1) { /* length m = 2*/
        xr2 = x + 1;
        tmp1 = *x + *xr2;
        *xr2 = *x - *xr2;
        *x = tmp1;
        return;
      }
      else if (logm==0) return; /* length m =1 */
    }

    /* compute a few constants */
    m=1<<logm; m2=m/2;m4=m2/2;m8=m4/2;

    /* build tables of butterfly coefficients if
    if ((logm >= 4) && (tab[logm-4]==NULL)) {
      /* allocate memory for tables */
      nel=m4-2;
      if ((tab[logm-4] = (FLOAT32 *) calloc(3 *
        fprintf(stderr,"cosine table memory erro
        exit(-1);
      }
      /* initialize pointers */
      cn = tab[logm-4]; spcn = cn+nel; smcn =spcn

      /* compute tables */
      for (n = 1; n<m4;n++) {
```

fft.c

```
    if (n==m8) continue;
    ang = n * TWOPI/m;
    c=cos(ang); s=sin(ang);
    *cn++ = c; *spcn++ = -(s+c); *smcn++ = s-c;
  }
}

/* step 1 */
xr1 = x; xr2 = xr1+m2;
for (n=0; n< m2; n++) {
  tmp1 = *xr1 + *xr2;
  *xr2 = *xr1- *xr2;
  *xr1 = tmp1;
  xr1++; xr2++;
}

/* step 2 */
xr1 = x  + m2 + m4;
for (n=0;n<m4;n++) {
  *xr1 = - *xr1;
  xr1++;
}

/* steps 3 and 4 */
xr1 = x + m2; xi1 = xr1 + m4;
if (logm >= 4) {
  nel = m4-2;
  cn = tab[logm-4]; spcn = cn + nel; smcn = spcn + nel;
}
xr1++; xi1++;
for (n = 1 ; n<m4; n++) {
  if (n == m8) {
    tmp1 = SQHALF * ( *xr1 + *xi1);
    *xi1 = SQHALF * ( *xi1 - *xr1);
    *xr1 = tmp1;
  } else {
    tmp2 = *cn++ * (*xr1 + *xi1);
    tmp1 = *spcn++ * *xr1 + tmp2;
    *xr1 = *smcn++ * *xi1 + tmp2;
    *xi1 = tmp1;
  }
  xr1++; xi1++;
}

/* call rsrec again with half DFT length */
__pyrogen_rsrec(x, logm-1);

/* call complex DFT routine with quarter DFT length */
m =1 <<logm; m2 =m/2; m4 = 3 * (m/4);
__pyrogen_srrec(x+m2, x+m4,logm-2);

/* step 5. sign change and data reorder */
m = 1 <<logm; m2 = m/2; m4=m2/2; m8=m4/2;
xr1 = x + m2 + m4;
xr2 = x + m -1;
for (n=0;n<m8;n++) {
  tmp1 = *xr1;
  *xr1++ = - *xr2;
  *xr2-- = - tmp1;
}
xr1 = x + m2 + 1;
xr2 = x + m - 2;
for (n=0; n<m8;n++) {
  tmp1 = *xr1;
  *xr1++ = - *xr2;
  *xr2-- = tmp1;
  xr1++;
  xr2--;
  }
  if (logm == 2) x[3]=-x[3];
}

/* direct transform for real inputs */
void __pyrogen_fft(FLOAT32 *arr_r, FLOAT32 *ar
  /* call recursive routine */
  __pyrogen_rsrec(arr_r, logm);

  /* output array unshuffling using bit revers
  if (logm >1 ) {
    __pyrogen_BR_permute (arr_r,logm);
  }
}
/*****************************************
```

idct.c

```
#include <pyrogen.h>

/* idct.c, inverse fast discrete cosine transform                          */

/* Copyright (C) 1996, MPEG Software Simulation Group. All Rights Reserved. */

/*
 * Disclaimer of Warranty
 *
 * These software programs are available to the user without any license fee or
 * royalty on an "as is" basis.  The MPEG Software Simulation Group disclaims
 * any and all warranties, whether express, implied, or statuary, including any
 * implied warranties or merchantability or of fitness for a particular
 * purpose.  In no event shall the copyright-holder be liable for any
 * incidental, punitive, or consequential damages of any kind whatsoever
 * arising from the use of these programs.
 *
 * This disclaimer of warranty extends to the user of these programs and user's
 * customers, employees, agents, transferees, successors, and assigns.
 *
 * The MPEG Software Simulation Group does not represent or warrant that the
 * programs furnished hereunder are free of infringement of any third-party
 * patents.
 *
 * Commercial implementations of MPEG-1 and MPEG-2 video, including shareware,
 * are subject to royalty fees to patent holders.  Many of these patents are
 * general enough such that they are unavoidable regardless of implementation
 * design.
 *
 */

/**********************************************************/
/* inverse two dimensional DCT, Chen-Wang algorithm       */
/* (cf. IEEE ASSP-32, pp. 803-816, Aug. 1984)             */
/* 32-bit integer arithmetic (8 bit coefficients)         */
/* 11 mults, 29 adds per DCT                              */
/*                                     sE, 18.8.91        */
/**********************************************************/
/* coefficients extended to 12 bit for IEEE1180-1990      */
/* compliance                          sE,  2.1.94        */
/**********************************************************/

/* this code assumes >> to be a two's-complement arithmetic */
/* right shift: (-2)>>1 == -1 , (-3)>>1 == -2               */

#define W1 2841 /* 2048*sqrt(2)*cos(1*pi/16) */
#define W2 2676 /* 2048*sqrt(2)*cos(2*pi/16) */
#define W3 2408 /* 2048*sqrt(2)*cos(3*pi/16) */
#define W5 1609 /* 2048*sqrt(2)*cos(5*pi/16) */
#define W6 1108 /* 2048*sqrt(2)*cos(6*pi/16) */
#define W7 565  /* 2048*sqrt(2)*cos(7*pi/16) */

/* private data */
extern INT16 iclip[1024]; /* clipping table */
extern INT16 *iclp;


/* row (horizontal) IDCT
 *
 *           7                       pi         1
 * dst[k] = sum c[l] * src[l] * cos( -- * ( k + - ) * l )
 *          l=0                      8          2
 *
 * where: c[0]    = 128
 *        c[1..7] = 128*sqrt(2)
 */

static void __pyrogen_idctrow(INT16 *blk) {
  INT32 x0, x1, x2, x3, x4, x5, x6, x7, x8;

  /* shortcut */
  if (!((x1 = blk[4]<<11) | (x2 = blk[6]) | (x
        (x4 = blk[1]) | (x5 = blk[7]) | (x6 =
  {
    blk[0]=blk[1]=blk[2]=blk[3]=blk[4]=blk[5]=
    return;
  }

  x0 = (blk[0]<<11) + 128; /* for proper round

  /* first stage */
  x8 = W7*(x4+x5);
  x4 = x8 + (W1-W7)*x4;
  x5 = x8 - (W1+W7)*x5;
  x8 = W3*(x6+x7);
  x6 = x8 - (W3-W5)*x6;
  x7 = x8 - (W3+W5)*x7;

  /* second stage */
  x8 = x0 + x1;
  x0 -= x1;
  x1 = W6*(x3+x2);
  x2 = x1 - (W2+W6)*x2;
  x3 = x1 + (W2-W6)*x3;
  x1 = x4 + x6;
  x4 -= x6;
  x6 = x5 + x7;
  x5 -= x7;

  /* third stage */
  x7 = x8 + x3;
  x8 -= x3;
  x3 = x0 + x2;
  x0 -= x2;
  x2 = (181*(x4+x5)+128)>>8;
  x4 = (181*(x4-x5)+128)>>8;

  /* fourth stage */
  blk[0] = (x7+x1)>>8;
  blk[1] = (x3+x2)>>8;
  blk[2] = (x0+x4)>>8;
  blk[3] = (x8+x6)>>8;
  blk[4] = (x8-x6)>>8;
  blk[5] = (x0-x4)>>8;
  blk[6] = (x3-x2)>>8;
  blk[7] = (x7-x1)>>8;
}

/* column (vertical) IDCT
 *
 *             7                         pi
 * dst[8*k] = sum c[l] * src[8*l] * cos( -- *
 *            l=0                        8
 *
 * where: c[0]    = 1/1024
 *        c[1..7] = (1/1024)*sqrt(2)
 */
static void __pyrogen_idctcol(short *blk) {
```

idct.c

```
  INT32 x0, x1, x2, x3, x4, x5, x6, x7, x8;

  /* shortcut */

  if (!((x1 = (blk[8*4]<<8)) | (x2 = blk[8*6]) | (x3 = blk[8*2]) |
        (x4 = blk[8*1]) | (x5 = blk[8*7]) | (x6 = blk[8*5]) | (x7 = blk[8*3]))) {
    blk[8*0]=blk[8*1]=blk[8*2]=blk[8*3]=blk[8*4]=blk[8*5]=blk[8*6]=blk[8*7]=
      iclp[(blk[8*0]+32)>>6];
    return;
  }

  x0 = (blk[8*0]<<8) + 8192;

  /* first stage */
  x8 = W7*(x4+x5) + 4;
  x4 = (x8+(W1-W7)*x4)>>3;
  x5 = (x8-(W1+W7)*x5)>>3;
  x8 = W3*(x6+x7) + 4;
  x6 = (x8-(W3-W5)*x6)>>3;
  x7 = (x8-(W3+W5)*x7)>>3;

  /* second stage */
  x8 = x0 + x1;
  x0 -= x1;
  x1 = W6*(x3+x2) + 4;
  x2 = (x1-(W2+W6)*x2)>>3;
  x3 = (x1+(W2-W6)*x3)>>3;
  x1 = x4 + x6;
  x4 -= x6;
  x6 = x5 + x7;
  x5 -= x7;

  /* third stage */
  x7 = x8 + x3;
  x8 -= x3;
  x3 = x0 + x2;
  x0 -= x2;
  x2 = (181*(x4+x5)+128)>>8;
  x4 = (181*(x4-x5)+128)>>8;

  /* fourth stage */
  blk[8*0] = iclp[(x7+x1)>>14];
  blk[8*1] = iclp[(x3+x2)>>14];
  blk[8*2] = iclp[(x0+x4)>>14];
  blk[8*3] = iclp[(x8+x6)>>14];
  blk[8*4] = iclp[(x8-x6)>>14];
  blk[8*5] = iclp[(x0-x4)>>14];
  blk[8*6] = iclp[(x3-x2)>>14];
  blk[8*7] = iclp[(x7-x1)>>14];
}

/* two dimensional inverse discrete cosine transform */
void __pyrogen_idct(INT16 *idctBlock) {
  INT32 i;

  for (i=0; i<8; i++)
    __pyrogen_idctrow(idctBlock+8*i);

  for (i=0; i<8; i++)
    __pyrogen_idctcol(idctBlock+i);
}
```

max_val.c

```
#include <pyrogen.h>

/*****************************************************************************/
/* lame - ix_max                                                             */
/*****************************************************************************/
INT32 __pyrogen_max_val(INT32 arr[], INT32 begin, INT32 end) {
  int i, max = 0;

  for ( i = begin; i < end; i++ ) {
    int x = abs( arr[i] );
    if ( x > max )
      max = x;
  }
  return max;
}
```

mix.c

```
#include <pyrogen.h>

/* From Timidity++, mix.c*/
/* Timidity spends over 48% time mixing signals */
/* count and cc cover a wide range of values */

void __pyrogen_mix_stereo(INT16 **sp_p, INT32 **lp_p, INT32 count, INT32 channel_1, IN
T32 channel_2) {
  INT16 s;
  INT16 *sp = *(sp_p);
  INT32 *lp = *(lp_p);

  while (count--) {
    s = *sp++;
    *lp++ += s*channel_1;
    *lp++ += s*channel_2;
  }

  *(sp_p) = sp;
  *(lp_p) = lp;
}
```

quantize.c

```
#include <pyrogen.h>


int __pyrogen_nint( double in ) {
  int    temp;

  if( in < 0 )  temp = (int)(in - 0.5);
  else    temp = (int)(in + 0.5);

  return(temp);
}


void __pyrogen_quantize(INT32 l_end, FLOAT64 xr[], INT32 ix[], FLOAT64 *istep_p) {
  INT32 i;
  FLOAT32 istep = *istep_p;
  FLOAT32 temp;
  static INT32 init=0;
#define LUTABSIZE 10000
  static INT32 lutab[LUTABSIZE];

  if (init==0) {
    init++;
    for (i=0;i<LUTABSIZE;i++)
      lutab[i]=__pyrogen_nint(pow((FLOAT64)i/10.0,0.75)-0.0946);
    for (i=1;i<LUTABSIZE;i++)
      if ((lutab[i]-lutab[i-1])==1) {  /* we have a change over this interval */
        lutab[i]=-1;
        lutab[i-1]=-1;
        i++;
      }
  }

  for (i=0;i<l_end;i++) {
    temp=istep*fabs(xr[i]); /* step always positive -> temp always postive */

    if (temp<0.499996)
      ix[i]=0;
    else if (temp<1.862955)
      ix[i]=1;
    else if (temp<3.565282)
      ix[i]=2;
    else if (temp<5.506396)
      ix[i]=3;
    else if (temp<7.638304)
      ix[i]=4;
    else if (temp<9.931741)
      ix[i]=5;
    else if (temp<1000.0) {
      ix[i]=lutab[(int)(temp*10.0)];
      if (ix[i]==-1) /* too close to an interface, calculate exact value */
        ix[i] = (int)( sqrt(sqrt(temp)*temp) + 0.4054);
    }
    else {
      ix[i] = (int)( sqrt(sqrt(temp)*temp) + 0.4054);
    }
  }

  /* zero remaining elements */
  for (;i<576;i++)
    ix[i] = 0;
}
```

short_term_anal.c

```
#include <pyrogen.h>

#define MIN_WORD        ((-32767)-1)
#define MAX_WORD        ( 32767)

#define MIN_LONGWORD    ((-2147483647)-1)
#define MAX_LONGWORD    ( 2147483647)

#ifdef  SASR            /* >> is a signed arithmetic shift right */
#undef  SASR
#endif
#define SASR(x, by)     ((x) >> (by))

/*
 * #define GSM_MULT_R(a, b) (* INT16 a, INT16 b, !(a == b == MIN_WORD) *)       \
 *      (0x0FFFF & SASR(((INT32)(a) * (INT32)(b) + 16384), 15))
 */
#define GSM_MULT_R(a, b) /* INT16 a, INT16 b, !(a == b == MIN_WORD) */  \
        (SASR( ((INT32)(a) * (INT32)(b) + 16384), 15 ))

# define GSM_MULT(a,b)   /* INT16 a, INT16 b, !(a == b == MIN_WORD) */  \
        (SASR( ((INT32)(a) * (INT32)(b)), 15 ))

# define GSM_L_MULT(a, b) /* INT16 a, INT16 b */        \
        (((INT32)(a) * (INT32)(b)) << 1)

# define GSM_L_ADD(a, b)        \
        ( (a) <  0 ? ( (b) >= 0 ? (a) + (b)     \
                : (utmp = (UINT32)-((a) + 1) + (UINT32)-((b) + 1)) \
                  >= MAX_LONGWORD ? MIN_LONGWORD : -(INT32)utmp-2 )   \
        : ((b) <= 0 ? (a) + (b)   \
                  : (utmp = (UINT32)(a) + (UINT32)(b)) >= MAX_LONGWORD \
                    ? MAX_LONGWORD : utmp))

/*
 * # define GSM_ADD(a, b)       \
 *      ((ltmp = (INT32)(a) + (INT32)(b)) >= MAX_WORD \
 *      ? MAX_WORD : ltmp <= MIN_WORD ? MIN_WORD : ltmp)
 */
/* Nonportable, but faster: */

#define GSM_ADD(a, b)   \
        ((UINT32)((ltmp = (INT32)(a) + (INT32)(b)) - MIN_WORD) > \
                MAX_WORD - MIN_WORD ? (ltmp > 0 ? MAX_WORD : MIN_WORD) : ltmp)

# define GSM_SUB(a, b)  \
        ((ltmp = (INT32)(a) - (INT32)(b)) >= MAX_WORD \
        ? MAX_WORD : ltmp <= MIN_WORD ? MIN_WORD : ltmp)

# define GSM_ABS(a)     ((a) < 0 ? ((a) == MIN_WORD ? MAX_WORD : -(a)) : (a))


/*
 *  This procedure computes the short term residual signal d[..] to be fed
 *  to the RPE-LTP loop from the s[..] signal and from the local rp[..]
 *  array (quantized reflection coefficients).  As the call of this
 *  procedure can be done in many ways (see the interpolation of the LAR
 *  coefficient), it is assumed that the computation begins with index
 *  k_start (for arrays d[..] and s[..]) and stops with index k_end
 *  (k_start and k_end are defined in 4.2.9.1).  This procedure also
 *  needs to keep the array u[0..7] in memory for each call.
 */
void __pyrogen_short_term_analysis_filtering(register INT16 *u, register INT16 *rp, re
gister INT32 k_n, register INT16 *s) {
  register int          i;
  register INT16        di, zzz, ui, sav, rpi;
  register INT32        ltmp;

  for (; k_n--; s++) {
    di = sav = *s;
    for (i = 0; i < 8; i++) {           /* YYY

      ui    = u[i];
      rpi   = rp[i];
      u[i]  = sav;

      zzz   = GSM_MULT_R(rpi, di);
      sav   = GSM_ADD(   ui,  zzz);

      zzz   = GSM_MULT_R(rpi, ui);
      di    = GSM_ADD(   di,  zzz );
    }

    *s = di;
  }
}
```

short_term_filt.c

```
#include <pyrogen.h>

#define MIN_INT16       -32768
#define MAX_INT16       +32767
#define ROUND           +16384

#define clip16(x) ((x)<MIN_INT16) ? MIN_INT16 : ((x)>MAX_INT16) ? MAX_INT16 : (x)

/* k has two modes - 13/14 and 120/130 with the sample data file */
void __pyrogen_short_term_synthesis_filtering(INT16 *v, INT16 *rrp, INT32 k, INT16 *wt
, INT16 *sr) {
  INT32 i;
  INT16 sri, tmp1, tmp2;

  while(k--) {
    sri = *wt++;
    i=8;
    while(i--) {
      tmp1 = (rrp[i]*v[i] + ROUND)>>15;
      sri = clip16(sri - tmp1);
      tmp2 = (rrp[i]*sri + ROUND)>>15;
      v[i+1] = clip16(v[i] + tmp2);
    }
    v[0] = sri;
    *sr++ = sri;
  }
}

/* 4.2.10 */
/*
 *  This procedure computes the short term residual signal d[..] to be fed
 *  to the RPE-LTP loop from the s[..] signal and from the local rp[..]
 *  array (quantized reflection coefficients).  As the call of this
 *  procedure can be done in many ways (see the interpolation of the LAR
 *  coefficient), it is assumed that the computation begins with index
 *  k_start (for arrays d[..] and s[..]) and stops with index k_end
 *  (k_start and k_end are defined in 4.2.9.1).  This procedure also
 *  needs to keep the array u[0..7] in memory for each call.
 */
/* k_n has two modes - 13/14 and 120/130 with the sample data file */
void __pyrogen_short_term_analysis_filtering(INT16 *u, INT16 *rp, INT32 k_n, INT16 *s)
 {
  INT32 i;
  INT16 tmp1, tmp2;
  INT16 sav[9];
  INT16 d[9];

  while(k_n--) {
    sav[0] = *s;
    d[0] = *s;

    for(i=0; i<8; i++) {
      tmp1 = (rp[i]*d[i] + ROUND)>>15;
      sav[i+1] = clip16(u[i] + tmp1); /* sav[8] not used */

      tmp2 = (rp[i]*u[i] + ROUND)>>15;
      d[i+1] = clip16(d[i] + tmp2);  /* d[8] is the next value of s */
    }

    for(i=0; i<8; i++) {
      u[i] = sav[i];
    }

    *s++ = d[8];
  }
}
```

short_term_synth.c

```
#include <pyrogen.h>

#define MIN_WORD        ((-32767)-1)
#define MAX_WORD        ( 32767)

#define MIN_LONGWORD    ((-2147483647)-1)
#define MAX_LONGWORD    ( 2147483647)

#ifdef  SASR            /* >> is a signed arithmetic shift right */
#undef  SASR
#endif
#define SASR(x, by)     ((x) >> (by))

/*
 * #define GSM_MULT_R(a, b) (* INT16 a, INT16 b, !(a == b == MIN_WORD) *)       \
 *      (0x0FFFF & SASR(((INT32)(a) * (INT32)(b) + 16384), 15))
 */
#define GSM_MULT_R(a, b) /* INT16 a, INT16 b, !(a == b == MIN_WORD) */  \
        (SASR( ((INT32)(a) * (INT32)(b) + 16384), 15 ))

# define GSM_MULT(a,b)   /* INT16 a, INT16 b, !(a == b == MIN_WORD) */  \
        (SASR( ((INT32)(a) * (INT32)(b)), 15 ))

# define GSM_L_MULT(a, b) /* INT16 a, INT16 b */        \
        (((INT32)(a) * (INT32)(b)) << 1)

# define GSM_L_ADD(a, b)        \
        ( (a) <  0 ? ( (b) >= 0 ? (a) + (b)     \
                 : (utmp = (UINT32)-((a) + 1) + (UINT32)-((b) + 1)) \
                   >= MAX_LONGWORD ? MIN_LONGWORD : -(INT32)utmp-2 )   \
        : ((b) <= 0 ? (a) + (b)   \
                  : (utmp = (UINT32)(a) + (UINT32)(b)) >= MAX_LONGWORD \
                    ? MAX_LONGWORD : utmp))

/*
 * # define GSM_ADD(a, b)       \
 *      ((ltmp = (INT32)(a) + (INT32)(b)) >= MAX_WORD \
 *      ? MAX_WORD : ltmp <= MIN_WORD ? MIN_WORD : ltmp)
 */
/* Nonportable, but faster: */

#define GSM_ADD(a, b)   \
        ((UINT32)((ltmp = (INT32)(a) + (INT32)(b)) - MIN_WORD) > \
                MAX_WORD - MIN_WORD ? (ltmp > 0 ? MAX_WORD : MIN_WORD) : ltmp)

# define GSM_SUB(a, b)  \
        ((ltmp = (INT32)(a) - (INT32)(b)) >= MAX_WORD \
        ? MAX_WORD : ltmp <= MIN_WORD ? MIN_WORD : ltmp)

# define GSM_ABS(a)     ((a) < 0 ? ((a) == MIN_WORD ? MAX_WORD : -(a)) : (a))


/* k has two modes - 13/14 and 120/130 with the sample data file */
void __pyrogen_short_term_synthesis_filtering(register INT16 *v, register INT16 *rrp,
register INT32 k, register INT16 *wt, register INT16 *sr) {
  register int i;
  register INT16        sri, tmp1, tmp2;
  register INT32 ltmp;  /* for GSM_ADD  & GSM_SUB */

  while (k--) {
    sri = *wt++;
    for (i = 8; i--;) {

      /* sri = GSM_SUB( sri, gsm_mult_r( rrp[i], v[i] ) );
       */
      tmp1 = rrp[i];
      tmp2 = v[i];
      tmp2 =  ( tmp1 == MIN_WORD && tmp2 == MI
                ? MAX_WORD
                : 0x0FFFF & (( (INT32)tmp1 * (
                              + 16384) >> 15)

      sri  = GSM_SUB( sri, tmp2 );

      /* v[i+1] = GSM_ADD( v[i], gsm_mult_r( r
       */
      tmp1  = ( tmp1 == MIN_WORD && sri == MIN_
                ? MAX_WORD
                : 0x0FFFF & (( (INT32)tmp1 * (
                              + 16384) >> 15)

      v[i+1] = GSM_ADD( v[i], tmp1);
    }
    *sr++ = v[0] = sri;
  }
}
```

sub_sample_horiz.c

```
#include <pyrogen.h>

extern UINT8 *Clip;

/* horizontal filter and 2:1 subsampling */
void __pyrogen_conv444to422_mpeg2(UINT8 *src, UINT8 *dst, INT32 width, INT32 height) {
  int i, j, im5, im3, im1, ip1, ip3, ip5;

  for (j=0; j<height; j++) {
    for (i=0; i<width; i+=2) {
      im5 = (i<5) ? 0 : i-5;
      im3 = (i<3) ? 0 : i-3;
      im1 = (i<1) ? 0 : i-1;
      ip1 = (i<width-1) ? i+1 : width-1;
      ip3 = (i<width-3) ? i+3 : width-1;
      ip5 = (i<width-5) ? i+5 : width-1;

      /* FIR filter coefficients (*512): 22 0 -52 0 159 256 159 0 -52 0 22 */
      dst[i>>1] = Clip[(int)(22*(src[im5]   + src[ip5])
                            -52*(src[im3]  + src[ip3])
                           +159*(src[im1] + src[ip1])
                           +256*src[i]    + 256)>>9];
    }
    src+= width;
    dst+= width>>1;
  }
}
```

sub_sample_vert.c

```
#include <pyrogen.h>

extern UINT8 *Clip;

/* vertical filter and 2:1 subsampling */
void __pyrogen_conv422to420_frame(UINT8 *src, UINT8 *dst, INT32 width, INT32 height) {
  int w, i, j, jm5, jm4, jm3, jm2, jm1;
  int jp1, jp2, jp3, jp4, jp5, jp6;

  w = width>>1;

  /* intra frame */
  for (i=0; i<w; i++) {
    for (j=0; j<height; j+=2) {
      jm5 = (j<5) ? 0 : j-5;
      jm4 = (j<4) ? 0 : j-4;
      jm3 = (j<3) ? 0 : j-3;
      jm2 = (j<2) ? 0 : j-2;
      jm1 = (j<1) ? 0 : j-1;
      jp1 = (j<height-1) ? j+1 : height-1;
      jp2 = (j<height-2) ? j+2 : height-1;
      jp3 = (j<height-3) ? j+3 : height-1;
      jp4 = (j<height-4) ? j+4 : height-1;
      jp5 = (j<height-5) ? j+5 : height-1;
      jp6 = (j<height-6) ? j+6 : height-1;

      /* FIR filter with 0.5 sample interval phase shift */
      dst[w*(j>>1)] = Clip[(int)(228*(src[w*j] + src[w*jp1])
                                 +70*(src[w*jm1] + src[w*jp2])
                                 -37*(src[w*jm2] + src[w*jp3])
                                 -21*(src[w*jm3] + src[w*jp4])
                                 +11*(src[w*jm4] + src[w*jp5])
                                 + 5*(src[w*jm5] + src[w*jp6])+256)>>9];
    }
    src++;
    dst++;
  }
}
```

synth_filt.c

```
#include <pyrogen.h>

/* mpg123 synthesis filtering kernel               */

INT32 __pyrogen_synthfilt(FLOAT32 *window, FLOAT32 *b0, INT16 *samples, INT32 bo1) {
  INT32 j;
  INT32 clip = 0;
  const INT32 step = 2;
  FLOAT32 sum;

  for (j=16; j>0; j--) {
    sum  = *window++ * *b0++;
    sum -= *window++ * *b0++;
    sum += *window++ * *b0++;
    sum -= *window++ * *b0++;
    sum += *window++ * *b0++;
    sum -= *window++ * *b0++;
    sum += *window++ * *b0++;
    sum -= *window++ * *b0++;
    sum += *window++ * *b0++;
    sum -= *window++ * *b0++;
    sum += *window++ * *b0++;
    sum -= *window++ * *b0++;
    sum += *window++ * *b0++;
    sum -= *window++ * *b0++;
    sum += *window++ * *b0++;
    sum -= *window++ * *b0++;

    /* clip output sample to 16-bit signed integer */
    if(sum > 32767.0) {
      *samples = 32767;
      /* clip++; */
    }
    else if(sum < -32768.0) {
      *samples = -32768;
      /* clip++; */
    }
    else {
      *samples = sum;
    }

    window += 0x10;
    samples += step;
  }

  sum  = window[0x0] * b0[0x0];
  sum += window[0x2] * b0[0x2];
  sum += window[0x4] * b0[0x4];
  sum += window[0x6] * b0[0x6];
  sum += window[0x8] * b0[0x8];
  sum += window[0xA] * b0[0xA];
  sum += window[0xC] * b0[0xC];
  sum += window[0xE] * b0[0xE];

  /* clip output sample to 16-bit signed integer */
  if(sum > 32767.0) {
    *samples = 32767;
    /* clip++; */
  }
  else if(sum < -32768.0) {
      *samples = -32768;
      /*  clip++; */
  }
  else {
    *samples = sum;
  }

  b0 -= 0x10;
  window -= 0x20;
  samples += step;
  window += bo1<<1;

  for (j=15; j>0; j--) {
    sum = -*(--window) * *b0++;
    sum -= *(--window) * *b0++;
    sum -= *(--window) * *b0++;
    sum -= *(--window) * *b0++;
    sum -= *(--window) * *b0++;
    sum -= *(--window) * *b0++;
    sum -= *(--window) * *b0++;
    sum -= *(--window) * *b0++;
    sum -= *(--window) * *b0++;
    sum -= *(--window) * *b0++;
    sum -= *(--window) * *b0++;
    sum -= *(--window) * *b0++;
    sum -= *(--window) * *b0++;
    sum -= *(--window) * *b0++;
    sum -= *(--window) * *b0++;
    sum -= *(--window) * *b0++;

    /* clip output sample to 16-bit signed int
    if(sum > 32767.0) {
      *samples = 32767;
      /* clip++; */
    }
    else if(sum < -32768.0) {
      *samples = -32768;
      /* clip++; */
    }
    else {
      *samples = sum;
    }

    b0 -= 0x20;
    window -=0x10;
    samples += step;
  }

  return clip;
}
```

xform.c

```
#include <pyrogen.h>
#include <stdio.h>
/*
 * Apply a transformation matrix to an array of normal vectors:
 *   for i in 0 to n-1 do  v[i] = u[i] * m
 * where u[i] and v[i] are 3-element row vectors and m is a 16-element
 * transformation matrix.
 * The normals will be scaled to length 1.
 */

void __pyrogen_xform_normals_3fv_transform_normalize(INT32 n, FLOAT32 v[][3], const FL
OAT32 m[16], FLOAT32 u[][3]) {
  INT32 i;

  /* Transform and normalize */
  FLOAT32 m0 = m[0], m4 = m[4], m8 = m[8];
  FLOAT32 m1 = m[1], m5 = m[5], m9 = m[9];
  FLOAT32 m2 = m[2], m6 = m[6], m10 = m[10];

  for (i=0;i<n;i++) {
    FLOAT64 tx, ty, tz;
    {
      FLOAT32 ux = u[i][0],  uy = u[i][1],  uz = u[i][2];
      tx = ux * m0 + uy * m1 + uz * m2;
      ty = ux * m4 + uy * m5 + uz * m6;
      tz = ux * m8 + uy * m9 + uz * m10;
    }
    {
      FLOAT64 len, scale;
      len = sqrt( tx*tx + ty*ty + tz*tz );
      scale = (len>1E-30) ? (1.0 / len) : 1.0;
      v[i][0] = tx * scale;
      v[i][1] = ty * scale;
      v[i][2] = tz * scale;
    }
  }
}
```