NORTHEASTERN UNIVERSITY

Graduate School of Engineering

Thesis Title: Multimedia Macros for Portable Optimized Programs

Author: Juan Carlos Rojas

Department: Electrical and Computer Engineering

Approved for Thesis Requirement of the Doctor of Philosophy Degree

_____	_____
Thesis Advisor	Date
_____	_____
Thesis Committee Member	Date
_____	_____
Thesis Committee Member	Date
_____	_____
Thesis Committee Member	Date
_____	_____
Department Chair	Date
_____	_____
Director of the Graduate School	Date

NORTHEASTERN UNIVERSITY

Graduate School of Engineering

Thesis Title: Multimedia Macros for Portable Optimized Programs

Author: Juan Carlos Rojas

Department: Electrical and Computer Engineering

Approved for Thesis Requirement of the Doctor of Philosophy Degree

_____	_____
Thesis Advisor	Date
_____	_____
Thesis Committee Member	Date
_____	_____
Thesis Committee Member	Date
_____	_____
Thesis Committee Member	Date
_____	_____
Department Chair	Date
_____	_____
Director of the Graduate School	Date

Copy Deposited in Library:

_____	_____
Reference Librarian	Date

MULTIMEDIA MACROS FOR PORTABLE OPTIMIZED PROGRAMS

A Thesis Presented

by

Juan Carlos Rojas

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in the field of

Electrical Engineering

Northeastern University
Boston, Massachusetts

August 2003

ABSTRACT

MULTIMEDIA MACROS FOR PORTABLE
OPTIMIZED PROGRAMS

by

Juan Carlos Rojas

Northeastern University, Boston

Thesis Advisor: Professor Miriam Leeser

Multimedia processor architectures offer a combination of long partitioned registers and complex instructions that can speed up applications significantly when programmed manually. Optimized programs for these architectures have been non-portable up to now, because of differences in the instruction sets, register lengths, alignment requirements and programming styles. This dissertation presents a method that overcomes all these obstacles by providing a virtual instruction set common to a group of target architectures. This virtual instruction set is implemented as a library of C pre-processor macros called MMM. The macros can emulate long registers on architectures with short ones, and emulate complex instructions that are missing in certain targets.

This research is the first to provide a general solution to the portability of optimized multimedia programs. No other method to date allows an arbitrary program to take advantage of the complex partitioned operations available in multimedia instruction sets, while remaining portable.

MMM libraries were implemented for MMX & SSE, SSE2, AltiVec and TriMedia TM1300 multimedia architectures. Three examples from video compression were implemented in a portable way using MMM. The examples include IDCT, block distance for motion estimation, and block distance with interpolation. The portable examples were automatically translated into optimized code for each of the targets. Their performance is comparable, and in several cases better, than equivalent examples optimized by the processor vendors.

To Ericka, Laura and Sofía

CONTENTS

ILLUSTRATIONS

TABLES

Chapter 1

INTRODUCTION

Multimedia computing has been one of the greatest challenges in computer engineering for the last decade. Great efforts have been put into developing applications that process audio, video and graphics information. At the same time, computer designers have been challenged to come up with solutions capable of processing the enormous amounts of data required by multimedia applications. The solutions came in the form of multimedia processors, and multimedia extensions to general-purpose processors.

Nowadays, most commercial general-purpose processors support some form of multimedia extension. Some well-known examples are MMX extensions to Pentium processors, and AltiVec extensions to PowerPC. All multimedia architectures follow the same basic approach: they partition the registers into sections that represent multiple data elements, and operate on all the sections in parallel. In addition, they added complex instructions to speed-up specific tasks found in multimedia applications. For example, some architectures include an instruction to compute the sum of absolute differences of two vectors, which is useful in video compression.

1.1 Optimization vs. Portabilty

My experiments and other published results show that multimedia architectures can speed-up applications by factors of up to 15, but manual optimization is required in order to take full advantage of the complex instructions available. Manual optimization is very time consuming, and up to now has resulted in non-portable programs. This is in part because different multimedia architectures have different register lengths, different programming styles, different alignment requirements, and they support different partitioned instructions.

1.2 MMM

I solved the problem by creating MMM: a library of target-independent C pre-processor macros that implements a common set of parallel operations available or efficiently emulated on a given set of target architectures. The contents of the library depend on the set of target architectures used, but the method can be applied to any group of target processors.

MMM provides a unique interface to architectures with different register lengths and instruction sets. Long data vectors are simulated by several small vectors, and operations of long vectors are emulated as a sequence of operations on short vectors. Similarly, vector operations that are missing on a given target are emulated using a sequence of simple vector operations, when it is efficient to do so. The same concept is used to resolve different alignment requirements. Some architectures require that vector loads and stores are done at aligned addresses. If an unaligned load is required, one must load two aligned vectors, and compose the desired result from them. All this can be encapsulated inside an MMM load macro, and thus provide with a general unaligned load virtual instruction.

Through emulation, MMM implements a large common virtual instruction set for several target architectures. By using MMM, it is possible to write multimedia applications that are portable among different multimedia processors, and take advantage of the complex partitioned operations available on them. I used it to write optimized versions of Inverse Discrete Cosine Transform of 8x8 blocks, and several variants of L_1-Distance of 16x16 blocks.

MMM programs are portable among diverse multimedia architectures. Using MMM, I was able to generate optimized code for Pentium III with MMX and SSE extensions, Pentium 4 with SSE2 extensions, PowerPC G4 with AltiVec extensions, and Philips TriMedia multimedia processors, all from the same source program. The performance of my example programs is comparable, and in several cases exceeds that of hand-optimized versions of the same programs provided by the processor vendors.

1.3 Other Approaches

Parallelizing compilers can generate some multimedia instructions from scalar code, but not the most complex ones. The problem is that some these complex parallel instructions cannot be expressed compactly in C, only through a sequence of operations that is very hard for a compiler to recognize. One can also write parallel programs explicitly using a data-parallel language. But this still does not solve the problem of expressing complex parallel instructions. The other alternatives are to write applications based on optimized kernel libraries, or use automatic code generators from abstract descriptions. These are good solutions for certain kinds of applications, but not a general solution. MMM is a more flexible approach.

1.4 Contributions

This research is the first to provide a general solution to the portability of optimized multimedia code. No other method to date allows an arbitrary program to take advantage of the complex partitioned operations available in multimedia instruction sets, while remaining portable. MMM can be used to write complex programs that are portable, yet perform comparably to hand-optimized versions for a single target. Parallelizing compilers only obtain comparable performance on simple programs. MMM is a flexible, general framework for writing multimedia programs; other optimized libraries are made up of kernels with limited applicability.

Most research on code generation for multimedia architectures uses simple, inherently parallel programs as examples. I used complex examples taken from real multimedia applications, and demonstrated that they can be written efficiently using MMM.

This research can generate optimized code for different families of multimedia architectures. Others have focused on a single kind of architecture: some for MMX-like architectures (SSE, SSE2, 3DNow!), some others for AltiVec, and some others for TriMedia. I was able to generate optimized code for very different kinds of multimedia architectures: MMX, SSE, SSE2, AltiVec and TriMedia.

1.5 Organization of this Dissertation

This dissertation is organized into six chapters besides this introduction. Chapter 2 describes in more detail the problem addressed by this research, presents the solution in depth, and relates it to other research approaches. Chapter 3 covers the methodology: defines the research objectives, and explains the steps followed to validate that they have been met. Then Chapter 4 discusses the design of a virtual instruction set. The example programs are covered in Chapter 5, and Chapter 6 presents the performance measurement results. Chapter 7 has conclusions, and advances on future research work.

In addition, there are three appendices. Appendix A shows the complete definition of the virtual instruction set. Appendix B has the actual implementation of the MMM library for the different targets, and Appendix C is the source code of the portable examples written in MMM.

Chapter 2

PROBLEM DESCRIPTION

This chapter discusses in depth the problem addressed by this dissertation, the solution presented, and related research. The first section introduces characteristics of multimedia architectures and how they are programmed. Section 2.2 describes the factors that make the portability of optimized programs a problem. Next, Section 2.3 explains how MMM can solve all these portability problems. Section 2.5 compares MMM to other approaches to the same problem.

2.1 Background

Multimedia applications are computationally very intensive for general-purpose processors, as they have to process enormous amounts of data. Processor designers have responded by adding multimedia instruction sets with partitioned registers and parallel SIMD instructions, including some complex instructions specifically tailored for multimedia applications. Table 2.1 shows a list of popular architectures that have multimedia instruction sets. They come in the form of multimedia extensions to general-purpose processors, or as special-purpose multimedia processors. There is a large variation in the length of the multimedia registers in these processors, from 32 bits to 128 bits.

Table 2.1
Popular processors that have multimedia instruction sets

Instruction Set	Architecture Type	Register Length	Reference
SSE2	Multimedia extensions to Intel Pentium 4 processors	128 bits	[37]
MMX + SSE	Multimedia extensions to Intel Pentium III and later processors	64 bits for integer 128 bits for floating point	[37]
AltiVec	Multimedia extensions to Motorola PowerPC G4 processors	128 bits	[36]
Enhanced 3DNow!	Multimedia extensions to AMD Athlon processors	64 bits	[38]
VIS	Multimedia extensions to SUN UltraSparc processors	64 bits	[39]
Phillips TriMedia TM1300	Multimedia processor	32 bits	[35]
Equator MAP-CA	Multimedia processor	64 and 128 bits	[40]

Multimedia data elements can often be represented by 8-bit or 16-bit integers. For example, image pixels are represented by 8 bits for each color component. It is possible to hold 16 pixels in a single 128-bit register, and operate on all of them in parallel. Multimedia architectures have been designed specifically to take advantage of this parallelism, by using long registers and partitioned instructions. While longer registers have a greater potential for speedup, it is not always possible to take full advantage of them; it depends on the amount of parallelism available in the algorithm.

Multimedia processors vary in the instructions they implement. All of the processors in Table 2.1 support basic integer arithmetic and logical instructions on registers partitioned into 8, 16 and 32-bit sections. Many support complex instructions like *sad* (sum of absolute differences), and *multiply-add-pairs* (parallel multiply and add adjacent pairs of products). Some support parallel floating-point operations too. Table 2.2 shows some of the complex parallel operations present in multimedia instruction sets.

Table 2.2
Some complex parallel instructions supported by multimedia architectures

Instruction	SSE2	MMX + SSE	AltiVec	Enhanced 3DNow!	VIS	TM1300	MAP-CA
sad (sum of absolute differences) of 8-bit integers	√	√		√	√	√	√
multiply-add-pairs of 16-bit integers	√	√	√	√		√	
multiply-high of 16-bit integers	√	√	√	√			√
Average of 8-bit integers	√	√	√	√		√	√
maximum and *minimum* of 8-bit integers	√	√	√	√		√	√

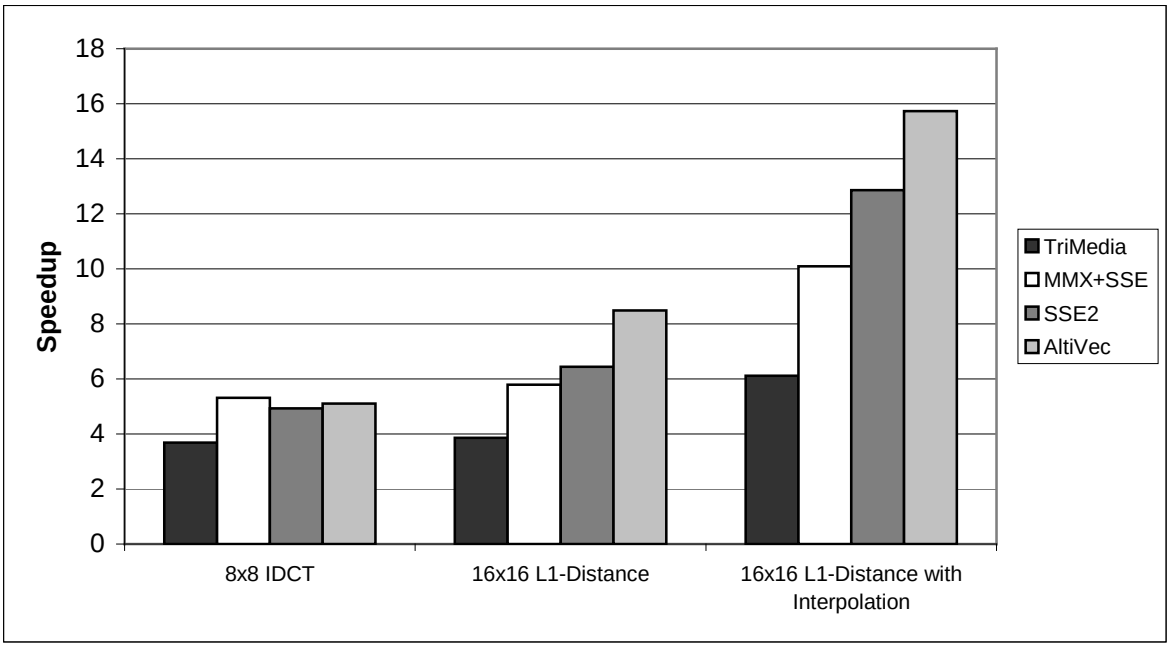
Optimized multimedia programs take advantage of the complex partitioned operations available on the target architecture, to obtain significant speedups with respect to scalar implementations. The speedup that can be obtained by using multimedia instruction sets varies, depending on the architecture and the application. Published results range from no speedup, up to factors of 12 for manually optimized multimedia and signal processing kernels. Selected published results are listed in Table 2.3. Refer to the Glossary at the end of this dissertation for definitions of the acronyms in this table.

Table 2.3
Published results for speedup obtained by hand-optimization using multimedia instruction sets

Benchmark	Target	Speedup	Reference
FIR	VIS	3.43	[1]
MPEG encoder	VIS	3.1	[2]
MPEG2 decoder	MMX	1.4 – 1.5	[3]
IDCT	MMX	3.25 – 4.37	[3]
H.263 encoder	MMX	1.67	[4]
FFT	MMX	1.98	[5]
Motion Estimation with Interpolation	MMX	3.1	[6]
IDCT	AltiVec	11.7	[7]
FIR	AltiVec	3.1	[8]

My own research shows that speedups of up to a factor of 15 can be obtained through manual optimization on different multimedia architectures. Figure 2.1 compares the speedup obtained by using complex partitioned instructions for several multimedia kernels on different architectures. More details about these measurements are available in Chapter 6.

Figure 2.1
Speedup of hand-optimized multimedia kernels over scalar versions.



Optimized multimedia programs are usually written in extended versions of C. Partitioned instructions are expressed by macros or functions called intrinsics. The alternatives to writing optimized programs in C with intrinsics are to write them in assembly, or to use libraries, a vectorizing compiler, or an automatic code generator. These approaches are discussed in Section 2.4.

Development environments for different multimedia architectures have different styles to define parallel data and operations. Table 2.3 shows several styles for a simple vector declaration and parallel addition. AltiVec uses the `vector` type qualifier to define vectors of basic types; operations infer the partition size from the type. TriMedia uses integers to represent vectors, and the partition sizes are specified by the operations. Intel supports both models: it has a set of C intrinsics that specify partition sizes, and also overloaded C++ operators for vector classes that infer the partition size from the argument type.

Table 2.3
Different styles for declaration and operations on partitioned data

Architecture	Example
TriMedia	`int A, B, C;  /*Each variable represents a vector of 2 16-bit values*/` `A = DSPIDUALADD(B, C);  /*Parallel add*/`
AltiVec	`vector short A, B, C;  /*Each variable represents a vector of 8 16-bit values*/` `A = vec_add(B, C);  /*Parallel add*/`
Intel C intrinsics	`__m64 A, B, C;  /*Each variable represents a vector of 4 16-bit values*/` `A = _mm_add_pi16(B, C); /*Parallel add */`
Intel C++ vector classes	`I16vec8 A, B, C;  /*Each variable represents a vector of 8 16-bit values*/` `A = B + C;  /*Parallel add */`

2.2 Problem

Multimedia programs written in C can be optimized to take advantage of the complex partitioned operations available in multimedia instruction sets by using intrinsics. These optimized programs are not portable to other architectures, even if the instruction sets are similar. Differences in register lengths, instructions supported, data alignment requirements and programming styles are obstacles to portability. Portable programs are desirable, but up to now there has been no way to make complex portable programs run as fast as hand-optimized ones.

The length of the registers on current multimedia architectures can vary from 32 to 128 bits, as shown in Table 2.1. For highly parallel algorithms and large input blocks, optimized programs iterate over the input data in sections the size of the registers. The number of iterations is inversely proportional to the register length.

The available partitioned instructions vary from architecture to architecture. For example, the *sad* instruction for 8-bit partitions is available on many processors, but not on AltiVec. The *multiply-high* instruction is available on TriMedia for 8-bit partitions, but not for 16-bit partitions. SSE and SSE2 support *multiply-high* on 16 and 32-bit partitions, and one operand can be a memory address. AltiVec has a variant of this instruction for 16-bit partitions only, where it adds the 17 most significant bits of the product, adds it to the corresponding partition of a third input vector, and returns the saturated 16-bit result.

Some processors require aligned vector loads and stores. For example, TriMedia can only read 32-bit words from addresses that are in a 32-bit boundary (the last 5 bits of the address are zero). Similarly, AltiVec can only read and write vectors on addresses that are 128-bit aligned. If one needs to load a vector from an unaligned address, one needs to load two vectors and extract the desired data from them. Intel SSE can read from unaligned addresses without any restriction. SSE2 has, in addition, fast load/store instructions for 128-bit aligned addresses.

There are obvious advantages from optimized programs. They can boost the performance of high-end processors, or perform equivalent tasks on lower-cost processors. But there are also benefits to portability. Optimizing programs is an expensive, time-consuming job, which has to be repeated for every target architecture. Having multiple versions of a program is hard to maintain and is prone to errors.

Scalar C programs are portable, but not optimized. Even parallelizing compilers cannot fully take advantage of the complex partitioned operations available in multimedia instruction sets. Compilers can generate basic parallel operations, like additions of floating-point multiplications on partitioned registers, but not complex ones. The problem is that C lacks syntax to explicitly express many complex operations available in DSP and multimedia instruction sets, like *multiply-high*, *sad*, or saturating arithmetic operations. Such instructions can only be expressed through a complex sequence of operations that is very hard for a compiler to recognize. Consider the case of the sum of absolute differences (*sad*) of two vectors of 16 8-bit integers. To express this operation in C would require a loop, in which the absolute value of the difference of the elements is summed:

```
sad = 0;
for (i=0; i<16; i++)
{
    diff = a[i] – b[i];
    sad += diff > 0 ? diff : -diff;
}
```

Several other representations are also possible for this operation. Complex operations like *sad* are awkward to write in C, and hard for compilers to recognize.

The only way to exploit the full potential of multimedia processors is to program in C with intrinsics or in assembly. Processor vendors provide compilers that support intrinsics for their own processors. Each architecture follows its own style, as was seen in the examples in Table 2.3.

2.3 Solution

Multimedia architectures differ in certain aspects, but are also similar in many ways. They are all programmable in C with intrinsics, the register lengths are all multiples of basic types, and similar partitioned instructions exist on them. Even if the instruction sets are not identical, it is often possible to emulate the missing instructions efficiently with a sequence of the available instructions. Similarly, parallel operations on long registers can be emulated with a sequence of operations on short registers.

My solution is to create a library of target-independent C pre-processor macros called MMM – for Multimedia Macros – that implements a common set of parallel operations available or efficiently emulated on a given set of target architectures. The programs use MMM macros as virtual instructions, which get translated by the libraries to C code with intrinsics for each target architecture. The C output is compiled to a program executable by the regular C compiler provided by each processor vendor. By using MMM, it is possible to write multimedia applications that are portable among different multimedia processors, and take advantage of the complex partitioned operations available on them.

MMM makes it possible to create portable programs for target architectures that have different register lengths. A long vector can be represented by several short vectors, and operations on long vectors can be emulated by repeated operations on short vectors. For example, if you wanted to load and add two arrays of 8 16-byte integers, you would write it in MMM as:

```
DECLARE_I16x8(A);
DECLARE_I16x8(B);
DECLARE_I16x8(C);
```

```
LOAD_A_I16x8(A, pSrcA);
LOAD_A_I16x8(B, pSrcB);

ADD_I16x8(C, A, B);
```

In this example, *A, B* and *C* represent 128-bit vectors, and the loads are from aligned addreses. The mapping of these macros to an architecture with 128-bit registers is straightforward. For example, the implementation of these macros for SSE2 is:

```
#define DECLARE_I16x8(var) \
    __m128i var;

#define LOAD_A_I16x8(var, ptr) \
    var = _mm_load_si128((__m128i *) (ptr));

#define ADD_I16x8(dst, src1, src2) \
    dst = _mm_add_epi16(src1, src2);
```

On architectures with registers smaller than 128 bits, these vectors are represented by several variables. TriMedia has 32-bit registers, so it needs to use four variables to represent each vector, and replicate the operations four times:

```
#define DECLARE_I16x8(var) \
    unsigned int var##_0; \
    unsigned int var##_1; \
    unsigned int var##_2; \
    unsigned int var##_3;

#define LOAD_A_I16x8(var, ptr) \
    var##_0 = *((int *) (ptr)); \
    var##_1 = *(((int *) (ptr))+1); \
    var##_2 = *(((int *) (ptr))+2); \
    var##_3 = *(((int *) (ptr))+3);

#define ADD_I16x8(dst, src1, src2) \
    dst##_0 = DSPIDUALADD(src1##_0, src2##_0); \
    dst##_1 = DSPIDUALADD(src1##_1, src2##_1); \
    dst##_2 = DSPIDUALADD(src1##_2, src2##_2); \
    dst##_3 = DSPIDUALADD(src1##_3, src2##_3);
```

The pre-processor construct ## represents concatenation, so four different variable names are generated from the macros. For example, `DECLARE_I16x8(A)` gets resolved as:

```
unsigned int A_0;
unsigned int A_1;
```

```
unsigned int A_2;
unsigned int A_3;
```

This technique uses several local variables to represent vectors. This is not a problem for register scheduling, because the variables are independent of each other and the compiler can schedule several of them into the same registers. The replication of the operations is equivalent to loop unrolling, a technique that many hand-optimized programs use anyway.

Some processors don't support certain instructions available in other ones. However, it is often possible to emulate these instructions efficiently with a sequence of operations. The idea is to use emulation only to simulate an instruction available in one of the target processors, in order to maintain the libraries at the instruction level and maximize reusability. For example, a *sad* instruction is not available in AltiVec, but can be emulated by using parallel *maximum*, *minimum* and *subtract*, followed by a sum of vector elements. Below is a simplified implementation of *sad* on AltiVec. Two operations are required to sum all 16 elements of the vector:

```
#define SAD_U8x16(dst, src1, src2) \
    dst = vec_sums( vec_sum4s( vec_sub( \
          vec_max(src1, src2), vec_min(src1, src2))));
```

Differences in alignment requirements can also be overcome by using MMM. Separate macros for aligned and unaligned loads and stores allow the programmer to avoid re-alignment overhead when it is not required. Unaligned loads can be implemented with a sequence of operations that extract the unaligned data from two aligned vectors. For example, TriMedia requires word loads to be from 32-bit aligned addresses. If an address is unaligned, the load behaves as if the lowest 5 bits of the address were zero. Unaligned loads can be implemented by loading two words beginning at the previous 32-bit boundary, and extracting the desired word with *shift*s and *or*s:

```
#define LOAD_U_U8x4(var, ptr)                                  \
{                                                              \
    int shift_right = (((int) (ptr)) & 0x3)<<3;                \
    var = ((*( ((int *) ptr)+1)) << (32 – shift_right) |       \
           (*( ((int *) ptr)   ) >> shift_right);              \
}
```

MMM can overcome the differences in programming styles for different architectures by providing a common set of macros to define and manipulate partitioned data. The examples above show how vectors can be declared and loaded in a machine-independent fashion. Other style-dependent manipulations, like setting vectors constants, or allocating aligned memory, can also be handled this way.

This research is focused on the problem of taking advantage of the complex parallel operations in multimedia instruction sets. There are other factors that affect the performance of a program, like the size of the caches, speed of the memory relative the CPU, instruction pipeline structure, operating system overhead, and compiler quality. This research does not attempt to address them. Sub-section 2.4.4 discusses complementary approaches that can deal with these issues.

2.4 Related Work

Researchers have approached the problem of portability of optimized code from four different angles: parallelizing compilers, data-parallel languages, optimized libraries, and automatic generation of optimized code from abstract descriptions. The next sub-sections analyze each of these approaches and describe the state of current research in these fields.

2.4.1 Parallelizing Compilers

A lot of research has been focused on generating partitioned instructions from scalar loops. There are several commercial and experimental compilers that can parallelize code to some

degree. Highly-parallel programs, like vector and matrix multiplications, dot products and linear equation solvers, can be efficiently parallelized by compilers. More complex applications like IDCT or L_1-Distance of blocks cannot. Compilers achieve only modest speedups on these types of kernels, if any.

A vectorizing compiler for MMX by Sreraman and Govindarajan [9] reports to have vectorized an L_1-Distance loop, but as a sequence of simple parallel operations, and not using the *sad* instruction available in MMX+SSE.

Lorenz, Wehmeyer and Dräger [10] report success in vectorizing dot-product loops, but not convolution or FIR kernels on their compiler targeted at the M3 DSP processor. Larsen and Amarasinghe developed a vectorizing compiler for AltiVec [11]. Their speedup results are good for inherently-parallel programs like color conversion, but a modest 1.24 to 1.57 on FIR, IIR, and SPECfp kernels.

Leupers [12] developed a parallelizing compiler for TriMedia and TI C62xx processors. This compiler is able to recognize sum of products patterns, and thus is able to parallelize FIR filters for TriMedia. It can get 1.2 to 1.3 speedups on IIR and convolution for C62xx, but no speedup on these kernels for TriMedia.

The commercially available Intel C/C++ compiler reports good speedup results for dot products, vector-matrix and scalar-matrix products, LU factorization, and linear equation solving for SSE and SSE2 [13][14]. The speedups for SpecCPU benchmarks ranges from 1.03 to 1.23. My own experiments with version 7.0 of this compiler show that it cannot parallelize IDCT or L_1-Distance kernels.

VectorC by Codeplay [15] is a vectorizing compiler aimed at games programming for MMX, SSE, SSE2 and 3DNow! It reports speedups from 1.5 to 2.9 on vector rotations, normalizations and projections. No results are reported for more complex examples.

Other experimental vectorizing compilers for VIS [16][17] only report successful vectorization of simple, single assignment loops. There are other commercial vectorizing compilers for AltiVec, by Green Hills Software and by Veridian Systems, and by The Portland Group for MMX, SSE, SSE2 and 3DNow!, but no speedup results are published.

Vectorizing compilers are an active area of research, and will undoubtedly improve in the future. They are a good solution for inherently parallel algorithms, like those in linear algebra, and to achieve modest speedups on existing scalar code. But vectorizing compilers are restricted by the lack of syntax in the C language to express complex operations available in multimedia instruction sets. In general, scalar C programs cannot achieve speeds comparable to hand-optimized versions.

2.4.2 Data-Parallel Languages

Data-parallel languages allow definition of parallel data types of different shapes; operations on parallel variables are defined to be parallel operations on each of the elements. This maps well to SIMD architectures of different sorts, from multimedia processors to massively parallel computers. In addition to strictly parallel operations, data-parallel languages support broadcast and reduction operations between scalar and parallel variables.

A number of data-parallel languages have been defined for different kinds of computers. Any of these languages can express basic parallel arithmetic and logic operations, but cannot explicitly express complex operations like `multiply-high` or `sad`.

Fortran 90 can define array types and operate natively on them [18]. C* supports parallel types of arbitrary shapes [19]. In both of these languages, parallel operations are limited to basic arithmetic and logic, plus *minimum* and *maximum*.

Vector Pascal [20] and SWARC [21][22] were designed specifically for multimedia instruction sets. They add syntax to express some more of the partitioned operations available in these architectures. SWARC supports parallel *average*, as well as saturation arithmetic to handle overflows. Vector Pascal supports saturating *add*s and *subtracts*, and allows user-defined unary functions to operate on vector variables. Although SWARC and Vector Pascal are richer than other data-parallel languages, they still cannot express many complex parallel operations like *multiply-high* or *sad*.

Some languages have been designed to express some of the complex operations typically available in DSP processors. For example, ISO Embedded C [23] provides native types for fixed-point variables, with qualifiers to specify either saturation or modulo arithmetic handling of overflow. It also defines native functions for *absolute-value*, *round* and *count-bits*. Using fixed-point types one can express a *multiply-high* operation, and can write a *sad* operation more concisely than in standard C, by using the *absolute-value* operator. Embedded C is a scalar language, so SIMD operations on vectors can only be expressed through a loop.

MMM is a macro library, but in a sense it is a data-parallel language. It can express vectors of data and parallel operations on them. It is different from others languages in that can express all kinds of complex parallel operations, and that it uses C pre-processor macros instead of language extensions. Using C pre-processor macros gives MMM an enormous flexibility to

expand as needed, and thus is useful in experimentation. The translation of MMM programs into C with target-specific intrinsics is very simple, and doesn't require a compiler.

The concept of C pre-processor macros as a portable language has been used before. Franchetti and Püschel [24] used macros to represent parallel loads/stores, parallel floating-point arithmetic operators, and permutations. MMM extends this idea to more complex operations, introduces the concept of emulation of instructions, and emulation of longer register lengths.

Partitioned data and operations can also be expressed with C++ classes and overloaded operators. This is the case of Intel's C++ SIMD Class Libraries [26]. C++ classes are defined for specific combinations of vector lengths and data types, and operators are overloaded to work on these types. For example, the `F32vec4` class represents vectors of 4 32-bit floating-point elements. Overloaded operators exist for loads, stores, standard logic, arithmetic and shift operations, `saturating-add` and `subtract`, `sum-vector-elements`, `maximum`, `minimum`, `average`, parallel comparisons, data packing, conversions between floating-point and integer, `multiply-high`, `multiply-add`, `square-root`, and `complex-reciprocal`.

The classes and operations implemented by this library match part of the Intel MMX, SSE and SSE2 instruction sets. There are still some instructions in these architectures that are not implemented as overloaded operators, like `sad`, `multiply-add-pairs`, and permutations. Operators are only implemented for the classes that there is hardware support for; there is no emulation. For example, parallel multiplication is available for floats and for 16-bit integers, but not for 8 or 32-bit integers. Also, 128-bit vector classes and operators are only

implemented for SSE2, and not for MMX and SSE, so a program written with 128-bit classes does not run on a processor that does not have SSE2.

MMM and Intel C++ SIMD classes share the same philosophy: they implement a common interface for the instruction sets of different architectures; both implement only vector lengths and element types that are supported by hardware, not arbitrary lengths. The difference is that MMM emulates longer vector lengths on architectures with short registers, and complex operations on architectures that don't have them, when it is efficient to do so. A program written with 128-bit vectors using MMM can run on processors with 64 or 32-bit SIMD registers. One advantage of C++ classes over MMM is that it overloads operators for different vector lengths and types, which makes the syntax more elegant. MMM cannot overload, so a different macro must be used for each vector length and type. This is acceptable for this research, as it is just a matter of style. A C language extension for multimedia, with overloaded operations is proposed as future work in Chapter 7.

2.4.3 Optimized Libraries

An alternative way to write portable optimized programs is to base the application on libraries of kernels that have been hand-optimized for the different targets. The problem of this approach is that developing and maintaining a large number of libraries for a large number of targets is very laborious and expensive. Also, these libraries are inflexible; there is no room for customization.

There are some examples of libraries optimized for multiple targets, for specific applications. The most notorious is BLAS [27]. BLAS is a set of basic floating-point vector-vector, vector-matrix and matrix-matrix operations, which serves as a base for various linear algebra

packages, like LAPACK and LINPACK. BLAS has been optimized for virtually every processor by the vendors or users. Due to the parallel nature of the operations in BLAS, very efficient implementations can be achieved by using parallel operations in multimedia instruction sets. This library is useful for scientific computation, and no so much for multimedia applications.

More applicable for multimedia are Intel's Integrated Performance Primitives [28]. This library includes kernels for signal, image, speech, graphics and audio processing, and operates on vectors or matrices of integer or floating-point data. Optimized versions of these libraries are available for all current Intel architectures, including MMX, SSE, SSE2 and XScale. Naturally, this library is only available for Intel processors. Some other vendors have their own libraries of signal processing kernels.

An effort to consolidate signal-processing libraries to a unique API is VSIPL [29]. VSIPL is a standard for a very complete library of signal and image processing kernels, operating on integer and floating point types of various precisions. This library supports signal processing operations, like FFT, FIR and IIR filters, convolution, correlation, as well as arithmetic, logic and linear algebra on one, two and three-dimensional arrays. An interesting concept in VSIPL is the portable precision types, where the minimum precision required is specified. This allows an implementation to use a more precise type when it is more efficient. Implementations of VSIPL are done by different vendors, conforming to the standard API. Currently, the basic VSIPL profile has been implemented for SSE and AltiVec multimedia architectures.

MMM is a library, but at the instruction level, rather than at the kernel level. MMM operations are much more reusable than the libraries above. Kernel libraries like BLAS and VSIPL could actually be built based on MMM macros.

Another low-level vector library is CVL [30]. It provides a set of parallel arithmetic, logic and comparison operations, reductions and permutations for arbitrary length vectors. CVL serves as a machine-independent interface for higher-level data parallel languages like NESL. CVL is intended for scientific applications on massively parallel architectures, and has been optimized for CM-2, CM-5 and Cray Y-MP computers. CVL could be implemented on uniprocessor multimedia architectures, but it would suffer from the high overhead of a function call per vector operation.

2.4.4 Code Generation from Abstract Descriptions

Optimized code can sometimes be generated from abstract descriptions of an algorithm. Franchetti and Püschel explored this approach for matrix transformations in their SPIRAL project [24][25]. They generate multiple implementations of a given matrix transformation iteratively, searching for the best run-time performance. Their system decomposes the matrix into operators that are vectorizable, and generates the appropriate partitioned instructions. Their output is C code with macros that represent parallel loads/stores, parallel floating-point arithmetic, and permutations. The macros can be resolved to intrinsics for different architectures. They have currently implemented the macros for SSE and SSE2.

A similar method is used by FFTW [31][32] for the generation of FFT and similar transforms. The transforms are decomposed into "codelets" of different sizes according to a plan. The execution time of multiple plans is compared in search for the optimal one. Distributed with

the system comes a library of codelets, which were either hand-coded, or automatically generated a priori. The most recent version of FFTW can take advantage of SIMD instructions in SSE, SSE2, 3DNow! and AltiVec. It does so by using generic parallel instructions in SIMD versions of the codelets. The generic instructions are translated to specific architecture instructions by the code generator, according to a description file. FFTW uses parallel *load*, *store*, *add*, *subtract*, *multiply-add*, *mutiply-subtract*, *unpack*, and *permute* on floating-point elements.

Another self-optimization project is ATLAS [33]. It generates adapted implementations of the BLAS library of linear-algebra kernels, and applies dynamic programming to search for the plan with optimal execution time. The code generator can vary several parameters, like the minimum block size that fits in registers, loop unrolling factor, support for *multiply-add* instructions, and fetch patterns. ATLAS supports SIMD instructions, based on hand-coded kernel libraries provided by the user community.

Feedback-based automatic code generators like SPIRAL, FFTW and ATLAS can optimize for many aspects of a computer's architecture, like the size of the caches, number and size of registers, and support of certain instructions. MMM can complement this approach by providing a common interface to the instruction sets of different architectures. As a matter of fact, SPIRAL uses C pre-processor macros, much like MMM, to represent parallel operations in different architectures. SPIRAL and FFTW only use a small subset of multimedia instruction sets. MMM implements a larger common set of instructions, because it emulates complex instructions on architectures that don't have them. As a result, MMM provides a much richer set of machine-independent instructions that a code generator could use.

2.4.5 Other Related Research

Some researchers have experimented with emulation of parallel operations on architectures that do not have explicit support for it, or to further subdivide existing partitions into smaller ones. Fisher and Dietz [21] describe how it is possible to execute parallel additions and subtractions without risk of carryover, by separating the elements with spacer bits. Zucker and Lee implemented partitioned addition, subtraction and multiplication by a scalar using floating-point instructions [34]. These techniques can easily be implemented within MMM macros.

2.5 Summary

Multimedia architectures can have different register lengths, alignment requirements, programming styles, and support different partitioned instructions. All these are obstacles to portability, but can be overcome by using MMM: a set of target-independent C pre-processor macros that provide an interface to the different target architectures. MMM emulates long vectors on architectures with short registers, and emulates complex instructions that are missing on some processors. MMM programs can be portable and optimized at the same time.

Other approaches are parallelizing compilers, data-parallel languages, optimized libraries, and automatic code generation from abstract descriptions. None of these methods provide the same level of performance and flexibility as MMM.

The next chapter describes the objectives and methodology used to validate MMM as a solution to the problem of portability of optimized code.

Chapter 3

RESEARCH

This chapter describes the objectives and methodology followed by this research. Section 3.1 defines in detail the objectives that are addressed by MMM: portability and performance. Section 3.2 goes through all the steps that were followed in order to validate the objectives stated.

3.1 Objectives

The goal of this research is to validate MMM as a solution that allows multimedia programs to be portable and optimized at the same time. There are two major parts to this claim: that MMM programs are portable among diverse multimedia architectures, and that they have good performance on all the targets. The next two sub-sections elaborate more on these two objectives. Ease of programming is not an objective of this research, but will be addressed as future work in Chapter 7.

3.1.1 Portability

By portable I define a program with a single source, without machine-specific sections, that can be compiled for different targets and produce the desired results. The type of portability that MMM accomplishes in not unlimited, an MMM program will not necessarily be portable to all current and future multimedia architectures, while remaining optimized. But MMM should provide portability among several diverse architectures that would otherwise be incompatible.

I say desired results, and not identical results, because it may be possible to approximate an operation in a way that is not bit-exact, but close enough for practical purposes. For example, one implementation may use more precision in the multiplications than required. This is fine as long as there is a clear criteria defining what the desired results are.

The only machine-specific section that is allowed in portable MMM programs is the inclusion of the MMM header file for the current target. The header files for each target are conditionally included based on an environment definition:

```
#ifdef SSE2
    #include "mmm_sse2.h"
#endif
#ifdef SSE
    #include "mmm_sse.h"
#endif
#ifdef TRIMEDIA
    #include "mmm_tm.h"
#endif
#ifdef ALTIVEC
    #include "mmm_altivec.h"
#endif
```

3.1.2 Performance

In the context of this research, an optimized program is one that makes efficient use of the target's instruction set in order to reduce the number of instructions necessary to perform the task. Although the ultimate goal of optimization is to reduce the execution time, there are factors that affect it, like the memory structure and instruction pipeline interactions, which are beyond the scope of MMM. I use both instruction counts and execution speed as measures of performance, and attempt to minimize the effects of the memory structure on my experiments.

The performance of MMM optimized programs should be better than equivalent scalar programs, even when compiled with a parallelizing compiler. It is not expected that MMM programs out-perform hand-optimized programs for a single target, but they should come close. An objective of this research is to determine how much performance is lost in order to obtain portability.

3.2 Methodology

The rest of this chapter outlines a sequence of steps that I followed in order to validate that MMM meets the objectives stated above. The steps include selecting a diverse group of target architectures, defining and implementing a common virtual instruction set, selecting and implementing several example programs and comparing their performance.

3.2.1 Target Architecture Selection

I selected four different target architectures with multimedia instruction sets. They are: the TriMedia TM1300 media-processor [35], AltiVec extensions to the PowerPC G4 processor [36], SSE2 extensions to the Pentium 4 [37], and MMX and SSE extensions to the Pentium III combined [37]. SSE is complementary to MMX, and MMX is always supported whenever SSE is, so they can be considered a single architecture. MMX and SSE are also available on Pentium 4 processors, but the SSE2 instruction set largely supersedes the previous ones, so I consider them different architectures. These architectures are very diverse, and thus present a good challenge to portability. Table 3.1 shows some characteristics of their instruction sets. They differ in their register lengths, partition sizes and types that they support, as well as in the instructions available for each partition type. The next four sub-sections discuss the each of these instruction sets in more depth.

Table 3.1
Characteristics of the instruction sets in the selected target architectures

Architecture	TM1300	MMX + SSE	SSE2	AltiVec
Register length	32 bits	64 bits	128 bits	128 bits
Integer partition types	8, 16 & 32 bits	8, 16, 32 & 64 bits	8, 16, 32 & 64 bits	8, 16 & 32 bits
Floating point partition types	32 bits	32 bits	32 & 64 bits	32 bits

3.2.1.1 AltiVec

AltiVec is the multimedia extension in Motorola PowerPC G4 processors. It is composed of a set of 128-bit registers that can be partitioned into 8, 16 and 32-bit integer partitions, and in 32-bit floating-point partitions. Most integer instructions are supported for all integer partition types, with a few exceptions. For example, `vec_madds` (`multiply-high`) and `vec_msum` (`multiply-add-pairs`) are supported only on 16-bit partitions.

AltiVec is programmed in an extended version of C that supports vector variables. All vectors are understood to be 128-bit long, so the type uniquely identifies the number of elements in the vector. For example:

```
vector char A;
vector int B;
```

means that *A* is a vector divided into 16 sections, each of which represents a signed 8-bit value, while *B* is a vector of 4 32-bit signed integer values. AltiVec also supports vector literals:

```
C = (vector char) (c)
D = (vector int) (c1, c2, c3, c4)
```

In this case, *C* results in a vector with all the elements equal to `c`, and *D* results in a vector whose four elements are equal to `c1, c2, c3` and `c4` respectively. Parallel operations are

executed using intrinsics. The intrinsics are overloaded for different vector types, so the following operations perform absolute value on partitions of different sizes:

```
vector char E, F;
vector int G, H;

F = vec_abs(E);
H = vec_abs(G);
```

Vectors can be loaded and stored in memory only at 16-byte aligned addresses. Unaligned accesses must be done through data rearrangement, using the permutation instruction. AltiVec's permutation instruction requires a vector of indices to define the permutation indices. Special instructions help set the permutation vector for data re-alignment:

```
perm_vector = vec_lvsl(0, pointer);
dst = vec_perm( vec_ld(0, ptr), vec_ld(0, ptr+1), perm_vector);
```

In this example, the intrinsic *`vec_lvsl`* creates a permutation vector from the unaligned address, which is later used to re-align the data using the permutation intrinsic *`vec_perm`*.

3.2.1.2 MMX + SSE

MMX is the first of a series of extensions to Pentium processors. MMX uses a set of 64-bit registers partitioned into 8, 16 and 32-bit integer sections. SSE (Streaming SIMD Extensions) is a set of instructions that are complementary to MMX. It adds some integer instructions on the same registers, and a new set of 128-bit registers partitioned into 32-bit floating-point sections. The MMX registers share resources with the scalar floating-point registers, so they cannot be used at the same time. A special *`EMMS`* instruction must be executed before and after using MMX and SSE integer instructions, unless no scalar floating-point operations can happen. The SSE floating-point registers do not contend with other resources.

There are two methods for programming this architecture. One is to use *_m64* and *_m128* types, which represent the integer and floating-point vector registers. The size and type of the partitions are determined by the operation intrinsics. For example:

```
_m64 A, B, C, D;

A = _mm_add_pi8(A, B)
C = _mm_add_pi16(C, D)
```

In this example *A* gets the addition of 8-bit sections, while *C* gets addition of 16-bit partitions. The other method of programming is to use C++ vector classes, which overload the standard C operators for vectors, and infer the type from the variable class. I use the first method of intrinsics in MMM declarations.

Many MMM and SSE instructions can take a memory location as a second argument, as in the following example:

```
_m64 A;
char *pB;

A = _mm_add_pi8(A, *pB);
```

There are no alignment restrictions for integer loads and stores in this architecture. Loads and stores for integer vectors are done by de-referencing pointers. Floating-point vector loads and stores do have different performance when the address is 16-byte aligned or not, so there are specific intrinsics to load and store floating-point vectors to aligned and unaligned addresses:

```
_m128 A;

A = _mm_load_ps(aligned_pointer);
A = _mm_loadu_ps(unaligned_pointer);
```

Memory accesses as second arguments to floating-point instructions are required to be 16-byte aligned.

3.2.1.3 SSE2

Pentium 4 processors, in addition to MMX and SSE, support the SSE2 instruction set. SSE2 reuses the 128-bit registers defined in SSE, but can now divide them into 8, 16, 32 and 64-bit integer partitions, or in 32 and 64-bit floating point sections.

This architecture is programmed very similarly to MMX and SSE, except that the new register types are `_m128i` for integer, and `_m128d` for double precision floating-point. Single-precision floating-point is supported the same way as in SSE, using the `_m128` type.

In SSE2 the integer vectors have the same alignment requirements as the floating-point vectors in SSE. Normal memory accesses are required to be at 16-byte aligned addresses. This is true for memory addresses as second arguments to operations. Unaligned loads and stores are supported through a special set of intrinsics:

```
_m128i A;
A = _mm_loadu_si128(pointer);
```

3.2.1.1 TriMedia TM1300

The TriMedia processor does not have a separate set of multimedia registers, but it does have several partitioned instructions that operate on the regular 32-bit registers. Vectors are declared as integer variables, and the operation intrinsics define the size and type of the partitions:

```
int A, B, C;
A = QUADAVG(B, C);
```

In this example, *A* gets the average of vectors *B* and *C* divided into 8-bit unsigned partitions. Vector loads and stores are restricted to be on 4-byte aligned addresses.

Unaligned loads must be emulated using two *load*s and *shift*s. Special *funnel-shift* instructions are provided to realign data:

```
int A;
A = FUNSHIFT1(*pA, *(pA+1));
```

This example loads two 32-bit words from the aligned address *pA*, and realigns them by concatenating the last 3 bytes of **pA* with the first byte of **(pA+1)*.

3.2.2 Definition of a Common Virtual Instruction Set

The next step was to define a virtual instruction set based on all of the selected targets. This virtual architecture is composed of vector registers as long as the longest target registers. In this case, it is 128-bit registers with 8, 16, and 32-bit integer partitions, and 32-bit floating point partitions. The virtual architecture can support shorter vectors (i.e. 64-bit vectors), but they map sub-optimally to 128-bit architectures, so their use is discouraged.

The virtual instruction set includes all operations that are common, or can be emulated efficiently on all the targets. Different MMM macros are defined for each combination of operations, input and output vector lengths and types. Other characteristics, like special handling of overflow, are also specified by each operation macro.

Virtual instructions can be defined in a way that the exact behavior under boundary conditions is undefined. For example, an addition operation may be defined to have unspecified behavior under overflow. This allows it to be mapped to target instructions that handle overflow differently (i.e. perform saturation, or modulo arithmetic), and thus provide for a common instruction that otherwise would not be available.

The virtual instruction set does not include operations that cannot be emulated efficiently on all targets. Therefore, there are instructions in some of the target architectures that are not available to MMM programs. An objective of this research is to determine how much performance is lost by not using these instructions. The virtual instruction set for the selected targets is discussed in Chapter 4, and the full definition appears in Appendix A.

3.2.3 Implementation of an MMM Library for each Target

Once a common virtual instruction was defined, it was possible to implement it for the different target architectures. I did not implement the full virtual instruction set, but only the part that was required by the selected example programs, described below. Appendix B shows the source code of the implementation of the MMM libraries for the four targets.

3.2.4 Example Program Selection

I selected the following examples to be implemented in MMM: 8x8 integer IDCT, 16x16 integer L_1-Distance, and 16x16 L_1-Distance with interpolation. These kernels are used by MPEG2, MPEG4, and H.263+ video compression applications, and represent a large portion of their computational load. The 8x8 IDCT is also used in JPEG still-image compression. Hand-optimized versions of some of these kernels are available from the selected target processor vendors, and take advantage of the complex parallel operations available.

The three examples are tested in the context of an MPEG2 video encoder. The MPEG Software Simulation Group test model 5 [42] is used with a sequence of 704x576 outdoor images as input. The IDCT example is a direct replacement for the idct() function in the MPEG2 model. The L_1-Distance examples replace portions of the dist1() function, corresponding to no interpolation, and both horizontal and vertical interpolation of 16x16 blocks. The dist1() function in the MPEG2 model also handles horizontal-only and vertical-

only interpolation, as well as 16x8 blocks, which are not of interest for this research. The MPEG2 model was modified to guarantee 16-byte alignment of the working image buffers, and to separate the dist1() function into various components, according to the block size and interpolation type.

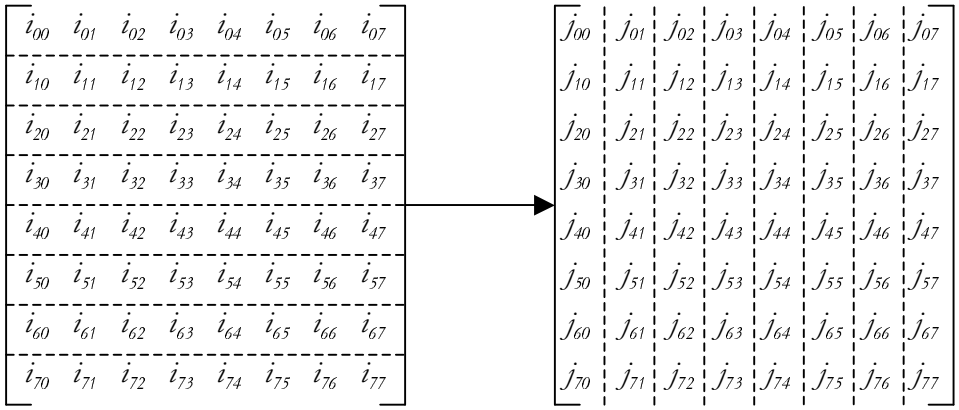
3.2.4.1 8x8 IDCT

The IDCT works on 8x8 blocks of 16-bit signed integers. The interface is a function call with two pointers to 16-bit integers, one for the input and one for the output, which can possibly overlap:

```
void Idct8x8 ( INT16 *pSrc, INT16 *pDst)
```

The input block is stored in a contiguous piece of memory in row-major format, so elements of each row are stored in adjacent locations in memory. The output is stored in the same format as the input. Each input element can have values between –300 and 300 inclusive. The function is to compute the two-dimensional IDCT of the input, and meet the accuracy requirements specified in the IEEE 1180-1990 standard [41].

Two-dimensional IDCTs are usually implemented using a separable approach: first a one-dimensional IDCT is applied to each row, and then an IDCT is applied to each column.



This reduces the problem to the computation of one-dimensional IDCTs of length 8 over rows and columns of an 8x8 block. The 8-element IDCT is defined as:

$$x_k = \sum_{n=0}^{7} \cos\frac{\pi n(2k+1)}{16} c_n y_n \tag{3.1}$$

where $c_0 = \frac{\sqrt{2}}{2}$ and $c_n = \frac{1}{2}$ for $n = 1,...,7$. This can be expressed in matrix form as:

$$C_8^{-1} = \frac{1}{2}\begin{bmatrix} c_4 & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 \\ c_4 & c_3 & c_6 & -c_7 & -c_4 & -c_1 & -c_2 & -c_5 \\ c_4 & c_5 & -c_6 & -c_1 & -c_4 & c_7 & c_2 & c_3 \\ c_4 & c_7 & -c_2 & -c_5 & c_4 & c_3 & -c_6 & -c_1 \\ c_4 & -c_7 & -c_2 & c_5 & c_4 & -c_3 & -c_6 & c_1 \\ c_4 & -c_5 & -c_6 & c_1 & -c_4 & -c_7 & c_2 & -c_3 \\ c_4 & -c_3 & c_6 & c_7 & -c_4 & c_1 & -c_2 & c_5 \\ c_4 & -c_1 & c_2 & -c_3 & c_4 & c_5 & c_6 & -c_7 \end{bmatrix} \tag{3.2}$$

where $c_k = \cos(\pi k/16)$. Borrowing the notation from [44], this matrix can be decomposed as

$$C_8^{-1} = \frac{1}{2} A_8^{-1} M_8^{-1} P_8^{-1} \tag{3.3}$$

where

$$A_8^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \end{bmatrix} \quad P_8^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M_8^{-1} = \begin{bmatrix} c_4 & c_2 & c_4 & c_6 & 0 & 0 & 0 & 0 \\ c_4 & c_6 & -c_4 & -c_2 & 0 & 0 & 0 & 0 \\ c_4 & -c_6 & -c_4 & c_2 & 0 & 0 & 0 & 0 \\ c_4 & -c_2 & c_4 & -c_6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & c_1 & c_3 & c_5 & c_7 \\ 0 & 0 & 0 & 0 & c_3 & -c_7 & -c_1 & -c_5 \\ 0 & 0 & 0 & 0 & c_5 & -c_1 & c_7 & c_3 \\ 0 & 0 & 0 & 0 & c_7 & -c_5 & c_3 & -c_1 \end{bmatrix}$$

This decomposition is the base of all fast IDCT algorithms. Most algorithms attempt to minimize the number of operations by further decomposing the operator M_8^{-1}. For example, a Chen IDCT [51] decomposes the even part (top left quadrant) of M_8^{-1} as:

$$M_{4E}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & -1 & 0 \\ 1 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} c_4 & c_4 & 0 & 0 \\ c_4 & -c_4 & 0 & 0 \\ 0 & 0 & c_6 & c_6 \\ 0 & 0 & c_2 & -c_2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.4}$$

And the odd part (bottom right quadrant) of M_8^{-1} is decomposed as:

$$M_{4O}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_7 & 0 & 0 & -c_1 \\ 0 & c_3 & -c_5 & 0 \\ 0 & c_5 & c_3 & 0 \\ c_1 & 0 & 0 & c_7 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -c_4 & c_4 & 0 \\ 0 & c_4 & c_4 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.5}$$

3.2.4.2 16x16 L_1-Distance

The L_1-Distance kernels are used as part of Motion Estimation algorithms. A 16x16 image block is compared against several possible locations of a reference image in search for the location with the minimal distance between the two blocks. The distance is computed as the sum of absolute differences of all the corresponding pixels in the blocks:

$$L_1 Dist = \sum_{i=0}^{15} \sum_{j=0}^{15} \left| x_{i,j} - y_{i,j} \right| \tag{3.6}$$

The basic function takes two pointers to 8-bit unsigned integers, one for the reference block, and one for the input block. `RowPitch` is an integer that represents the distance in memory between consecutive rows, for both blocks. A fourth input (`Limit`) specifies the minimal distance found by the motion estimation algorithm on other blocks, which is useful to exit the function early if a partial distance exceeds this limit; I refer to this as a shortcut path. The output is an integer representing the L_1 distance of the two blocks:

```
int L1Dist16x16(UINT8 *pRef, UINT8 *pIn, int RowPitch, int Limit)
```

Both input blocks are stored in row-major format, with a separation of `RowPitch` between consecutive rows. The input block is assumed to be aligned to a 16-byte boundary, but the reference block is not.

3.2.4.3 16x16 L_1-Distance with Interpolation

The third example is a variation of the 16x16 L_1-Distance that computes the half-pixel horizontal and vertical interpolation of the reference block before computing the distance. The parameters are the same as above:

```
int L1Dist16x16_InterpXY(UINT8 *pRef, UINT8 *pIn,
                         int RowPitch, int Limit)
```

The interpolation computes the rounded average of each pixel with the pixels to the right and/or below, according to the formula:

$$\bar{x}_{i,j} = \left\lfloor \frac{x_{i,j} + x_{i+1,j} + x_{i,j+1} + x_{i+1,j+1} + 2}{4} \right\rfloor \tag{3.7}$$

An average error of 0.5 in the results is allowed for the L_1-Distance function with interpolation. This error does not affect significantly the quality of the motion estimator, and allows for more efficient implementations of the interpolation function.

It is important to note that the performance of L_1-Distance implementations with a shortcut path depends on the motion estimation algorithm used, and on the input data itself. The execution speed depends on how often is the shortcut path taken. In order to separate the effects of the shortcut path on the execution speed, I created versions of both L_1-Distance examples with and without a shortcut path.

3.2.5 Analysis of Reference Implementations of Examples

In order to verify that portable MMM programs have good performance, they have to be compared with the best known implementations of the same programs. I will refer to these programs as reference implementations. Reference implementations are hand-optimized for each target platform, and represent the upper bound for the performance of the portable implementations. It is also interesting to compare the performance of portable and reference implementations with that of a scalar implementation of the same algorithm. This provides a measurement of speedup. One of the reasons for choosing IDCT and L_1-Distance as examples is that there are hand-optimized reference implementations available from the processor vendors.

3.2.5.1 8x8 IDCT

I found two versions of IDCT optimized for AltiVec [45, 50]. Both were implemented by Motorola, but the second one is distributed by Apple. Both implementations perform only vertical IDCTs, but transpose the matrix after each pass. The vertical IDCT is performed on the eight columns in parallel, since each 128-bit register can hold one element of every column. The algorithm used for the IDCT is a standard Chen algorithm [51] for the case of Apple's, and a modified scaled Chen algorithm for Motorola's. Each operation in the IDCT algorithm becomes a parallel vector operation on the whole rows. Parallel multiplications are done with

the *vec_mradds* instruction (equivalent to *multiply-high*, but with an extra factor of 2) and take advantage of the addition of a third argument whenever possible (equivalent to *multiply-high-add*). The coefficients are represented with 15 bits of fractional precision, which compensates for the factor of 2 introduced by the *vec_mradds* instruction. Matrix transposition is done by repeatedly applying the *vec_mergeh* and *vec_mergel* instructions. A total of 24 instructions are necessary to perform the transposition. Constants are broadcasted through the vectors by using *vec_splat*. Neither of these implementations meet the IEEE 1180 standard for accuracy of IDCTs [41].

An optimized implementation of IDCT for TriMedia is discussed as an optimization case study in their documentation [46]. Their implementation of both horizontal and vertical IDCTs is based on the parallel multiplication variant of Loeffler's IDCT algorithm [52]. The horizontal IDCT takes advantage of the *IFIR16* instruction (equivalent to *multiply-add-pairs*) to multiply by coefficients and add adjacent products. The results are then added and subtracted as 32-bit values until the final result is converted back to 16 bits. The horizontal IDCT is computed for two rows, and their results are packed into the high and low halves of the output registers. This helps avoid transposition in the vertical IDCT. The vertical IDCT can then use *IFIR16* instructions to do the multiplications in the first IDCT stage, and then do 32-bit additions and subtractions to complete the IDCT. This implementation is only efficient because TriMedia has relatively short registers. It does not scale well to longer register lengths. This design meets the requirements of the IEEE 1180 accuracy standard.

The reference IDCT implementations for MMX+SSE and SSE2 are from Intel [43, 44]. They both use basically the same algorithm. The vertical IDCT is done for all columns in parallel, like in the case of AltiVec. The algorithm uses a decomposition with minimal number of

multiplications, and avoids some by moving an operator into the last stage of the horizontal IDCT. Vector multiplications are done using the `PMULHW` instruction (`multiply-high`), and constants are represented with 16 bits of fractional precision. The horizontal IDCT uses the basic decomposition in equation (3.3). Operator P_8^{-1} is a vector permutation that is done using the `PSHUFW` instruction. Operator M_8^{-1} is computed using `PMADDWD` (`multiply-add-pairs`). The data is assigned to vectors in such a way that it is possible to use parallel additions and subtractions, rather than summing elements in the same vector. More on this is Chapter 5, as this design is the base of the portable MMM version. This IDCT meets the IEEE 1180 accuracy requirements.

For the scalar version of IDCT, I used the one from the MPEG2 model, but with one obvious optimization: inline the row and column IDCT sections instead of invoking two function calls. Also I removed a shortcut path intended to accelerate transforms of DC signals. I found that this shortcut at best provides a speedup of 10% on the chosen target architectures, and it is easier to measure the instruction counts when there is only one path. Such a shortcut is impractical for optimized IDCT implementations, because it requires a comparison of all vector elements with zero, and that is not directly supported by most multimedia instruction sets.

3.2.5.2 16x16 L_1-Distance

The 16x16 L_1-Distance was implemented for AltiVec by Motorola [49]. They provide different versions: one for when both blocks are aligned, and one for when one of the blocks is aligned and the other unaligned. Since the alignment of the reference block is not known, I use the unaligned version always. The 16x16 block is completely unrolled. It computes the re-alignment permutation vector using `vec_lvsl`, and uses it with `vec_perm` to re-align all input vectors prior to computing the absolute differences. For each row it uses the emulation `vec_sub(vec_max(a,b), vec_min(a,b))` to compute the absolute differences of corresponding vector elements. Then it uses `vec_sum4s` to obtain four partial sums. It does the same for all rows, and accumulates the partial sums using the second argument to `vec_sum4s`. At the end, the four partial sums are added using `vec_sums`, and converted to integer using `vec_splat` and `vec_ste`.

Reference optimized 16x16 L_1-Distance implementations are available for MMX+SSE in assembly and in C with intrinsics [47] and for SSE2 in C with instrinsics [48]. The SSE versions were capable of doing a full-search motion estimation over a region, so they have some outer loops that I don't use. I removed the loops from the C version, but the assembly one still has the overhead. For SSE there are no alignment considerations. It uses the `PSADBW` instruction to compute the sum of absolute differences of 8 elements, which is half a row. It uses `PADDW` to add the partial results. On SSE2 it uses aligned loads for the input vector, and unaligned loads for the reference. The `PSADBW` instructions can compute the sum of absolute differences of a whole row at a time, but results in two partial results. At the end, the two partial results are added using `shift-right` and `add`, prior to being converted into an integer value.

TriMedia discusses motion estimation in their documentation [35] for aligned loads only. It uses the *UME8UU* instruction to compute the sum of absolute differences of 4 elements. It uses a loop over rows, unrolled by a factor of eight. I did not use this example because it cannot deal with unaligned addresses. Instead I created my own optimized versions with different re-alignment strategies. They are discussed as target-specific optimizations in Chapter 5.

All these reference implementations are for L_1-Distance without interpolation, and without shortcut paths. There are no optimized reference designs for L_1-Distance with interpolation. The scalar implementations are derived from the dist1() function in the MPEG2 model.

3.2.6 Implementation of Portable Optimized Examples in MMM

I wrote portable-optimized implementations of the example programs in MMM, using the virtual instruction set defined previously for the group of target architectures. Selecting an algorithm that maps well to all the targets took some experimentation. I started with the algorithms used by some of the hand-optimized examples, and saw how well they performed on other targets. Since MMM can emulate long vectors efficiently on architectures with short registers, but not the other way around, I based the MMM programs on the hand-optimized versions for the targets with longest register lengths. In the case of my selected targets, I based them on the 128-bit implementations for SSE2 and AltiVec. Then I applied several modifications taken from the other reference examples, and some of my own, until I settled on a design that performs well on all targets.

The selected algorithms for the MMM example programs, and details about their implementation are discussed in Chapter 5. The full source of the example programs in MMM is included in Appendix C.

3.2.7 Performance Measurement

I measured the performance of the example programs on each target architecture. Two measurements of performance are of interest: the instruction count and the execution speed.

The instruction count was determined by adding the number of instructions in the assembly output of the examples for each target. When there were loops, the loop instruction count of the loop was multiplied by the number of loop iterations. In the cases of L_1-Distance with shortcut paths, the number of iterations depends on the input data. I did not measure instruction counts for these cases.

The execution speed of the example programs was measured in the context of an MPEG2 encoder model processing real input images. The MMM example programs replaced equivalent functions in the MPEG2 model, and loops were added around them. The loops iterate thousands of times through the same functions. This helps improve measurement accuracy, and minimize the effect of cache misses on the function calls. I timed the whole loops using high-precision timers when available, and computed the average time per call to each example function. On TriMedia I used a hardware cycle counter to time the programs. On Intel architectures I used multimedia timers, which are cycle-accurate. On AltiVec I used system timers, which are not as precise, but I compensated by increasing the loop repetitions.

The performance of MMM programs was compared to the hand-optimized versions for each target, when available. This indicates how much performance is lost in order to obtain

portability with MMM. The performance of the MMM examples was also compared with that of scalar implementations. This shows a measurement of the speedup that can be obtained by using only portable constructs.

Another interesting experiment was to compare the portable optimized version of an example program with a non-portable optimized version of the same algorithm. This tells how much performance is lost by using only portable virtual instructions, and not by the difference in algorithms. In order to do this, I attempted to further optimize the MMM examples by using non-portable instructions available on each target architecture. All the performance measurements are shown in Chapter 6.

3.3 Summary

In order to validate MMM as a solution to the problem of portability of optimized programs, I chose four distinct target architectures, studied their instruction sets, designed a common virtual instruction set, and implemented it as MMM libraries for all the targets. Then I selected three example multimedia programs, studied hand-optimized implementations of them for the different targets, selected a portable algorithm for each, and implemented them using MMM. I experimented with variations of the programs until I obtained a single version of each that performs well on all targets. I measured the performance as execution times and instruction counts. The next three chapters present the results of the steps described above. This includes the design of the common virtual instruction set, the design of the MMM examples, and the performance measurements.

Chapter 4

COMMON VIRTUAL INSTRUCTION SET

This chapter discusses the design of a common virtual instruction set. Through analysis of the instruction sets of the four target architectures, I produced a set of instructions that map efficiently to all targets. This instruction set is valid only for these four specific targets, but the approach is valid for any other set of architectures. The common set supports 128-bit vectors divided into 8, 16 and 32-bit integer, and into 32-bit floating-point partitions. Even though SSE2 supports 64-bit integer and floating-point partitions, they are not commonly used by multimedia programs. It is possible to emulate them on the other architectures if need arises, but will be left out of the initial common instruction set. Vectors shorter than 128 bits are not supported at this time because they map sub-optimally to architectures with 128-bit registers. For best performance it is important that the portable programs use vectors as long as the longest registers in the set of target architectures.

The virtual instruction set implements parallel operations that are supported or can be emulated easily on all the targets. The following sections give an overview of the common operations supported grouped by type, with emphasis on the strategies used to emulate instructions when required. Appendix A shows the complete common virtual instruction set, and the mapping of each macro into intrinsics for each target.

4.1 Vector Declarations

MMM macros provide a unique interface to vector declarations in different architectures. The definition of the macros for each target follows the particular style of each. On AltiVec it uses the *vector* attribute of basic types:

```
#define DECLARE_I16x8(var) \
    vector INT16 var;
```

where *INT16* is defined as a short signed integer. On SSE2, vectors are declared using the *_m128* types:

```
#define DECLARE_I16x8(var) \
    __m128i var;
```

For MMX and SSE, integer vectors are 64-bit long, so two are necessary to simulate a 128-bit vector:

```
#define DECLARE_I16x8(var) \
    __m64 var##_0;         \
    __m64 var##_1;
```

In the case of TriMedia, four 32-bit variables are needed to declare a 128-bit vector:

```
#define DECLARE_I16x8(var) \
    int var##_0; \
    int var##_1; \
    int var##_2; \
    int var##_3;
```

A different MMM declaration macro is required for every partition type. Table 4.1 shows all the MMM vector declaration macros supported.

Table 4.1
MMM vector declarations

Partition Type	MMM Macro Name
8-bit signed integer	`DECLARE_I8x16`
8-bit unsigned integer	`DECLARE_U8x16`
16-bit signed integer	`DECLARE_I16x8`
16-bit unsigned integer	`DECLARE_U16x8`
32-bit signed integer	`DECLARE_I32x4`
32-bit unsigned integer	`DECLARE_U32x4`
32-bit floating-point	`DECLARE_F32x4`

Another important function is to declare constant arrays of vectors. These are really memory buffers that are statically initialized, but with guaranteed alignment. Constant arrays of vectors are useful to declare large sets of constants that can later be loaded into vectors when needed. Declaration of arrays of vectors is done slightly different on each architecture. For example, a macro to declare an array of 4 32x4 vectors on AltiVec is:

```
#define DECLARE_CONST_I32x4x4(var, c11, c12, c13, c14,          \
                                   c21, c22, c23, c24,          \
                                   c31, c32, c33, c34,          \
                                   c41, c42, c43, c44)          \
    vector INT32 var[4] = {(vector INT32) (c11, c12, c13, c14), \
                           (vector INT32) (c21, c22, c23, c24), \
                           (vector INT32) (c31, c32, c33, c34), \
                           (vector INT32) (c41, c42, c43, c44)};
```

The parameter *var* is the name of the array. Note that the assignment uses vector literals. Using the vector qualifier in the declaration guarantees 16-byte alignment. Individual constant vectors can be accessed by indexing into the array:

```
DECLARE_CONST_I32x4x2(A, 1, 1, 1, 1, 2, 2, 2, 2);
DECLARE_I32x4 B;

B = A[1];
```

The first line in this example declares a constant array of vectors called *A*, and initializes it to constant values. Then the vector *B* loads the second vector of the array *A*, which was set to [2 2 2 2]. Constant arrays of vectors on MMX, SSE and SSE2 are declared as a two-dimensional array of scalars. Using the qualifier `__declspec(align(16))` guarantees the alignment:

```
#define DECLARE_CONST_I32x4x4(var, c11, c12, c13, c14           \
                                   c21, c22, c23, c24           \
                                   c31, c32, c33, c34           \
                                   c41, c42, c43, c44)          \
    __declspec(align(16)) INT32 var[4][4] = {c11, c12, c13, c14, \
                                             c21, c22, c23, c24, \
                                             c31, c32, c33, c34, \
                                             c41, c42, c43, c44};
```

On TriMedia there is no construct to force 16-byte alignment of static variables. However, there is also no requirement that constant vectors are 16-byte aligned, only that they are 4-byte aligned, which is the default alignment of all static variables.

MMM needs to know the size of the constant array at compile time, and it is impractical to have different macros for all possible array sizes. For this reason, MMM implements only the sizes required by the example programs. A more general solution is discussed in Chapter 7 as future work.

4.2 Set Instructions

These instructions allow programs to set the values of vector elements to specified values. They are implemented in AltiVec using vector literals:

```
#define SET_I16x8(dst, c1, c2, c3, c4, c5, c6, c7, c8) \
    dst = (vector INT16) (c1, c2, c3, c4, c5, c6, c7, c8);
```

On MMX, SSE and SSE2 there are intrinsics for this purpose. For example, in SSE2:

```
#define SET_I16x8(var, c1, c2, c3, c4, c5, c6, c7, c8) \
    var = _mm_set_epi16(c1, c2, c3, c4, c5, c6, c7, c8);
```

On TriMedia it uses assignment for 32-bit elements, and packing with *shift*s and *or*s for smaller partitions:

```
#define SET_I16x8(dst, c1, c2, c3, c4, c5, c6, c7, c8) \
    dst##_0 = (c2 << 16) | c1; \
    dst##_1 = (c4 << 16) | c3; \
    dst##_2 = (c6 << 16) | c5; \
    dst##_3 = (c8 << 16) | c7;
```

If the *SET* macro is used with constant arguments, then these *shift*s and *or*s are eliminated through constant propagation by the vendor compiler. A special case is when all elements are to be set to the same value. In this case the implementation is faster by using assignment:

```
#define SET1_I16x8(var, c) \
    var##_0 = var##_1 = var##_2 = var##_3 = (c << 16) | c;
```

There are intrinsics that can set all elements of a vector to the same value on MMX, SSE and SSE2. On AltiVec it must be done by using vector literals. Another special case is when all elements are to be set to zero, which is implemented more efficiently in MMX, SSE and SSE2 by using *xor*.

Another operation that falls in this group is vector copy. Since vectors may be represented by several variables, a macro is required to copy one vector to another. Table 4.2 summarizes the supported set macros.

Table 4.2
MMM set instructions

MMM Macro	Description	I8x16	U8x16	I16x8	U16x8	I32x4	U32x4	F32x4
SET	Set each element	√	√	√	√	√	√	√
SET1	Set all elements to the same value	√	√	√	√	√	√	√
CLEAR	Set all elements to zero	√	√	√	√	√	√	√
COPY	Copy one vector to another	√	√	√	√	√	√	√

4.3 Load and Store Instructions

There are separate MMM macros for loading and storing vectors to 16-byte aligned and unaligned addresses. Aligned loads and stores are done as straightforward pointer dereferences, or through load/store intrinsics. Table 4.3 shows the definition of an *aligned-load* macro on all targets:

Table 4.3
Implementations of vector load macro on all targets

AltiVec	`#define LOAD_A_I16x8(var, ptr) \` `  var = vec_ld(0, (vector INT16 *) (ptr));`
SSE2	`#define LOAD_A_I16x8(var, ptr) \` `  var = _mm_load_si128((__m128i *) (ptr));`

MMX+SSE	`#define LOAD_A_I16x8(var, ptr) \`
	`  var##_0 = *((__m64 *) (ptr)); \`
	`  var##_1 = *(((__m64 *) (ptr))+1);`

TriMedia	`#define LOAD_A_I16x8(var, ptr) \`
	`  var##_0 = *((int *) (ptr)); \`
	`  var##_1 = *(((int *) (ptr))+1); \`
	`  var##_2 = *(((int *) (ptr))+2); \`
	`  var##_3 = *(((int *) (ptr))+3);`

Multiple loads are required in TriMedia and MMX+SSE, because the registers are smaller than 128 bits. Even though loading and storing is independent of the partition size, different macros are required for each vector type because of type checking requirements in AltiVec. Also, floating-point vector loads/stores use different intrinsics in SSE and SSE2.

Unaligned loads/stores for integer vectors are the same as aligned ones in MMX+SSE, but different for floating-point vectors. There are different intrinsics for aligned and unaligned loads/stores for floating-point vectors in SSE, and for both integer and floating-point vectors in SSE2. In AltiVec it is necessary to perform two aligned loads, and re-align the data using permutation. The permutation vector must be computed separately, but can be reused for re-aligning multiple input vectors (i.e. multiple rows of a matrix that have the same alignment). This is supported in MMM by using a separate macro to prepare the alignment:

```
#define PREPARE_LOAD_ALIGNMENT(index, ptr) \
    mmm_align_vector##index = vec_lvsl(0, ptr);
```

The permutation vector is stored statically, and can be used later by multiple unaligned loads:

```
#define LOAD_U_U8x16(var, ptr, index)                        \
    var = vec_perm(vec_ld(0, (vector UINT8 *) (ptr)),        \
                   vec_ld(0, ((vector UINT8 *) (ptr)) + 1), \
                   mmm_align_vector##index);
```

There are multiple instances of the permutation vector, and are selected by passing in the index to both the *prepare* and the *load* macros. This concept of alignment preparation is also useful in TriMedia, where unaligned loads also need to be emulated. In the case of TriMedia, the re-alignment is done through *shift*s, so the preparation macro records the shift amounts that will be required later by the unaligned loads:

```
#define PREPARE_LOAD_ALIGNMENT(index, ptr)                   \
    mmm_shift_right_##index = (((int) (ptr)) & 0x3)<<3;      \
    mmm_shift_left_##index = 32 - mmm_shift_right_##index;
```

Note that in TriMedia the alignment requirement is 4 bytes, and not 16 like in AltiVec and SSE2. But in order to homogenize the interface, the alignment requirement is kept at 16 bytes for all MMM programs. The unaligned loads use the prepared shift amounts to re-align the data:

```
#define LOAD_U_U8x16(var, ptr, index)                                  \
    var##_0 = (*(((UINT8 *) ptr)+1) << mmm_shift_left_##index) |  \
              (*( (UINT8 *) ptr   ) >> mmm_shift_right_##index);  \
    var##_1 = (*(((UINT8 *) ptr)+2) << mmm_shift_left_##index) |  \
              (*(((UINT8 *) ptr)+1) >> mmm_shift_right_##index);  \
    var##_2 = (*(((UINT8 *) ptr)+3) << mmm_shift_left_##index) |  \
              (*(((UINT8 *) ptr)+2) >> mmm_shift_right_##index);  \
    var##_3 = (*(((UINT8 *) ptr)+4) << mmm_shift_left_##index) |  \
              (*(((UINT8 *) ptr)+3) >> mmm_shift_right_##index);
```

I have also defined a special *load-adjacent* macro that loads two overlapping vectors with one byte offset between them. This is used in one of the example programs to interpolate adjacent vectors. This operation is implemented very efficiently by applying two sets of re-alignments to the same inputs. For example, in AltiVec it is done by using two permutations on the same inputs:

```
#define LOAD_ADJ_U8x16(var1, var2, ptr, index1, index2) \
    var1 = vec_perm(vec_ld(0, (vector UINT8 *) (ptr)),       \
                    vec_ld(0, ((vector UINT8 *) (ptr)) + 1), \
                    mmm_align_vector##index1);               \
    var2 = vec_perm(vec_ld(0, (vector UINT8 *) (ptr)),       \
                    vec_ld(0, ((vector UINT8 *) (ptr)) + 1), \
                    mmm_align_vector##index2);
```

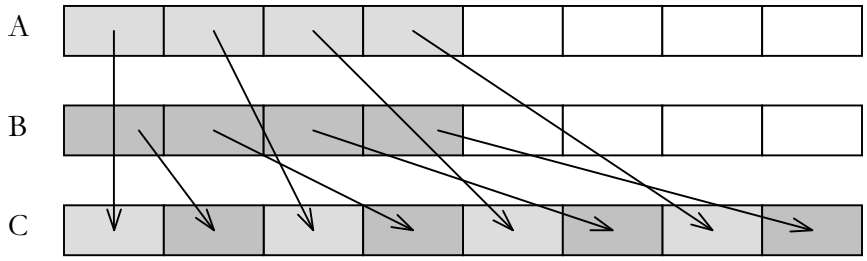
The compiler keeps the loaded inputs in registers and reuses them in the second permutation. Another useful operation is a *masked-store*. This operation uses a vector as a mask to stored elements. There are intrinsics that directly support this for 8-bit partitions in MMX, SSE and SSE2, and can be emulated easily on the other platforms by doing a *load*, a bit-wise *select* and a *store*. All the different load and store instructions are shown in Table 4.4.

Table 4.4
MMM load and store instructions

MMM Macro	Description	I8x16	U8x16	I16x8	U16x8	I32x4	U32x4	F32x4
LOAD_A	Load aligned	√	√	√	√	√	√	√
STORE_A	Store aligned	√	√	√	√	√	√	√
PREPARE_LOAD_ALIGNMENT	Prepare unaligned load	√	√	√	√	√	√	√
PREPARE_STORE_ALIGNMENT	Prepare unaligned store	√	√	√	√	√	√	√
LOAD_U	Load unaligned	√	√	√	√	√	√	√
STORE_U	Store unaligned	√	√	√	√	√	√	√
LOAD_ADJ	Load two adjacent vectors	√	√	√	√	√	√	√
STORE_MASKED	Conditional store	√	√					

4.4 Rearrangement Instructions

There are a number of instructions in the target instruction sets to deal with data rearrangement within vectors, or to combine data from two vectors. One operation that is well supported in all the targets is interleaving partitions of two vectors. Since interleaving the whole input vectors would result in a vector twice as long, the instructions actually operate on one half of each input vector. For example, an *interleave-high* operation on 16-bit partitions combines the top four partitions of each input vector to compose an 8-element result vector:



Similarly, there is an *interleave-low* instruction. Another useful rearrangement operation is a *broadcast*, where the value of a particular element is copied to all the vector elements. *Broadcast* can be implemented in SSE and SSE2 with permutation intrinsics. AltiVec has an instruction specifically for this purpose: *vec_splat*. TriMedia can easily do broadcasts on 32-bit partitions using assignment, because each 32-bit section of the vector is represented by a different variable, but on smaller partitions it has to emulate it. For example, to broadcast the second 16-bit element in a 16x8 vector, it has to replicate it once into a 32-bit variable, and then copy it into the other 32-bit variables:

```
#define BROADCAST_2_I16x8(dst, src) \
    dst##_0 = PACK16MSB(src##_1, src##_1); \
    dst##_3 = dst##2 = dst##_1 = dst##_0;
```

It is also possible to broadcast pairs of elements. This is equivalent to broadcasting elements of twice the size, but different macros are required to satisfy the type checking requirements in AltiVec.

Other permutations are possible, but not a general permutation. AltiVec does have general-purpose permutation instructions, controlled by a permutation vector. SSE and SSE2 have permutation intrinsics with the indices passed as immediate values, but with some restrictions: they can operate on 16 and 32-bit partitions, but not on 8-bit ones. Also, on SSE2, permutation of 16-bit partitions is restricted to one half of the vector, i.e. the destination of each element must be in the same half of the vector as the source. With these restrictions it is not possible to have a general-purpose permutation operation in MMM. A pre-compiler might be able to emulate arbitrary permutations, but that is outside the scope of MMM; such a system is proposed as future work in Chapter 7. For this research, I implemented only specific

permutations that are required by the example programs. The list of supported rearrangement instructions is shown below in Table 4.5.

Table 4.5
MMM rearrangement instructions

MMM Macro	Description	I8x16	U8x16	I16x8	U16x8	I32x4	U32x4	F32x4
INTERLEAVE_H	Interleave high halves	√	√	√	√	√	√	√
INTERLEAVE_L	Interleave low halves	√	√	√	√	√	√	√
BROADCAST_x	Broadcast x^{th} element	√	√	√	√	√	√	√
BROADCAST_PAIR_x	Broadcast x^{th} pair of elements	√	√	√	√	√	√	√
PERMUTE_02134657	Specific permutation			√	√			
PERMUTE_01237654	Specific permutation			√	√			

4.5 Conversion Instructions

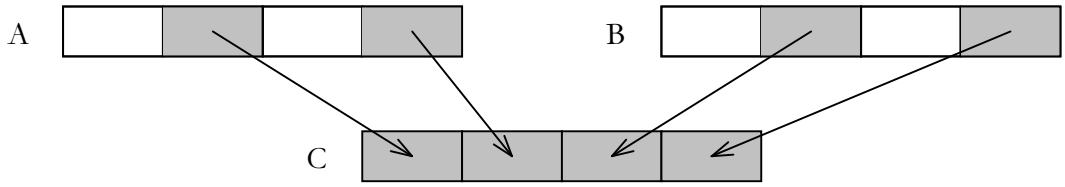
This section groups several instructions that perform type conversions. There are instructions to convert between vector and scalar variables, between integer and floating-point, and to reduce or expand the precision of the partitions.

Vectors in TriMedia are represented by scalar variables, so the conversion between them is trivial. This is not so in the other architectures, where vectors are held in different registers as scalars, so special intrinsics have to be used to convert between them. Scalar to vector conversions store the value of the scalar variable into the lowest element of the vector. The opposite happens in vector-to-scalar conversions. The conversion process is especially complicated in AltiVec; one must store a single element of the vector into the address of the scalar variable, but the address of the scalar variable is required to match the alignment of the vector element. Since the alignment of the scalar variable is not known, the solution is to

broadcast the lowest element to the whole vector, and then store whichever element matches the alignment:

```
#define CVT_U32_U32x4(dst, src) \
    vec_ste(vec_splat(src, 3), 0, &dst);
```

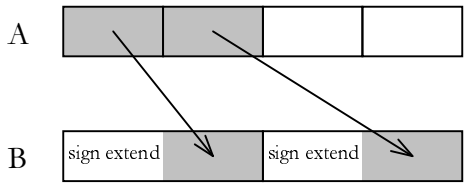
Pack instructions are used to reduce the precision of the vector elements. This instruction combines two input vectors into one. The lower half of each element in the first vector is packed together with the lower half of each element in the second vector, to produce a vector with twice as many partitions as each input vector:



This operation can be done with truncation, or with saturation. A *truncating-pack* ignores the top half of each element. A *saturating-pack* clips the full-precision value before packing. SSE, SSE2 and AltiVec support *pack* instructions only with saturation, and TriMedia supports them only with truncation. One can emulate truncation by masking-out the upper half of each element prior to running a *saturating-pack*. On TriMedia one can emulate a *saturating-pack* by clipping the inputs prior to the *truncating-pack*. If the inputs are known not to exceed the lower half, one could use either type of pack. For these cases, MMM defines pack instructions with unspecified reduction type.

The opposite operation is *extend*, where partitions are extended to partitions with twice the precision. Signed partitions are extended with sign-extension, while unsigned partitions are padded with zeros. *Extend* operations operate on one half of the input vector. For example,

an *extend-high* operation on a 16x4 vector extends the first two elements of the input into 32-bit partitions of a 32x2 vector:



The *extend-low* instructions do the same but on the last half of the input vector. Table 4.6 lists all the conversion operations supported in MMM.

Table 4.6
MMM conversion instructions

MMM Macro	Description	I8x16	U8x16	I16x8	U16x8	I32x4	U32x4	F32x4
CVT_vector_scalar	Convert scalar to vector					√	√	
CVT_scalar_vector	Convert vector to scalar					√	√	
CVT_float_int	Convert integer to floating-point					√		
CVT_int_float	Convert floating-point to integer							√
PACK_T	Pack with truncation			√	√	√	√	
PACK_S	Pack with saturation			√	√	√		
PACK_N	Pack with unspecified reduction			√	√	√	√	
EXTEND_H	Extend high half of vector	√	√	√	√			
EXTEND_L	Extend low half of vector	√	√	√	√			

4.6 Bit-wise Logic Instructions

Bit-wise operations are independent of the partition type and size. But in SSE and SSE2 there are different intrinsics for bit-wise operations on integer than on floating-point vectors, so two different macros are required. For consistency, macros are defined for each vector type. The basic logic operations *and*, *or* and *xor* are supported, as well as *andn* (and-not). A bit-wise conditional *select* operation uses the bits in one vector to select between the corresponding bits of two other input vectors. AltiVec has an intrinsic operation for *select*. In the other architectures it is emulated using *and*, *or* and *andn*. For example, on SSE2 it is implemented as:

```
#define SEL_I8x16(dst, src1, src2, mask)          \
    dst = _mm_or_si128(_mm_and_si128(src1, mask),  \
                       _mm_andnot_si128(src2, mask));
```

Table 4.7 shows all the bit-wise logic operations supported in the common virtual instruction set.

Table 4.7
MMM bit-wise logic instructions

MMM Macro	Description	I8x16	U8x16	I16x8	U16x8	I32x4	U32x4	F32x4
AND	Logical and	√	√	√	√	√	√	√
ANDN	And-not	√	√	√	√	√	√	√
OR	Logical or	√	√	√	√	√	√	√
XOR	Logical xor	√	√	√	√	√	√	√
SEL	Bit-wise select	√	√	√	√	√	√	√

4.7 Shift Instructions

Partitioned shift instructions operate on each section of a vector, without carrying over to the adjacent ones. The usual types of shifts are supported: *shift-left* (*SLL*), *shift-right-logical* (*SRL*), *shift-right-arithmetic* (*SRA*), plus a *rotate-left* instruction (*ROL*). The shift amounts can be immediate values or run-time variables; different MMM macros exist for these two cases. Not all parallel shift operations are supported for all partition types on the target processors. The only partitioned shift instruction supported by TriMedia is *shift-right-arithmetic* on 16-bit partitions. Only AltiVec supports parallel *shift* on 8-bit partitions. One can emulate shifts on smaller partitions by masking the run-over bits. For example, to emulate a *shift-left* instruction on 8-bit partitions in TriMedia, one can compute result for the even and odd partitions separately using masks and 32-bit *shift*s, and then combine the results:

```
#define SLL_I16x8(dst, src, amount)                          \
    dst##_0 = ((src##_0 << amount) & 0x00FF00FF) |           \
        (((src##_0 & 0xFF00FF00) << amount) & 0xFF00FF00);
```

The emulation could be done simpler if one could prepare a mask for just the run-over bits. Unfortunately it is not easy to generate the mask for arbitrary shift amounts. For shifts with immediate amounts, a pre-compiler would be able to generate the appropriate masks and do a better emulation. I propose this solution as future work in Chapter 7. Table 4.8 shows all the shift macros supported in MMM. Shift instructions do not apply to floating-point vectors.

Table 4.8
MMM shift instructions

MMM Macro	Description	I8x16	U8x16	I16x8	U16x8	I32x4	U32x4	F32x4
SLL	Shift-left logical	√	√	√	√	√	√	
SLL_I	Shift-left logical immediate	√	√	√	√	√	√	
SRL	Shift-right logical	√	√	√	√	√	√	
SRL_I	Shift-right logical immediate	√	√	√	√	√	√	
SRA	Shift-right arithmetic	√	√	√	√	√	√	
SRA_I	Shift-right arithmetic immediate	√	√	√	√	√	√	
ROL	Rotate left	√	√	√	√	√	√	
ROL_I	Rotate left immediate	√	√	√	√	√	√	

4.8 Floating-Point Arithmetic Instructions

Parallel floating-point arithmetic instructions are supported by all the target architectures. The usual basic arithmetic: *add*, *subtract*, *multiply*, and *divide* are supported directly or through simple emulations. Other instructions supported are *multiply-add*, *minimum*, *maximum*, *reciprocal* (1/x), and *square-root*. The list of supported floating-point instructions is shown in Table 4.9.

Table 4.9
MMM floating-point arithmetic instructions

MMM Macro	Description	F32x4
`ADD`	Add	√
`SUB`	Subtract	√
`MULT`	Multiply	√
`MULT_ADD`	Multiply-add	√
`DIV`	Divide	√
`MIN`	Minimum	√
`MAX`	Maximum	√
`SQRT`	Square root	√
`REC`	Reciprocal	√
`RSQRT`	Reciprocal of square root	√

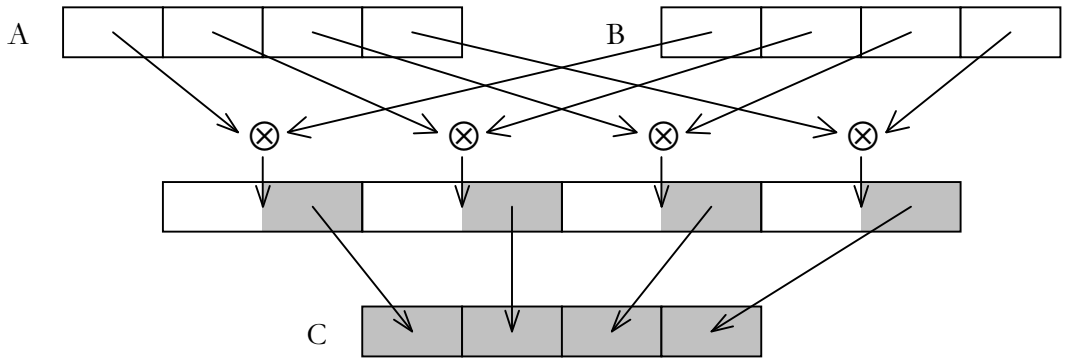
4.9 Integer Arithmetic Instructions

Parallel integer arithmetic is more complex than floating-point because of precision and overflow issues. Integer additions and subtractions can overflow. The standard way of handling overflow is to use modulo arithmetic, which basically ignores the carry bit. Many instruction sets also support saturation handling of overflow, where the result under overflow is the largest number representable by the precision. Not all targets support both types of overflow handling on all vector types. For example, TriMedia supports 16-bit parallel addition with saturation directly, but not with modulo. In order to emulate it, one can do a 32-bit addition, but prevent an overflow from bit 15 to bit 16. This is done by masking out bit 15 from both operands, which prevents any overflow, doing the 32-bit addition, and then adding the masked bits again using `xor`:

```
#define ADD_M_i16x8(dst, src1, src2)                                    \
    dst##_0 = (src1##_0 & 0xFFFFEFFF) + (src2##_0 & 0xFFFFEFFF) \
            ^ (src1##_0 & 0x00008000) ^ (src2##_0 & 0x00008000);\
    dst##_1 = (src1##_1 & 0xFFFFEFFF) + (src2##_1 & 0xFFFFEFFF) \
            ^ (src1##_1 & 0x00008000) ^ (src2##_1 & 0x00008000);\
    dst##_2 = (src1##_2 & 0xFFFFEFFF) + (src2##_2 & 0xFFFFEFFF) \
            ^ (src1##_2 & 0x00008000) ^ (src2##_2 & 0x00008000);\
    dst##_3 = (src1##_3 & 0xFFFFEFFF) + (src2##_3 & 0xFFFFEFFF) \
            ^ (src1##_3 & 0x00008000) ^ (src2##_3 & 0x00008000);
```

It is possible that an application does not care about the overflow handling, because the range of the data cannot result in overflow. For cases like these, I created a set of instructions with no specified overflow handling. These instructions are mapped to either modulo or saturation instructions, or into instructions on larger partitions (if no overflow is possible, then a 32-bit addition is equivalent to partitioned 8 or 16-bit additions).

Full-precision integer products require twice as many bits as the operands, so they don't fit in the same type of vectors as the inputs. Several variants of parallel multiplication deal with this in different ways. One way is to discard the most-significant half of the product, and keep the lower half. I call this *multiply-low*:



Similarly, a *multiply-high* stores the most-significant half of the product, and discards the lower half. MMM supports *multiply-high* and *multiply-low* for 16-bit integer partitions only. TriMedia does not support them directly, but can emulate them using 32-bit *multiply-*

high and *multiply-low*. For example, to emulate the *multiply-high* operation on the left half of a 32-bit register, one can mask the lower 16 bits and use a 32-bit *multiply-high*:

```
dst_high = IMULM(src1##_0 & 0xFFFF0000, src2##_0 & 0xFFFF0000);
```

The right half of the result can be computed with *multiply-low* (the standard * operator). The lower 16 bits of the operands are sign-extended to 32-bits using the *SEX16* intrinsic:

```
dsl_low = SEX16(src1##_0) * SEX16(src2##_0);
```

The two partial results are combined using the *PACK16MSB* instrinsic to form the packed *multiply-high* result of the 32-bit vector. The same process is repeated for each 32-bit section in the 128-bit vectors:

```
#define MULT_H_I16x8(dst, src1, src2) \
    dst##_0 = PACK16MSB(IMULM(src1##_0 & 0xFFFF0000,  \
                              src2##_0 & 0xFFFF0000), \
                        SEX16(src1##_0) * SEX16(src2##_0)); \
    dst##_1 = PACK16MSB(IMULM(src1##_1 & 0xFFFF0000,  \
                              src2##_1 & 0xFFFF0000), \
                        SEX16(src1##_1) * SEX16(src2##_1)); \
    dst##_2 = PACK16MSB(IMULM(src1##_2 & 0xFFFF0000,  \
                              src2##_2 & 0xFFFF0000), \
                        SEX16(src1##_2) * SEX16(src2##_2)); \
    dst##_3 = PACK16MSB(IMULM(src1##_3 & 0xFFFF0000,  \
                              src2##_3 & 0xFFFF0000), \
                        SEX16(src1##_3) * SEX16(src2##_3));
```

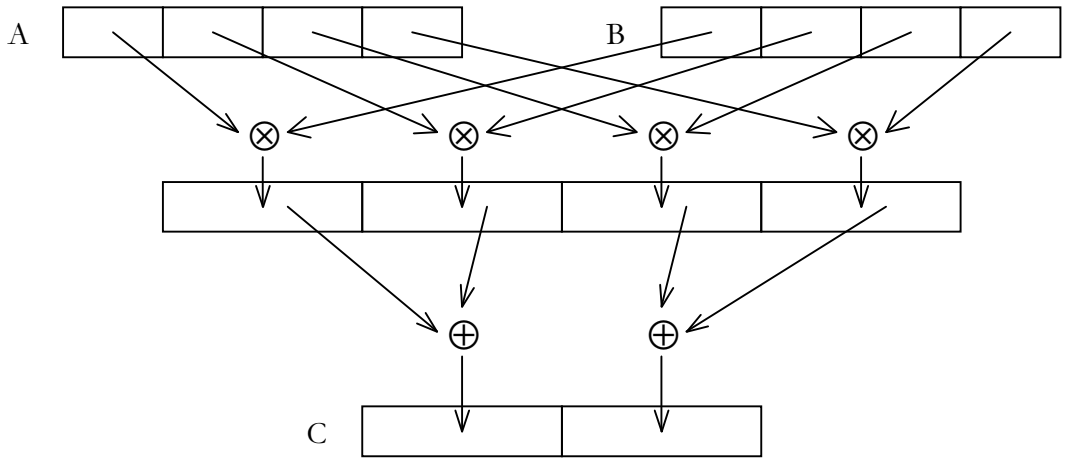
AltiVec supports a variant of *multiply-high* with the intrinsic *vec_madds*. This instruction extracts the most-significant 17 bits of the 32-bit products, adds corresponding 16-bit elements of a third input vector, and saturates the sum into a 16-bit result. Using *vec_madds* with a zero third input is almost equivalent to *multiply-high*, except that the results are bits 15-30 of the 32-bit product, instead of bits 16-31. One way to emulate *multiply-high* is to shift one of the operands to the right by one bit prior to the multiplication:

```
#define MULT_H_I16x8(dst, src1, src2)                              \
    dst = vec_madds(src1, vec_sra(src2, (vector UINT16) (1)),  \
                    (vector INT16) (0));
```

This emulation loses one bit of precision in one of the operands, but it is still useful to many applications. This instruction is used in the implementation of the IDCT example, with satisfactory results. The fact that AltiVec's *vec_madds* instruction can add a third input vector opens room for a combined MMM macro *multiply-high-add*. This can be easily emulated on the other targets by a separate vector addition following the *multiply-high* operation. For example in SSE2, a macro for *multiply-high-add* with saturation is defined as:

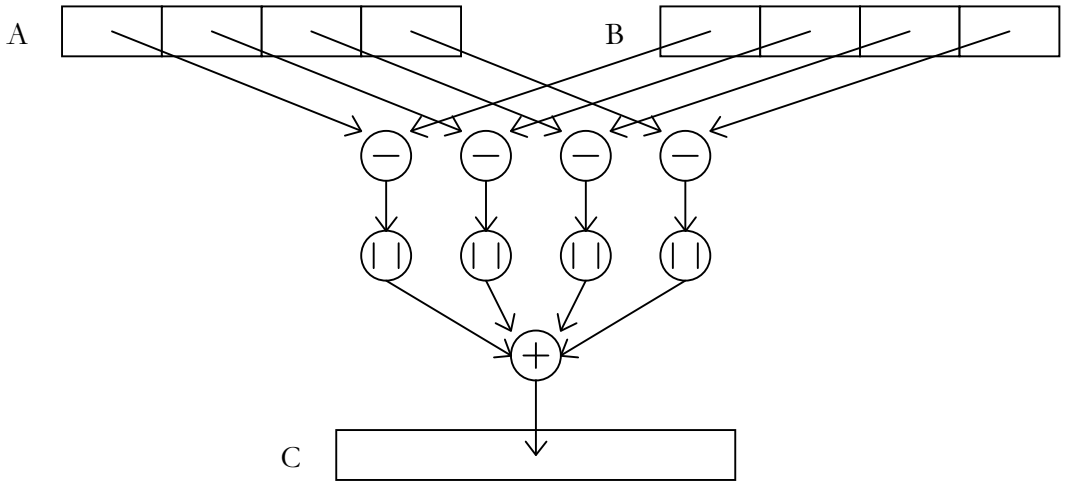
```
#define MULT_H_ADD_S_I16x8(dst, src1, src2, src3) \
    dst = _mm_add_epi16(_mm_mulhi_epi16(src1, src2), src3);
```

Another useful way to do integer multiplications is to add two products together. The *multiply-add-pairs* instruction performs 16-bit multiplication and adds the 32-bit products of adjacent partitions. The results are 32-bit values:



The *multiply-add-pairs* instruction on 16-bit integer partitions is supported directly by all the target instruction sets. In addition, AltiVec supports this operation with an additional parallel addition to the product result, with saturation or modulo handling of overflow. Since the addition of the third vector can be easily emulated in the other targets, it is included in the common instruction set.

One special operation available on several targets is *sad* (sum of absolute differences). This instruction computes the absolute value of the differences of corresponding vector elements, and adds the results together:



Sad is only supported on 8-bit unsigned partitions. None of the target architectures can do a full *sad* on 128-bit vectors, but they do provide pieces that help implement it. SSE2 has a *sad* instruction on 128-bit vectors, but returns two partial sums: the sum of the lower 8 sections is returned in bits 0-15 of the result vector, and the sum of the upper 8 sections on bits 64-79. MMM supports *sad* in the same way as SSE2, with two partial results:

```
#define SAD2_U8x16(dst, src1, src2) \
    dst = _mm_sad_epu8(src1, src2);
```

SSE operates on 64-bit registers, so it has to use two instructions to emulate the operation on 128-bit vectors. The two instructions result naturally in two partial results:

```
#define SAD2_U8x16(dst, src1, src2) \
    dst##_0 = _m_psadbw(src1##_0, src2##_0); \
    dst##_1 = _m_psadbw(src1##_1, src2##_1);
```

TriMedia supports *sad* on 32-bit registers with the *UME8UU* intrinsic. For a 128-bit vector, a total of four partial results result from the four *UME8UU* instructions. One can emulate the desired two partial sums by adding pairs of partial results:

```
#define SAD2_U8x16(dst, src1, src2)          \
    dst##_0 = UME8UU(src1##_0, src2##_0) +   \
              UME8UU(src1##_1, src2##_1);    \
    dst##_2 = UME8UU(src1##_2, src2##_2) +   \
              UME8UU(src1##_3, src2##_3);
```

AltiVec does not have a *sad* instruction, but one can emulate it by using parallel *maximum*, *minimum* and *subtract*: |a-b| = max (a, b) – min (a, b). Then the results need to be summed into two partial results using the intrinsics *vec_sum4s* and *vec_sum2s*:

```
#define SAD2_U8x16(dst, src1, src2)                              \
    dst = (vector UINT32) vec_sum2s(vec_sum4s(                   \
          vec_sub(vec_max(src1, src2), vec_min(src1, src2)),     \
          (vector UINT32)(0)), (vector INT32) (0));
```

The *vec_sum2s* intrinsic in AltiVec can add a third vector. This can be useful to accumulate partial results of multiple vectors. A *sad-add* instruction is supported in MMM for this purpose. A separate macro *SUM2_32x4* is used to sum the two partial results into a single scalar value.

Other parallel arithmetic operations supported are *average*, *minimum* and *maximum* of 8 and 16-bit partitions. Table 4.10 shows all the integer arithmetic instructions supported by the virtual instruction set.

Table 4.10
MMM integer arithmetic instructions

MMM Macro	Description	I8x16	U8x16	I16x8	U16x8	I32x4	U32x4
ADD_M	Add with modulo	√	√	√	√	√	√
ADD_S	Add with saturation	√	√	√	√		
ADD_N	Add with unspecified handling of overflow	√	√	√	√	√	√
SUB_M	Subtract with modulo	√	√	√	√	√	√
SUB_S	Subtract with saturation	√	√	√	√		
SUB_N	Subtract with unspecified handling of overflow	√	√	√	√	√	√
MULT_L	Multiply low			√			
MULT_L_ADD_M	Multiply low and add with modulo			√			
MULT_L_ADD_N	Multiply low and add with unspecified handling of overflow			√			
MULT_H	Multiply high			√			
MULT_H_ADD_S	Multiply high and add with saturation			√			
MULT_ADDPAIRS	Multiply and add pairs			√			
MULT_ADDPAIRS_ADD_M	Multiply, add pairs and add a third input vector with modulo			√			
MULT_ADDPAIRS_ADD_S	Multiply, add pairs and add a third input vector with saturation			√			
MULT_ADDPAIRS_ADD_N	Multiply, add pairs and add a third input with unspecified handling of overflow			√			
AVG	Average		√		√		
MIN	Minimum		√	√			
MAX	Maximum		√	√			
CLIP	Clip all elements to a value			√			
SAD2	Sum of absolute differences with two partial sums		√				
SAD2_ADD_M	SAD and add with modulo		√				
SUM2	Add two partial sums					√	

4.10 Comparison Instructions

There are several instructions that can perform parallel comparisons. For each partition, the result is either zero if the comparison is false, or all bits set to ones if it is true. Parallel comparison operators are useful when combined with bit-wise *select*, or *masked-store* operations. SSE, SSE2 and AltiVec support parallel comparison instructions directly; TriMedia does not. On TriMedia a parallel comparison can be emulated using 32-bit comparisons and masks, and the *MUX* intrinsic to set the result to all zeros or all ones:

```
#define CMP_EQ_I16x8(dst, src1, src2)                          \
    dst##_0 = MUX((src1##_0 & 0xFFFF0000) ==                   \
                  (src2##_0 & 0xFFFF0000), 0xFFFF0000, 0) |    \
              MUX((src1##_0 & 0x0000FFFF) ==                   \
                  (src2##_0 & 0x0000FFFF), 0x0000FFFF, 0);     \
```

The same is done for *dst##_1, dst##_2* and *dst##_3*, to complete the 128-bit comparison. The parallel comparison operations supported in the common virtual instruction set are shown in Table 4.11.

Table 4.11
MMM comparison instructions

MMM Macro	Description	I8x16	U8x16	I16x8	U16x8	I32x4	U32x4	F32x4
CMP_EQ	Compare equal	√	√	√	√	√	√	√
CMP_GT	Compare greater-than	√	√	√	√	√	√	√
CMP_GTE	Compare greater-than or equal							√
CMP_LT	Compare less-than	√	√	√	√	√	√	√
CMP_LTE	Compare less-than or equal							√
CMP_NEQ	Compare not equal							√

4.11 Summary

This chapter discussed the common virtual instruction set, a group of instructions that can be emulated efficiently on all the four target architectures. The instructions defined cover vector declaration, load and store, set, shift, bit-wise logical, floating-point and integer arithmetic, conversion and rearrangement operations. All vectors and operations are 128 bits long. Multiple variables are used to emulate the vectors on architectures with smaller registers. Several strategies are used to emulate operations that are not directly supported on some instruction sets. The resulting common virtual instruction set is fairly complete, and should be enough for many applications.

The complete list of MMM macros in the common virtal instruction set is in Appendix A. It lists the instruction or instructions each macro maps to on each target. The next chapter shows how example programs are written using these macros.

Chapter 5

EXAMPLE PROGRAMS

This chapter discusses the implementation of the example programs selected: Inverse Discrete Cosine Transform (IDCT) of 8x8 blocks, L_1-Distance of 16x16 blocks, and 16x16 L_1-Distance with horizontal and vertical interpolation. The context and background of these examples was covered in Section 3.2.4. Optimized reference implementations were discussed in Section 3.2.5.

The examples in this chapter are implemented in MMM using the common virtual instruction set defined in Chapter 4. For each example, I discuss the algorithm and instructions used by the portable MMM programs. Then I describe variations that execute the fastest on each target architecture.

5.1 8x8 IDCT

The portable version of IDCT is based on Intel's algorithm for SSE2 [44], which is also the same algorithm used in the MMM+SSE version [43]. It performs first a horizontal IDCT of every row, and then a vertical IDCT of each column. I will explain the horizontal IDCT portion first, and then the vertical in the next section.

5.1.1 Horizontal IDCT

The horizontal IDCT implements the decomposition in equation (3.3), repeated below for convenience:

$$C_8^{-1} = \frac{1}{2} A_8^{-1} M_8^{-1} P_8^{-1} \tag{5.1}$$

$$A_8^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \end{bmatrix} \quad P_8^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M_8^{-1} = \begin{bmatrix} c_4 & c_2 & c_4 & c_6 & 0 & 0 & 0 & 0 \\ c_4 & c_6 & -c_4 & -c_2 & 0 & 0 & 0 & 0 \\ c_4 & -c_6 & -c_4 & c_2 & 0 & 0 & 0 & 0 \\ c_4 & -c_2 & c_4 & -c_6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & c_1 & c_3 & c_5 & c_7 \\ 0 & 0 & 0 & 0 & c_3 & -c_7 & -c_1 & -c_5 \\ 0 & 0 & 0 & 0 & c_5 & -c_1 & c_7 & c_3 \\ 0 & 0 & 0 & 0 & c_7 & -c_5 & c_3 & -c_1 \end{bmatrix}$$

The first thing the program does is to declare all the vector variables that it will need. All data vectors are 128-bit long, with 16 or 32-bit signed integer partitions. The mapping of these variables into registers is done by the target compiler.

```
void Idct8x8 ( INT16 *pSrc, INT16 *pDst)
{

   DECLARE_I16x8(X)    /* Input row */
   DECLARE_I16x8(XP)   /* Input row permuted */
   DECLARE_I16x8(XB)   /* Two columns of row repeated 4 times */

   DECLARE_I32x4(MP)   /* Partial results of operator M */
   DECLARE_I32x4(ME)   /* Result of operator M, even part */
   DECLARE_I32x4(MO)   /* Result of operator M, odd part */
   DECLARE_I32x4(A1)   /* Partial results of operator A */
   DECLARE_I32x4(A2)
```

```
DECLARE_I16x8(Y0)   /* Row IDCT outputs */
DECLARE_I16x8(Y1)
DECLARE_I16x8(Y2)
DECLARE_I16x8(Y3)
DECLARE_I16x8(Y4)
DECLARE_I16x8(Y5)
DECLARE_I16x8(Y6)
DECLARE_I16x8(Y7)
```

The results of the horizontal IDCT are held in the local vector variables *Y0* – *Y7*, and then used as inputs to the vertical IDCT. This is different from Intel's implementation, which stores all but two of the intermediate results into memory. Small constant vectors are declared as local variables, and then set to their desired values:

```
DECLARE_I32x4(ConstRound12Bit)

SET1_I32x4(ConstRound12Bit, 0x800)
```

Larger arrays of constants are declared as static arrays of vectors outside the scope of the IDCT function, using the MMM macros designed for this purpose:

```
/* Operator M8 coefficients in 2x4 groups, scaled by C1 */
DECLARE_CONST_I16x8x4(ConstM_C1, C1C4, C1C2, C1C4, C1C6, C1C4, -C1C6, C1C4, -C1C2,
                                 C1C4, C1C6, -C1C4, -C1C2, -C1C4, C1C2, C1C4, -C1C6,
                                 C1C1, C1C3, C1C3, -C1C7, C1C5, -C1C1, C1C7, -C1C5,
                                 C1C5, C1C7, -C1C1, -C1C5, C1C7, C1C3, C1C3, -C1C1)

/* Operator M8 coefficients in 2x4 groups, scaled by C2 */
DECLARE_CONST_I16x8x4(ConstM_C2, C2C4, C2C2, C2C4, C2C6, C2C4, -C2C6, C2C4, -C2C2,
                                 C2C4, C2C6, -C2C4, -C2C2, -C2C4, C2C2, C2C4, -C2C6,
                                 C1C2, C2C3, C2C3, -C2C7, C2C5, -C1C2, C2C7, -C2C5,
                                 C2C5, C2C7, -C1C2, -C2C5, C2C7, C2C3, C2C3, -C1C2);

/* Operator M8 coefficients in 2x4 groups, scaled by C3 */
DECLARE_CONST_I16x8x4(ConstM_C3, C3C4, C2C3, C3C4, C3C6, C3C4, -C3C6, C3C4, -C2C3,
                                 C3C4, C3C6, -C3C4, -C2C3, -C3C4, C2C3, C3C4, -C3C6,
                                 C1C3, C3C3, C3C3, -C3C7, C3C5, -C1C3, C3C7, -C3C5,
                                 C3C5, C3C7, -C1C3, -C3C5, C3C7, C3C3, C3C3, -C1C3);

/* Operator M8 coefficients in 2x4 groups, scaled by C4 */
DECLARE_CONST_I16x8x4(ConstM_C4, C4C4, C2C4, C4C4, C4C6, C4C4, -C4C6, C4C4, -C2C4,
                                 C4C4, C4C6, -C4C4, -C2C4, -C4C4, C2C4, C4C4, -C4C6,
                                 C1C4, C3C4, C3C4, -C4C7, C4C5, -C1C4, C4C7, -C4C5,
                                 C4C5, C4C7, -C1C4, -C4C5, C4C7, C3C4, C3C4, -C1C4);
```

Each of these arrays represents the coefficients in operator M_8^{-1}, but scaled by C_1, C_2, C_3 and C_4 respectively. Using four different sets of coefficients saves operations in the vertical IDCT.

This is discussed further in the next section. The coefficients are represented as 16-bit signed numbers, with 15 bits of fractional precision. The definition of the constants is as follows:

```
#define C1C1 31521 /* Cos(1*pi/16)*Cos(1*pi/16) << 15 */
#define C1C2 29692 /* Cos(1*pi/16)*Cos(2*pi/16) << 15 */
#define C1C3 26722 /* Cos(1*pi/16)*Cos(3*pi/16) << 15 */
#define C1C4 22725 /* Cos(1*pi/16)*Cos(4*pi/16) << 15 */
#define C1C5 17855 /* Cos(1*pi/16)*Cos(5*pi/16) << 15 */
#define C1C6 12299 /* Cos(1*pi/16)*Cos(6*pi/16) << 15 */
#define C1C7 6270  /* Cos(1*pi/16)*Cos(7*pi/16) << 15 */

#define C2C2 27969 /* Cos(2*pi/16)*Cos(2*pi/16) << 15 */
#define C2C3 25172 /* Cos(2*pi/16)*Cos(3*pi/16) << 15 */
#define C2C4 21407 /* Cos(2*pi/16)*Cos(4*pi/16) << 15 */
#define C2C5 16819 /* Cos(2*pi/16)*Cos(5*pi/16) << 15 */
#define C2C6 11585 /* Cos(2*pi/16)*Cos(6*pi/16) << 15 */
#define C2C7 5906  /* Cos(2*pi/16)*Cos(7*pi/16) << 15 */

#define C3C3 22654 /* Cos(3*pi/16)*Cos(3*pi/16) << 15 */
#define C3C4 19266 /* Cos(3*pi/16)*Cos(4*pi/16) << 15 */
#define C3C5 15137 /* Cos(3*pi/16)*Cos(5*pi/16) << 15 */
#define C3C6 10426 /* Cos(3*pi/16)*Cos(6*pi/16) << 15 */
#define C3C7 5315  /* Cos(3*pi/16)*Cos(7*pi/16) << 15 */

#define C4C4 16384 /* Cos(4*pi/16)*Cos(4*pi/16) << 15 */
#define C4C5 12873 /* Cos(4*pi/16)*Cos(5*pi/16) << 15 */
#define C4C6 8867  /* Cos(4*pi/16)*Cos(6*pi/16) << 15 */
#define C4C7 4520  /* Cos(4*pi/16)*Cos(7*pi/16) << 15 */
```

The horizontal IDCT of each row is computed using the appropriate set of coefficients. The order is chosen so that the last outputs are the first to be used by the vertical IDCT, which improves the chance that they can be kept in registers, and not have to be stored in memory.

```
ROW_IDCT(Y3, (pSrc + 3 * 8), ConstM_C3);
ROW_IDCT(Y5, (pSrc + 5 * 8), ConstM_C3);
ROW_IDCT(Y1, (pSrc + 1 * 8), ConstM_C1);
ROW_IDCT(Y7, (pSrc + 7 * 8), ConstM_C1);
ROW_IDCT(Y2, (pSrc + 2 * 8), ConstM_C2);
ROW_IDCT(Y6, (pSrc + 6 * 8), ConstM_C2);
ROW_IDCT(Y0, (pSrc + 0 * 8), ConstM_C4);
ROW_IDCT(Y4, (pSrc + 4 * 8), ConstM_C4);
```

Each horizontal IDCT is computed using the decomposition in equation (5.1) directly. It first loads one row of the input array:

```
#define ROW_IDCT(Y, pSrc, pConst);                                  \
{                                                                   \
    /* Load input row */                                            \
    LOAD_A_I16x8(X, pSrc);                                          \
```

Operator P_8^{-1} permutes the inputs from order [0 1 2 3 4 5 6 7] into order [0 2 4 6 1 3 5 7]. This particular permutation is expensive to implement in SSE2. Instead I permute it into order [0 2 1 3 4 6 5 7]:

```
PERMUTE_I16x8_02134657(XP, X);
```

It is not important that the pairs of elements are shuffled, because of the way they are used next. The implementation of operator M_8^{-1} is easiest understood by working backwards. Consider the even part of the operator. It is of the form:

$$M_{4E}^{-1} = \begin{bmatrix} c_{0,0} & c_{1,0} & c_{2,0} & c_{3,0} \\ c_{0,1} & c_{1,1} & c_{2,1} & c_{3,1} \\ c_{0,2} & c_{1,2} & c_{2,2} & c_{3,2} \\ c_{0,3} & c_{1,3} & c_{2,3} & c_{3,3} \end{bmatrix} \tag{5.2}$$

The desired result of this operator is:

$$\begin{bmatrix} x_0c_{0,0} + x_2c_{1,0} + x_4c_{2,0} + x_6c_{3,0} \\ x_0c_{0,1} + x_2c_{1,1} + x_4c_{2,1} + x_6c_{3,1} \\ x_0c_{0,2} + x_2c_{1,2} + x_4c_{2,2} + x_6c_{3,2} \\ x_0c_{0,3} + x_2c_{1,3} + x_4c_{2,3} + x_6c_{3,3} \end{bmatrix} = M_{4E}^{-1} \begin{bmatrix} x_0 \\ x_2 \\ x_4 \\ x_6 \end{bmatrix} \tag{5.3}$$

We can use the *multiply-add-pairs* instruction to compute two products and an addition of the form $A \cdot B + C \cdot D$, where A & C and B & D are adjacent partitions in two vectors. Since the vectors are 128-bits long, four of these operations can be done in parallel. If we pack the first two coefficients of every row into a vector C_{02}=[$c_{0,0}$ $c_{1,0}$ $c_{0,1}$ $c_{1,1}$ $c_{0,2}$ $c_{1,2}$ $c_{0,3}$ $c_{1,3}$], and do a *multiply-add-pairs* operation with a vector that has inputs 0 and 2 repeated: X_{02}=[x_0 x_2 x_0

$x_2\ x_0\ x_2\ x_0\ x_2$], we obtain a vector with the left half of the desired results for each row of equation (5.3):

$$M_{02} = [x_0c_{0,0}+x_2c_{1,0}\ \ x_0c_{0,1}+x_2c_{1,1}\ \ x_0c_{0,2}+x_2c_{1,2}\ \ x_0c_{0,3}+x_2c_{1,3}]$$

Each of the elements in vector M_{02} is a 32-bit integer. A similar arrangement can produce a vector with the right half of the desired results:

$$M_{46} = [x_4c_{2,0}+x_6c_{3,0}\ \ x_4c_{2,1}+x_6c_{3,1}\ \ x_4c_{2,2}+x_6c_{3,2}\ \ x_4c_{2,3}+x_6c_{3,3}]$$

These two vectors can be added using parallel addition to obtain the desired result. The same technique can be applied to the odd part of operator M_8^{-1}. In order to create the vector X_{02} in MMM I used the *broadcast-pair* operation, which copies two adjacent elements to the rest of the vector. Elements x_0 and x_2 are the first pair in vector *XP*:

```
BROADCAST_PAIR_0_I16x8(XB, XP);
```

The coefficients for operator M_8^{-1} are stored in memory, so they need to be read into a temporary vector before they are used:

```
LOAD_A_I16x8(Temp, &pConst[0]);
MULT_ADDPAIRS_I16x8(MP, XB, Temp);
```

For SSE and SSE2 this *load* operation is removed by the target compiler, because the *multiply-add-pairs* instruction can take a memory address as the second argument. The same instructions are used to compute M_{46}, which is then added to M_{02} to complete the even part of operator M_8^{-1}. The addition is combined with the second *multiply-add-pairs*, as this can be done with a single instruction in AltiVec. Since the inputs to the IDCT only use 9 bits, and the constants use 15, the products use at most 24 bits of each 32-bit partition. We

can safely use addition with unspecified handling of overflow. The inputs x_4 and x_6 are the third pair in vector *XP*, and the constants are the second row in the array *pConst*:

```
BROADCAST_PAIR_2_I16x8(XB, XP);
LOAD_A_I16x8(Temp, &pConst[1]);
MULT_ADDPAIRS_ADD_N_I16x8(ME, XB, Temp, MP);
```

Now vector ME holds the four results of the even part of M_8^{-1}, as 32-bit values. The odd part is computed in a very similar way:

```
BROADCAST_PAIR_1_I16x8(XB, XP);
LOAD_A_I16x8(Temp, &pConst[2]);
MULT_ADDPAIRS_I16x8(MP, XB, Temp);

BROADCAST_PAIR_3_I16x8(XB, XP);
LOAD_A_I16x8(Temp, &pConst[3]);
MULT_ADDPAIRS_ADD_N_I16x8(MO, XB, Temp, MP);
```

The last part of the horizontal IDCT is operator $A_8^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \end{bmatrix}$.

The top half of this operator adds the first four rows of the input with the second four. This is simply the parallel addition of vectors *ME* and *MO*. The bottom part is the subtraction of *ME* and *MO*, but in reverse order. For the time being it will just do the subtraction, and correct the order later. Once again we use addition and subtraction with unspecified overflow handling:

```
ADD_N_I32x4(ME, ME, ConstRound12Bit);
ADD_N_I32x4(A1, ME, MO);
SUB_N_I32x4(A2, ME, MO);
```

The first operation adds a rounding amount to both *A1* and *A2*. This helps preserve accuracy when converting back to 16 bits. This is done next with arithmetic *shift-right*s and *pack*s:

```
SRA_I_I32x4(A1, A1, 12);
SRA_I_I32x4(A2, A2, 12);
PACK_N_I32x4(Y, A1, A2);
```

Only the least significant 12 bits are shifted out, to preserve accuracy. This is compensated at the end of the vertical IDCT. After the shifts, the quantities are known to be within the lower 16 bits, so we can use the conversion instruction with unspecified reduction type. Lastly, we correct the order of the last four elements by using a permutation:

```
PERMUTE_I16x8_01237654(Y, Y);
```

5.1.2 Vertical IDCT

The vertical IDCT is performed for the eight columns in parallel. The eight outputs of the horizontal IDCT (`Y0 - Y7`) are the inputs to the vertical IDCT. Every operation in the IDCT becomes a parallel operation on the input vectors, so it makes sense to use an IDCT with minimal number of operations. The IDCT algorithm is based on decomposition (5.1), but factorizes matrix M_8^{-1} further as follows:

$$M_8^{-1} = F_8^{-1} E_8^{-1} B_8^{-1} D_8^{-1} \tag{5.4}$$

where $F_8^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & c_4 & c_4 & 0 \\ 0 & 0 & 0 & 0 & 0 & c_4 & -c_4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$, $E_8^{-1} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \end{bmatrix}$,

$$B_8^{-1} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & t_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & t_2 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & t_1 & 0 & 0 \\ 0 & 0 & 0 & 0 & t_1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & t_3 \\ 0 & 0 & 0 & 0 & 0 & 0 & t_3 & -1 \end{bmatrix}, \quad D_8^{-1} = \begin{bmatrix} c_4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & c_4 & 0 & 0 & 0 & 0 & 0 \\ 0 & c_2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & c_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & c_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & c_1 \\ 0 & 0 & 0 & 0 & 0 & c_3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & c_3 & 0 \end{bmatrix}$$

and $t_k = \tan(\pi k/16)$. Operator P_8^{-1} in (5.1) is a permutation of the inputs. No operation is necessary because each input is represented by a different variable. Operator D_8^{-1} represents a scaling factor for each input. This has already been done by using scaled coefficients in the horizontal IDCTs. The constants t_1, t_2, t_3 and c_4 are represented as unsigned 16-bit integers with 16 fractional bits.

```
#define TAN1 (UINT16) 13036 /* Tan(1*pi/16) << 16 */
#define TAN2 (UINT16) 27146 /* Tan(2*pi/16) << 16 */
#define TAN3 (UINT16) 43790 /* Tan(3*pi/16) << 16 */
#define COS4 (UINT16) 46341 /* Cos(4*pi/16) << 16 */
```

The horizontal IDCT declares four vectors and sets all their elements to these constants, plus a few other constant vectors used for rounding:

```
DECLARE_I16x8(ConstTan1)
DECLARE_I16x8(ConstTan2)
DECLARE_I16x8(ConstTan3)
DECLARE_I16x8(ConstCos4)
DECLARE_I16x8(ConstCorr)
DECLARE_I16x8(ConstRound5Bit)
DECLARE_I16x8(ConstRound5BitCorr)

SET1_I16x8(ConstTan1, TAN1)
SET1_I16x8(ConstTan2, TAN2)
SET1_I16x8(ConstTan3, TAN3)
SET1_I16x8(ConstCos4, COS4)
SET1_I16x8(ConstCorr, 0x1)
SET1_I16x8(ConstRound5Bit, 0x10)
SET1_I16x8(ConstRound5BitCorr, 0xF)
```

The vectors are declared as signed instead of unsigned because the *multiply-high* operations are only available for signed partitions. This makes no difference for the constants *TAN1* and *TAN2*, because they are less than 0.5 and thus have the same meaning as signed or unsigned quantities. But *TAN3* and *COS4* are greater than 0.5 and become negative when interpreted as signed quantities. For example, the hexadecimal number *0xB505* interpreted as an unsigned value equals 46341, which is $\cos(\pi/4)$ with 16 fractional bits. But *0xB505* interpreted as a signed number is –19195, which equals $\cos(\pi/4)-1$. When using signed multiplication with these constants we can obtain the desired result by adding the operand to the product: $x \bullet c = x \circ (c+1) = x \circ c + x$, where $\bullet$ denotes unsigned multiplication, and $\circ$ is signed multiplication.

Operator B_8^{-1} is done using parallel 16-bit multiplications (*multiply-high*), additions and subtractions. Whenever possible, a combined *multiply-high-add* instruction is used. There is no risk of overflow, so I use instruction with no special handling of overflow:

```
ADD_N_I16x8(B0, Y0, Y4)
SUB_N_I16x8(B1, Y0, Y4)
MULT_H_ADD_N_I16x8(B2, Y6, ConstTan2, Y2)
MULT_H_I16x8(Temp, Y2, ConstTan2)
SUB_N_I16x8(B3, Temp, Y6)
MULT_H_ADD_N_I16x8(B4, Y7, ConstTan1, Y1)
MULT_H_I16x8(Temp, Y1, ConstTan1)
SUB_N_I16x8(B5, Temp, Y7)
MULT_H_ADD_N_I16x8(Temp, Y5, ConstTan3, Y5)
ADD_N_I16x8(B6, Temp, Y3)
MULT_H_ADD_N_I16x8(Temp, Y3, ConstTan3, Y3)
SUB_N_I16x8(B7, Y5, Temp)
```

Operator E_8^{-1} requires only additions and subtractions. Some constant correction vectors are added to help reduce the truncation error. The rounding vectors are also added here, so that they get propagated into all the outputs:

```
ADD_N_I16x8(E0, B0, B2)
ADD_N_I16x8(E0, E0, ConstRound5Bit)
SUB_N_I16x8(E3, B0, B2)
ADD_N_I16x8(E3, E3, ConstRound5BitCorr)
ADD_N_I16x8(E1, B1, B3)
ADD_N_I16x8(E1, E1, ConstRound5Bit)
SUB_N_I16x8(E2, B1, B3)
ADD_N_I16x8(E2, E2, ConstRound5BitCorr)
ADD_N_I16x8(E4, B4, B6)
ADD_N_I16x8(E4, E4, ConstCorr)
SUB_N_I16x8(E5, B4, B6)
SUB_N_I16x8(E6, B5, B7)
ADD_N_I16x8(E6, E6, ConstCorr)
ADD_N_I16x8(E7, B5, B7)
```

Operator F_8^{-1} involves a couple of products and a few more additions:

```
ADD_I16x8(Temp, E5, E6)
MULT_H_ADD_N_I16x8(F5, Temp, ConstCos4, Temp)
ADD_N_I16x8(F5, F5, ConstCorr)
SUB_N_I16x8(Temp, E5, E6)
MULT_H_ADD_N_I16x8(F6, Temp, ConstCos4, Temp)
ADD_N_I16x8(F6, F6, ConstCorr)
```

Finally, operator A_8^{-1} adds or subtracts pairs of vectors, scales the outputs to the right level by shifting-out the lowest 5 bits, and stores the results. The shift amount is fixed, so we can use the *`shift-immediate`* instruction:

```
/* Y0 */
ADD_N_I16x8(Temp, E0, E4)
SRA_I_I16x8(Temp, Temp, 5)
STORE_A_I16x8((pDst + 0*8), Temp)

/* Y7 */
SUB_N_I16x8(Temp, E0, E4)
SRA_I_I16x8(Temp, Temp, 5)
STORE_A_I16x8((pDst + 7*8), Temp)

/* Y1 */
ADD_N_I16x8(Temp, E1, F5)
SRA_I_I16x8(Temp, Temp, 5)
STORE_A_I16x8((pDst + 1*8), Temp)

/* Y6 */
SUB_N_I16x8(Temp, E1, F5)
SRA_I_I16x8(Temp, Temp, 5)
STORE_A_I16x8((pDst + 6*8), Temp)

/* Y2 */
ADD_N_I16x8(Temp, E2, F6)
```

```
    SRA_I_I16x8(Temp, Temp, 5)
    STORE_A_I16x8((pDst + 2*8), Temp)

    /* Y5 */
    SUB_N_I16x8(Temp, E2, F6)
    SRA_I_I16x8(Temp, Temp, 5)
    STORE_A_I16x8((pDst + 5*8), Temp)

    /* Y3 */
    ADD_N_I16x8(Temp, E3, E7)
    SRA_I_I16x8(Temp, Temp, 5)
    STORE_A_I16x8((pDst + 3*8), Temp)

    /* Y4 */
    SUB_N_I16x8(Temp, E3, E7)
    SRA_I_I16x8(Temp, Temp, 5)
    STORE_A_I16x8((pDst + 4*8), Temp)
```

The IDCT concludes with the macro:

```
    END_OPTIMIZED();
}
```

which empties the MMX registers. This is required before any other program can use floating-point registers; it does nothing on the other architectures. As a general rule MMM programs use this instruction after any optimized module.

This IDCT meets the IEEE 1180 accuracy requirements in all the four target architectures. The final form of the IDCT implementation in MMM appears in Appendix C.

5.1.3 Target-Specific Optimizations

This section discusses attempts to further optimize the IDCT for each target architecture, but maintaining the same algorithm. I tested variations in the implementation and instructions that favor each specific architecture, even if it makes the program non-portable. The purpose of this exercise is to determine how much performance is lost to portability for a given algorithm.

On AltiVec, the emulation of *multiply-high* instructions requires a shift of one of the arguments, to account for the implicit factor of 2 added by the *vec_madds* intrstruction.

```
#define MULT_H_I16x8(dst, src1, src2)                          \
    dst = vec_madds(src1, vec_sra(src2, (vector UINT16) (1)),  \
                    (vector INT16) (0));
```

This can be avoided by using constants with 15 fractional bits, instead of 16. This also resolves the problem of having constants greater than 0.5 be interpreted as negative numbers. Constants with 15 bits of precision can never be negative. This saves three extra additions. With these optimizations the IDCT runs 4% faster than the portable version.

On TriMedia the IDCT can be made a little faster (1.3%) by storing all the coefficients for operator M_8^{-1} in local variables, rather than in memory. This is possible because TriMedia has a very large number of registers. For SSE and SSE2 I could find no way to improve upon the portable MMM version. The performance measurements of all versions are shown in Chapter 6.

5.2 16x16 L_1-Distance

This example computes the L_1-Distance of a 16x16 block, as described in Section 3.2.4.2.

5.2.1 Portable MMM Design

The portable MMM version of 16x16 L_1-Distance is completely unrolled, and it always assumes that the reference block is unaligned. It accumulates two partial sums of the absolute differences of each row, and adds them into a single quantity at the end. It starts by declaring vector variables that hold one row of each block (*R1* and *I*), one vector to accumulate the partial sums, and the integer result. It clears the partial sum to zero:

```
UINT32 L1Dist16x16(UINT8 *pRef, UINT8 *pIn,
                   int RowPitch, int Limit)
{
   DECLARE_U8x16(R1)   /* Holds one row of reference block */
   DECLARE_U8x16(I)    /* Holds one row of input block /
   DECLARE_U32x4(Sad)  /* Vector with two partial sums */
   UINT32 Sum;         /* Integer result */

   CLEAR_U32x4(Sad)
```

The reference block is assumed to be unaligned. I use the following MMM macro to prepare the re-alignment of all rows:

```
   PREPARE_LOAD_ALIGNMENT(1, pRef)
```

The first parameter is the realignment index, as there can be multiple re-alignments prepared. On AltiVec, this macro computes a permutation vector based on the address of *pRef*. On TriMedia, it computes shift amounts. On SSE and SSE2 it does nothing. Then it accumulates the sum of absolute differences of each row in a completely unrolled manner:

```
   SAD_ROW(Sad, pRef + 0*RowPitch, pIn + 0*RowPitch, 1)
   SAD_ROW(Sad, pRef + 1*RowPitch, pIn + 1*RowPitch, 1)
   SAD_ROW(Sad, pRef + 2*RowPitch, pIn + 2*RowPitch, 1)
   SAD_ROW(Sad, pRef + 3*RowPitch, pIn + 3*RowPitch, 1)
   SAD_ROW(Sad, pRef + 4*RowPitch, pIn + 4*RowPitch, 1)
   SAD_ROW(Sad, pRef + 5*RowPitch, pIn + 5*RowPitch, 1)
   SAD_ROW(Sad, pRef + 6*RowPitch, pIn + 6*RowPitch, 1)
   SAD_ROW(Sad, pRef + 7*RowPitch, pIn + 7*RowPitch, 1)
   SAD_ROW(Sad, pRef + 8*RowPitch, pIn + 8*RowPitch, 1)
   SAD_ROW(Sad, pRef + 9*RowPitch, pIn + 9*RowPitch, 1)
   SAD_ROW(Sad, pRef +10*RowPitch, pIn +10*RowPitch, 1)
   SAD_ROW(Sad, pRef +11*RowPitch, pIn +11*RowPitch, 1)
   SAD_ROW(Sad, pRef +12*RowPitch, pIn +12*RowPitch, 1)
   SAD_ROW(Sad, pRef +13*RowPitch, pIn +13*RowPitch, 1)
   SAD_ROW(Sad, pRef +14*RowPitch, pIn +14*RowPitch, 1)
   SAD_ROW(Sad, pRef +15*RowPitch, pIn +15*RowPitch, 1)
```

The *SAD_ROW* macro computes the sum of absolute differences of a row and accumulates the partial results. It uses the *SAD2_ADD* MMM macro to accumulate the result, which is more efficient than a separate add in some architectures:

```
#define SAD_ROW(dst, pRef, pIn, index)    \

    LOAD_U_U8x16(R1, pRef, index)         \
    LOAD_A_U8x16(I, pIn)                  \
    SAD2_ADD_M_U8x16(dst, R1, I, dst)
```

After all rows have been processed, the vector *Sad* holds two partial sums. They are added and the result is converted into an integer value. The sum is returned after clearing the state:

```
    SUM2_U32x4(Sum, Sad)

    END_OPTIMIZED()
    return Sum;
}
```

This example does not take advantage of the *Limit* parameter. In order to decide to exit the function early, one must compare a partial distance with the limit. In a scalar implementation this does not represent much overhead, and can be done after every row. But in an optimized implementation, the branch penalty introduced can be expensive. Also, since the optimized MMM implementation maintains two partial sums in a vector, one needs to add the two and convert the result to an integer before doing the comparison. For this reason, I implemented a version of 16x16 L_1-Distance with a single shortcut path after half the rows:

```
    SAD_ROW(Sad, pRef + 6*RowPitch, pIn + 6*RowPitch, 1)
    SAD_ROW(Sad, pRef + 7*RowPitch, pIn + 7*RowPitch, 1)

    /* Shortcut */
    SUM2_U32x4(Sum, Sad)
        if (Sum > Limit) {
            END_OPTIMIZED()
            return Sum;
        }

    SAD_ROW(Sad, pRef + 8*RowPitch, pIn + 8*RowPitch, 1)
    SAD_ROW(Sad, pRef + 9*RowPitch, pIn + 9*RowPitch, 1)
```

The benefit of the shortcut path depends on the input data and in the motion estimation algorithm. Chapter 6 shows speed measurements of these examples in the context of an MPEG2 video encoder with natural outdoor images as inputs.

5.2.2 Target-Specific Optimizations

On TriMedia it is slightly faster to accumulate the sum of absolute differences of each row into a single integer value, which is accumulated for all rows. This has the same number of operations per row, but saves some overhead in the beginning and end. It results in a 1.4% speed improvement. On AltiVec it is more efficient to keep four partial sums, rather than two. This saves one `vec_sum2s` instruction per row. Summing four partial results at the end is no more complex than two, using the `vec_sums` instruction. Also, keeping the permutation vector in a local variable rather than a global saves some instructions. After these optimizations, this example becomes identical to the reference implementation by Motorola [49]. I could find no target-specific improvements to this program for SSE and SSE2.

5.3 16x16 L_1-Distance with Interpolation

This example computes the L_1-Distance of a 16x16 block, but also performs horizontal and vertical half-pixel interpolation on the reference block.

5.3.1 Portable MMM Design

The interpolation is defined as the average of four pixels: the current, the one to the right, the one below, and the one below and to the right:

$$avg(a,b,c,d) = \left\lfloor \frac{a+b+c+d+2}{4} \right\rfloor \tag{5.5}$$

MMM supports instructions that compute the average of two vectors. Using them one can produce an approximation to the four-pixel average:

$$avg(a,b,c,d) \approx avg(avg(a,b),avg(c,d)) = \left\lfloor \frac{\left\lfloor \frac{(a+b+1)}{2} \right\rfloor + \left\lfloor \frac{(c+d+1)}{2} \right\rfloor + 1}{2} \right\rfloor \quad (5.6)$$

In simulations, this approximation was found to introduce a mean error of 0.37 to the average value (in the range of 0-255). This is acceptable for the purposes of motion estimation.

This example starts very similar to the previous one, except that it declares two more vectors to hold additional rows of the reference block:

```
int L1Dist16x16_InterpXY(UINT8 *pRef, UINT8 *pIn,
                         int RowPitch, int Limit)
{
   DECLARE_U8x16(R1)   /* Holds one row of reference block */
   DECLARE_U8x16(R2)
   DECLARE_U8x16(R3)
   DECLARE_U8x16(B)    /* Holds one row of input block */

   DECLARE_U32x4(Sad)  /* Vector with two partial sums */
   UINT32 Sum;         /* Integer result */
   CLEAR_U32x4(Sad)
```

The horizontal interpolation needs to load two adjacent vectors of the reference block; this is the top row of the block, and an overlapping row that starts one pixel to the right. Both of these can possibly be unaligned with respect to 16-byte boundaries, so I prepare the re-alignment of both using different indices:

```
   PREPARE_LOAD_ALIGNMENT(1, pRef)
   PREPARE_LOAD_ALIGNMENT(2, pRef+1)
```

Then I load the two adjacent rows using the MMM macro defined for this purpose, and average them:

```
   LOAD_ADJ_U8x16(R2, R3, pRef, 1, 2)
   AVG_U8x16(R2, R2, R3)
```

Now R2 holds the first row interpolated. I need to do the same for the second row, do vertical interpolation, load the input block row and compute the sum of absolute differences. All this is done inside a macro:

```
#define SAD_INTERP_ROW(dst, pRef, pIn, index1, index2)  \

   COPY_U8x16(R1, R2)                                   \
   LOAD_ADJ_U8x16(R2, R3, pRef, index1, index2)         \
   AVG_U8x16(R2, R2, R3) /* Interpolate horizontally */ \
   AVG_U8x16(R1, R1, R2) /* Interpolate vertically */   \
   LOAD_A_U8x16(I, pIn)                                 \
   SAD2_ADD_M_U8x16(dst, R1, I, dst)
```

This macro assumes that R2 holds the average of the previous row. It copies it to *R1* before loading and averaging the next row into *R2*. It then averages *R1* and *R2* as vertical interpolation. Then it loads the input row, which is aligned, and computes the sum of absolute differences. This is applied to all rows in an unrolled fashion:

```
   SAD_INTERP_ROW(Sad, pRef + 1*RowPitch, pIn + 0*RowPitch, 1, 2)
   SAD_INTERP_ROW(Sad, pRef + 2*RowPitch, pIn + 1*RowPitch, 1, 2)
   SAD_INTERP_ROW(Sad, pRef + 3*RowPitch, pIn + 2*RowPitch, 1, 2)
   SAD_INTERP_ROW(Sad, pRef + 4*RowPitch, pIn + 3*RowPitch, 1, 2)
   SAD_INTERP_ROW(Sad, pRef + 5*RowPitch, pIn + 4*RowPitch, 1, 2)
   SAD_INTERP_ROW(Sad, pRef + 6*RowPitch, pIn + 5*RowPitch, 1, 2)
   SAD_INTERP_ROW(Sad, pRef + 7*RowPitch, pIn + 6*RowPitch, 1, 2)
   SAD_INTERP_ROW(Sad, pRef + 8*RowPitch, pIn + 7*RowPitch, 1, 2)
   SAD_INTERP_ROW(Sad, pRef + 9*RowPitch, pIn + 8*RowPitch, 1, 2)
   SAD_INTERP_ROW(Sad, pRef +10*RowPitch, pIn + 9*RowPitch, 1, 2)
   SAD_INTERP_ROW(Sad, pRef +11*RowPitch, pIn +10*RowPitch, 1, 2)
   SAD_INTERP_ROW(Sad, pRef +12*RowPitch, pIn +11*RowPitch, 1, 2)
   SAD_INTERP_ROW(Sad, pRef +13*RowPitch, pIn +12*RowPitch, 1, 2)
   SAD_INTERP_ROW(Sad, pRef +14*RowPitch, pIn +13*RowPitch, 1, 2)
   SAD_INTERP_ROW(Sad, pRef +15*RowPitch, pIn +14*RowPitch, 1, 2)
   SAD_INTERP_ROW(Sad, pRef +16*RowPitch, pIn +15*RowPitch, 1, 2)
```

Finally, the two partial sums are added and converted into an integer result:

```
   SUM2_U32x4(Sum, Sad)
   END_OPTIMIZED()
   return Sum;

}
```

There is also a version with a shortcut path in the middle, after 8 rows. It is shown in Appendix C.

5.3.2 Target-Specific Optimizations

On TriMedia one can use `FUNSHIFTx` instructions (combine right-most *x* bytes from one vector with left-most 4-*x* bytes from second) to re-align data, but they require that the offset amount is known at compile time. TriMedia requires 4-byte alignment in the `load`s, so only four cases or realignment are possible. Using a switch statement, one can replicate the L_1-Distance function four times, one for each alignment offset. This technique adds some branching overhead, and does not help in the L_1-Distance example without interpolation, but in this case it improves the execution speed by 3.3%. On AltiVec it is beneficial to use four partial sums, and keep the re-alignment permutation vectors in local variables, like in the previous example.

5.4 Summary

This chapter discussed how portable versions of the example programs were written based on the MMM macros for the common virtual instruction set defined in Chapter 4. These examples use some of the most complex partitioned instructions available in the instruction set. The two 16x16 L_1-Distance examples have two versions each, one with a shortcut path, and one without. The portable implementations of the examples were designed to perform fairly well on all targets. I discussed possible target-specific optimizations that can be used by non-portable versions to run even faster, to compare against the portable versions. Chapter 6 presents the performance measurements of all the example programs.

Chapter 6

RESULTS

This chapter discusses the performance of the MMM example programs and the reference implementations. There are a total of five example programs, counting the variations with and without shortcut paths. The examples are 8x8 IDCT, 16x16 L_1-Distance without shortcut, 16x16 L_1-Distance with shortcut, 16x16 L_1-Distance with interpolation without shortcut, and 16x16 L_1-Distance with interpolation and shortcut. For each of these programs there are up to five versions running on each of the four targets: the portable MMM version, one or two vendor optimized reference versions, one version based on the portable design but further optimized for each target (I refer to this as the MMM-Opt version), plus a scalar version. The execution speed is measured for all programs, and the instruction count for the examples without shortcuts. The execution times presented are averages derived from the measurement of loops of calls to each example function. The average results are rounded to a single fractional digit.

From these measurements I derive the speedup and reduction in instruction counts for all optimized examples with respect to the scalar versions. The speedup is computed as:

$$Speedup = \frac{Time_{Scalar}}{Time_{Optimized}} \tag{6.1}$$

The reduction in instruction counts is computed similarly:

$$R = \frac{InstructionCount_{Scalar}}{InstructionCount_{Optimized}} \tag{6.2}$$

The following sub-sections present the results on each target platform.

6.1 TriMedia TM1300

The programs in this section were compiled with the TriMedia compiler version 2.1. They were run on a TriMedia TM1300 processor at 133 MHz. The execution speed was measured using the hardware cycle counter. The programs were invoked consecutively several thousand times to dilute the effects of cache misses on the first call. Table 6.1 shows the execution times of all the programs on TriMedia. The speed of the reference version is quoted from the documentation [46].

Table 6.1
Execution times in cycles on TriMedia TM1300

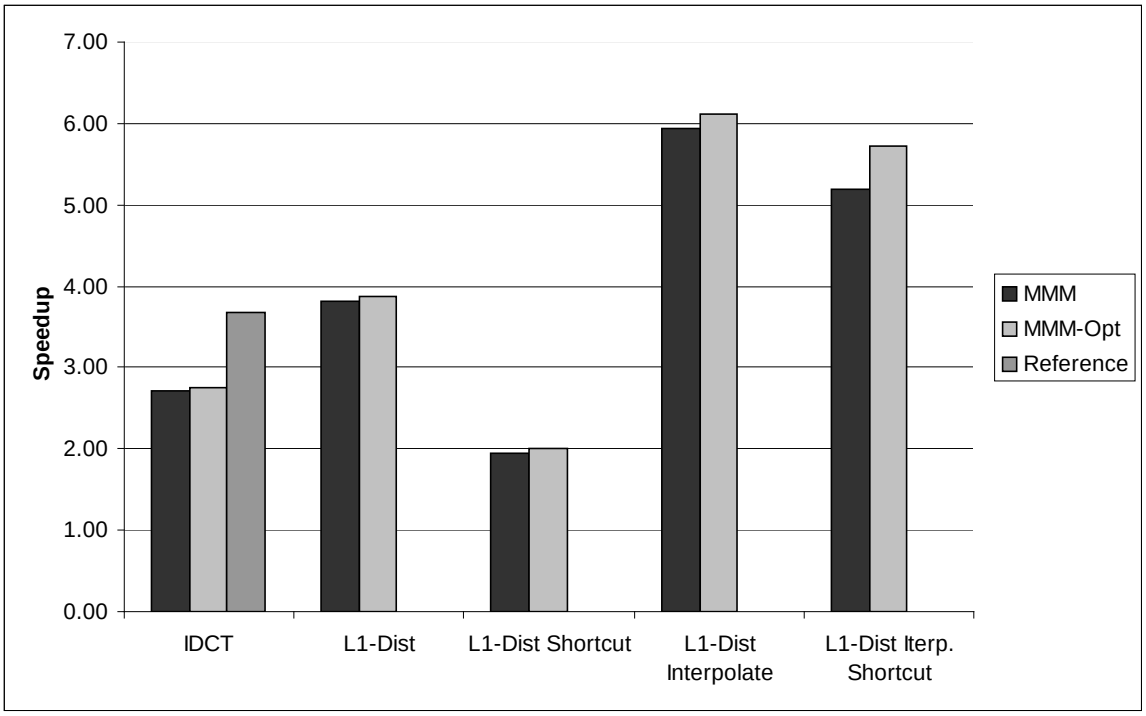
Version	8x8 IDCT	16x16 L_1-Distance		16x16 L_1-Distance with Interpolation	
		No shortcut	Shortcut	No shortcut	Shortcut
MMM	230	119.8	81.3	202.5	205
MMM-Opt	227	118.3	78.6	196.1	185.5
Reference	170				
Scalar	626	457.3	158.3	1200.5	1062.6

From these measurements we can derive the speedup obtained by the different optimized versions with respect to the speed of the scalar implementation. Table 6.2 and Figure 6.1 show the speedup of all versions of the examples running on TriMedia.

Table 6.2
Speedup of optimized examples on TriMedia TM1300

Version	8x8 IDCT	16x16 L_1-Distance		16x16 L_1-Distance with Interpolation	
		No shortcut	Shortcut	No shortcut	Shortcut
MMM	2.72	3.82	1.95	5.93	5.18
MMM-Opt	2.76	3.87	2.01	6.12	5.73
Reference	3.68				

Figure 6.1
Speedup of optimized examples on TriMedia TM1300



The execution speeds of the portable MMM examples are very close to the target-specific implementations of the same algorithms (MMM-Opt versions). The speedup is within 10% of the MMM-Opt versions in all examples. This indicates that no more than 10% of performance is lost for using portable instructions only. The reference version of IDCT is 26% faster than the portable one, because it uses an algorithm that fits better to TriMedia's short register length.

The instruction counts were measured for the programs without shortcuts. They are shown in Table 6.3. I have no measurement for the reference version, as I don't have an actual working implementation of it. I also could not measure the instruction count of the MMM-Opt version of L_1-Distance with interpolation, because it has a variable execution path.

Table 6.3
Instruction counts on TriMedia TM1300

Version	8x8 IDCT	16x16 L_1-Distance	16x16 L_1-Distance with Interpolation
MMM	918	503	862
MMM-Opt	911	498	957
Scalar	1644	1703	4943

Like before, it is useful to compare the instruction counts with respect to the scalar implementation. Table 6.4 below shows the reduction in the instruction counts for these examples.

Table 6.4
Reduction in instruction counts on TriMedia TM1300

Version	8x8 IDCT	16x16 L_1-Distance	16x16 L_1-Distance with Interpolation
MMM	1.79	3.39	5.73
MMM-Opt	1.80	3.42	

The instruction counts are very similar for the portable and target-specific versions. This confirms that not much is lost by using portable instructions only. It is interesting to note that the speedups measured on this platform are greater than the reduction in the instruction counts. This happens because the optimized versions improve the scheduling of instructions into parallel functional units.

6.2 MMX + SSE

The programs were run on a Pentium III processor running at 600 MHz. The programs were compiled with the Intel C/C++ compiler version 7.0. The execution time was measured using multimedia timers, which are cycle accurate. The reference IDCT example is written in assembly [43]. There are two L_1-Distance reference implementations from [47]. The first is written in assembly, but has some unnecessary loop overhead. The second reference is written in C with intrinsics; I removed the unnecessary loops from this example. The MMM-Opt versions are identical to the portable ones, because I could find no way to improve upon them. Table 6.5 shows the execution times of the examples on this processor; Table 6.6 and Figure 6.2 show the speedup obtained by the optimized examples with respect to the scalar version.

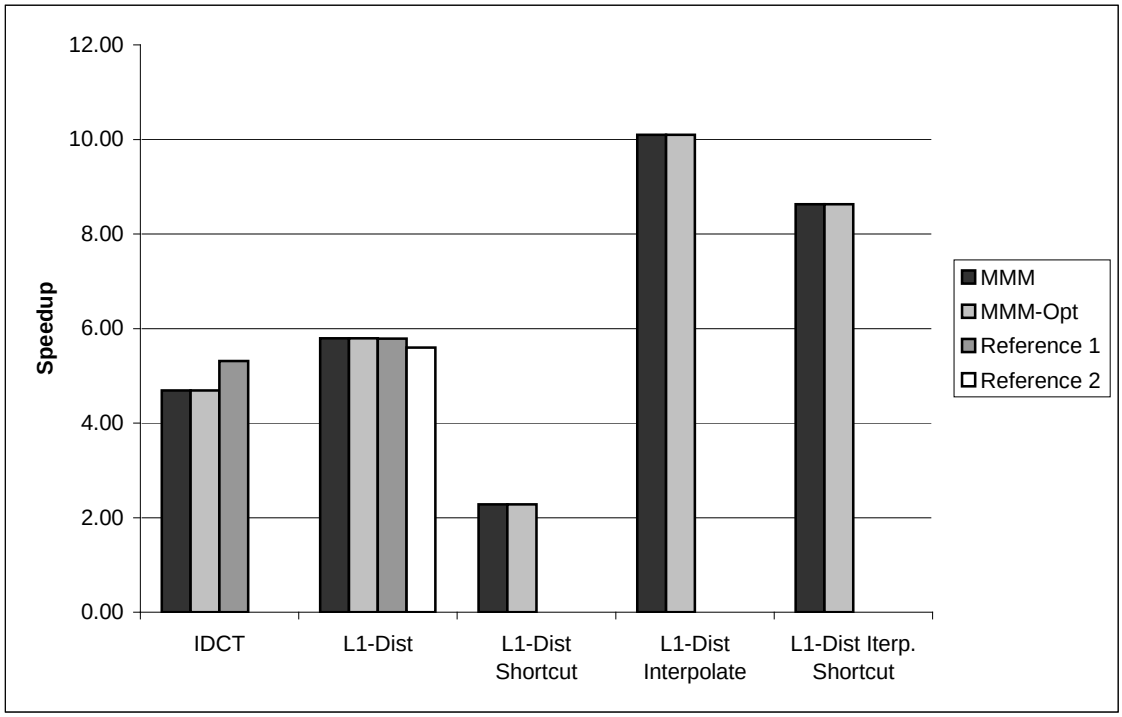
Table 6.5
Execution times in cycles on MMX + SSE

Version	8x8 IDCT	16x16 L_1-Distance		16x16 L_1-Distance with Interpolation	
		No shortcut	Shortcut	No shortcut	Shortcut
MMM	348.8	226.2	138.2	335.9	334
MMM-Opt	348.8	226.2	138.2	335.9	334
Reference1	307.9	226.4			
Reference2		234.2			
Scalar	1634.9	1310.7	315	3391.8	2882.3

Table 6.6
Speedup of optimized examples on MMX + SSE

Version	8x8 IDCT	16x16 L_1-Distance		16x16 L_1-Distance with Interpolation	
		No shortcut	Shortcut	No shortcut	Shortcut
MMM	4.69	5.79	2.28	10.10	8.63
MMM-Opt	4.69	5.79	2.28	10.10	8.63
Reference1	5.31	5.79			
Reference2		5.60			

Figure 6.2
Speedup of optimized examples on MMX + SSE



The reference IDCT example is 12% faster than the portable version, even though they implement the same algorithm. Both implementations have the same number of arithmetic operations, but the reference version has less data moves. The portable version has 5.2% more instructions, as seen below on Table 6.7. The rest of the speed difference is because the assembly version has a more efficient instruction schedule then what the Intel compiler generates for the portable version.

The L_1-Distance example is actually faster than both reference implementations. It might be possible to improve the assembly version by removing the unnecessary outer loops, but I cannot tell how much it would improve. The L_1-Distance examples with shortcut paths produce less speed improvement. This is because of optimized version has a single shortcut path in the middle, while the scalar version checks after every row, and because the shortcut

path represents a larger overhead to the optimized versions than to the scalar one. The instruction counts of the examples without shortcut paths appear in Table 6.7. Then Table 6.8 and Figure 6.3 show the reduction in instruction counts with respect to the scalar versions.

Table 6.7
Instruction counts on MMX + SSE

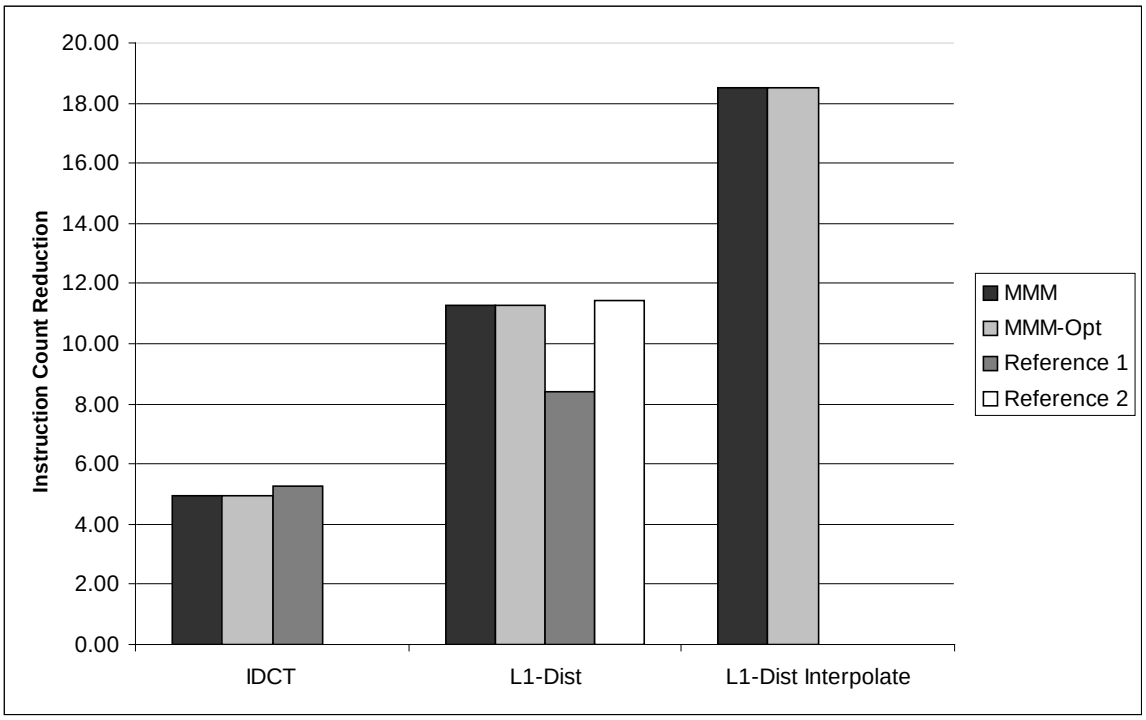
Version	8x8 IDCT	16x16 L_1-Distance	16x16 L_1-Distance with Interpolation
MMM	556	156	231
MMM-Opt	556	156	231
Reference1	527	210	
Reference2		154	
Scalar	2764	1762	4278

Table 6.8
Reduction in instruction counts on MMX + SSE

Version	8x8 IDCT	16x16 L_1-Distance	16x16 L_1-Distance with Interpolation
MMM	4.97	11.29	18.52
MMM-Opt	4.97	11.29	18.52
Reference1	5.24	8.39	
Reference2		11.44	

The reduction in the instruction count for the IDCT example is similar to the speedup obtained. But in the L_1-Distance examples, the reduction in the instruction count is much larger than the speedup. For example, the portable L_1-Distance example has over 11 times less instructions than the scalar version, but is only about 5.8 times faster. The L_1-Distance with interpolation has over 18 times less instructions and is about 10 times faster. This indicates that other resources, like memory access, are the bottleneck of these programs.

Figure 6.3
Reduction in instruction counts on MMX + SSE



6.3 SSE2

This architecture was tested on a Pentium 4 processor running at 1.7 GHz. The programs were generated by the Intel C/C++ compiler 7.0, and measured using cycle-accurate multimedia timers. The reference IDCT implementations are from Intel [44]; the first is written in assembly, and the second in C++ vector classes. Both implement the same algorithm as the MMM version. The reference L_1-Distance implementation [48] is written in C with intrinsics, and does not have the outer loop overhead that the SSE examples had. Table 6.9 shows the execution times of the example programs on SSE2. Once again, I could not find a way to speed-up the portable MMM examples further, so the MMM-Opt versions are the same as the portable MMM ones.

Table 6.9
Execution times in cycles on SSE2

Version	8x8 IDCT	16x16 L_1-Distance		16x16 L_1-Distance with Interpolation	
		No shortcut	Shortcut	No shortcut	Shortcut
MMM	315.4	166.3	117.4	301.2	326
MMM-Opt	315.4	166.3	117.4	301.2	326
Reference1	325.9	172.9			
Reference2	370.8				
Scalar	1555.5	1071.8	285.7	3873	3094.5

Table 6.10 and Figure 6.4 show the speedup with respect to the scalar version. On this platform, the portable MMM programs out-performed all other versions. The reference IDCTs and the MMM version use the same algorithm (the MMM version was derived from this reference design). They differ only in that the MMM version keeps the results of the horizontal IDCT in local variables, instead of storing them in memory. This helped save some load/store instructions.

Table 6.10
Speedup of optimized examples on SSE2

Version	8x8 IDCT	16x16 L_1-Distance		16x16 L_1-Distance with Interpolation	
		No shortcut	Shortcut	No shortcut	Shortcut
MMM	4.93	6.44	2.43	12.86	9.49
MMM-Opt	4.93	6.44	2.43	12.86	9.49
Reference1	4.77	6.20			
Reference2	4.19				

Figure 6.4
Speedup of optimized examples on SSE2

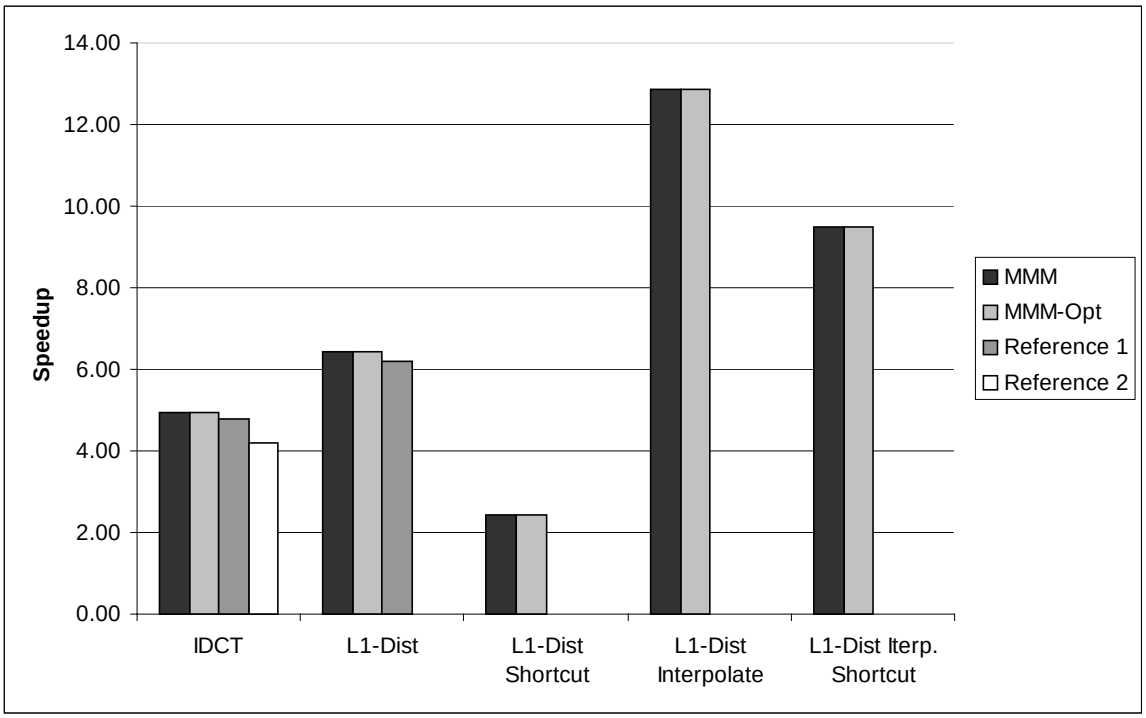


Table 6.11 shows the instruction counts of these programs. The MMM versions have the lowest instruction counts of all versions. Table 6.12 and Figure 6.5 show the reduction in instruction counts achieved by the optimized programs on SSE2.

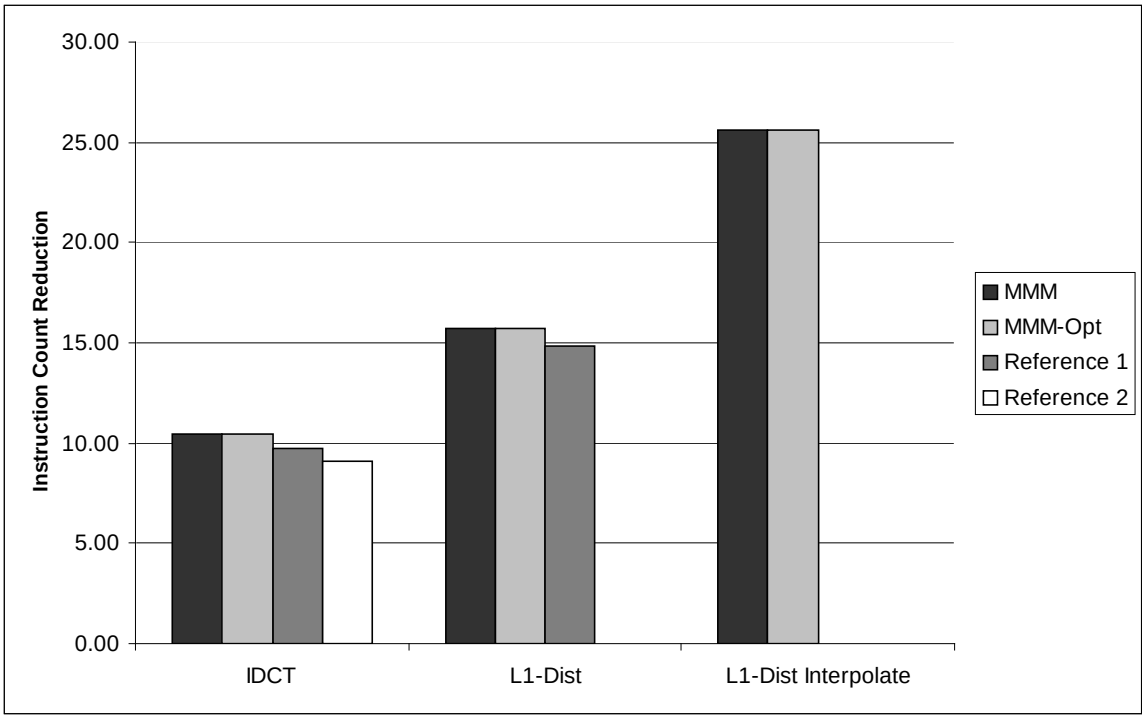
Table 6.11
Instruction counts on SSE2

Version	8x8 IDCT	16x16 L_1-Distance	16x16 L_1-Distance with Interpolation
MMM	265	112	167
MMM-Opt	265	112	167
Reference1	285	119	
Reference2	304		
Scalar	2764	1762	4278

Table 6.12
Reduction in instruction counts on SSE2

Version	8x8 IDCT	16x16 L_1-Distance	16x16 L_1-Distance with Interpolation
MMM	10.43	15.73	25.62
MMM-Opt	10.43	15.73	25.62
Reference1	9.70	14.81	
Reference2	9.09		

Figure 6.5
Reduction in instruction counts on SSE2



We see that the reductions in instruction counts are much higher than the speedups obtained, even more so than in the SSE examples. This tells that the resource contention is even more severe in this architecture.

6.4 AltiVec

The example programs were compiled for AltiVec using Apple's gcc 931.1, and run on a PowerPC G4 processor running at 400 MHz. The execution speed was measured using the clock() system call. To improve the measurement accuracy, I timed blocks of thousands of calls to the programs. This also reduces the effect of cache misses on the execution times. The first IDCT reference implementation is the one from Motorola [45], and the second one is from Apple [50]. Both are written in C with intrinsics. The L_1-Distance reference is from Motorola [49]. Table 6.13 has the execution times of the examples on AltiVec. Table 6.14 and Figure 6.6 show the speedups.

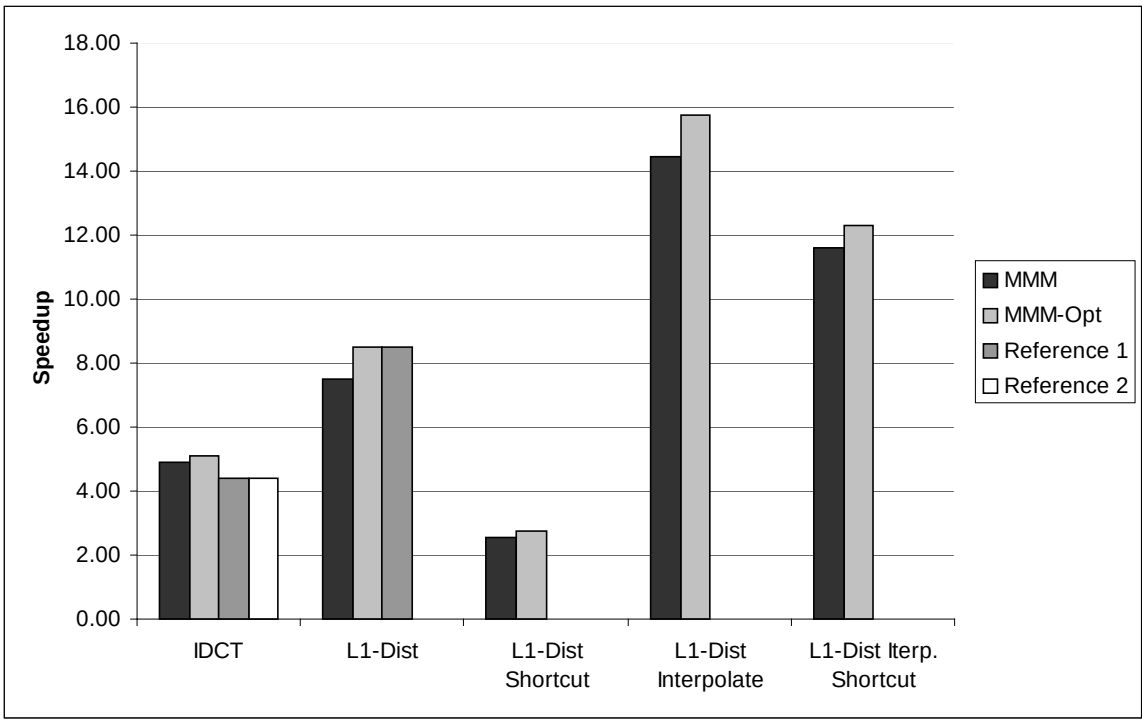
Table 6.13
Execution times in clocks on AltiVec

Version	8x8 IDCT	16x16 L_1-Distance		16x16 L_1-Distance with Interpolation	
		No shortcut	Shortcut	No shortcut	Shortcut
MMM	44.7	33.3	26.9	44.8	45.9
MMM-Opt	42.9	29.4	25.1	41.1	43.2
Reference1	49.6	29.4			
Reference2	49.9				
Scalar	219	249.6	69	646.8	532.4

Table 6.14
Speedup of optimized examples on AltiVec

Version	8x8 IDCT	16x16 L_1-Distance		16x16 L_1-Distance with Interpolation	
		No shortcut	Shortcut	No shortcut	Shortcut
MMM	4.90	7.50	2.57	14.44	11.60
MMM-Opt	5.10	8.49	2.75	15.74	12.32
Reference1	4.42	8.49			
Reference2	4.39				

Figure 6.6
Speedup of optimized examples on AltiVec



The reference IDCT implementations are both slower than the portable MMM version. The algorithm used by them requires transposition of the block, which is very inefficient. The best performance is from the MMM-Opt version, which is 4% faster than the portable one. The portable L_1-Distance example is slower than the reference because it has an extra operation to compute two partial results in the `sad` macros. The MMM-Opt version uses four partial results, and obtains the same performance as the reference version. Overall, the execution speeds of the MMM versions are within 12% of the best implementations. The instruction counts of the examples are shown in Table 6.15:

Table 6.15
Instruction counts on AltiVec

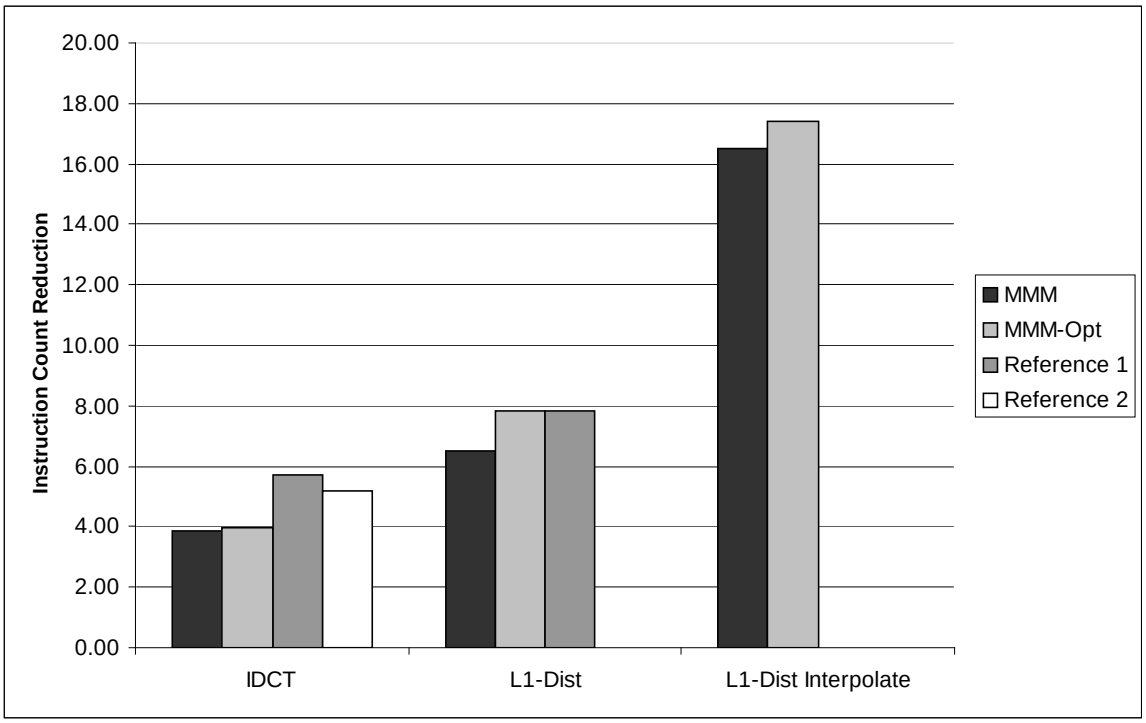
Version	8x8 IDCT	16x16 L_1-Distance	16x16 L_1-Distance with Interpolation
MMM	372	225	308
MMM-Opt	365	186	292
Reference1	253	186	
Reference2	279		
Scalar	1446	1461	5080

Figure 6.7 and Table 6.16 show the reduction in the instruction counts for the programs without shortcuts on AltiVec. We observe than the reduction factors are comparable to the speedups obtained for these programs. The reference IDCT examples use 30% less instructions, but are actually slower than the portable MMM version, because they have more memory accesses.

Table 6.16
Reduction in instruction counts on AltiVec

Version	8x8 IDCT	16x16 L_1-Distance	16x16 L_1-Distance with Interpolation
MMM	3.89	6.49	16.49
MMM-Opt	3.96	7.85	17.40
Reference1	5.72	7.85	
Reference2	5.18		

Figure 6.7
Reduction in instruction counts on AltiVec



6.5 Summary

This chapter presented the performance measurements of the example programs and reference implementations. For each program, it presented the execution time and the speedup with respect to the scalar version. Similary, for examples without shortcut paths it showed the instruction counts and the reduction factor with respect to the scalar versions.

It was seen that for all but one case, the performance of the portable MMM versions is within 12% of the best known implementations. The only case in which the performance difference is more is the IDCT example for TriMedia, which is 26% slower than the reference version. The design in the reference version is more suitable to TriMedia's short register lengths. Even so, the portable IDCT still provides a significant speedup of 2.72 over the scalar version on

TriMedia. On the other architectures, the IDCT obtain speedups factors close to 5. For some of the other examples the speedup is even higher, up to a factor of 15 for L_1-Distance with interpolation on AltiVec. Instruction counts are reduced by factors of up to 26, like in the case of L_1-Distance with interpolation on SSE2. For SSE2 the portable programs perform better than any other version, including the assembly versions by Intel.

Chapter 7

CONCLUSIONS AND FUTURE WORK

In this dissertation I have presented a simple, yet powerful method to write portable optimized code. The results presented prove that using MMM, complex multimedia programs can be portable and optimized at the same time. The example programs written in MMM were compiled into very diverse multimedia architectures, and obtained speedups comparable to the best available hand-optimized versions. There is a natural tradeoff between performance and portability; it is expected that some performance will be lost by using only portable instructions. The results presented show that with MMM this loss is within 12% for a given algorithm. Beyond this, higher performance can only be obtained by using completely different algorithms for each target.

This is the first method to address the portability of arbitrary, complex optimized programs that use complex partitioned instructions. Other portability methods cannot provide the same flexibility and performance as MMM. Parallelizing compilers are only efficient on simple program structures, optimized kernel libraries and automatic code generators are limited to specific applications, and existing data-parallel languages cannot express complex partitioned operations.

Four very distinct target architectures were chosen to prove the method: AltiVec, MMX+SSE, SSE2 and TriMedia TM1300. The register lengths vary from 32 bits to 128-bits. The alignment requirements are unique on each of them. Regarding the instruction sets and

programming styles, only SSE and SSE2 are alike; the other architectures are completely different. Very few research efforts have been able to generate optimized code for such a wide variety of architectures.

The method presented can be applied to any group of architectures, as long as they have enough in common. Multimedia instruction sets are particularly challenging because they combine long partitioned registers with complex instructions. But the method can also be applied to other types of architectures, like DSP instruction sets.

MMM has been demonstrated to be effective for writing portable optimized multimedia applications. The same method can be applied to other fields, like signal processing and scientific applications. Any optimized program that uses complex and/or partitioned instructions can be made portable by this method.

7.1 MMM Limitations

Using C pre-processor macros proved to be a simple and flexible way to emulate instructions, but there are a few instructions that cannot be handled in a general way through static macros. This is the case for permutation instructions. It is not possible to have a general permutation macro because not all architectures have general permutation instructions. But certain permutations can be implemented efficiently using a sequence of rearrangment instructions. An interesting research topic would be to create a "permutation compiler" that takes an arbitrary permutation of two vectors (like in AltiVec's permutation instruction), and tries to emulate it using a library of available rearrangment instructions.

Another case where static macros have difficulty is in the emulation of shift-immediate instructions for small partitions, as discussed in Section 4.7. The most efficient emulation technique is to shift the entire vector, and mask-out the run-over bits. The shift amount is known ahead of time, so creating the mask wouldn't be hard, but it does require a dynamic system.

A third example of static macro limitations is in the declaration of arrays of constant vectors. This was observed in Section 4.1. A static macro is not capable of initializing an array of arbitrary size. The best solution for this is to define a language syntax for declaring arrays of vectors, and have a compiler translate it into the style required by each architecture.

There is also the issue of code style when using macros like MMM. For a given operation, a different macro exists for every partition type, which makes the macro names very long. Also, the requirement that the destination of each operation has to be passed as an argument is not very intuitive. A more elegant approach would be to use data-parallel extensions to C with native vector types and overloaded operations. This is posposed in the next section.

7.2 The Next Step: MMC

The next step from MMM is to develop a Multimedia C language, or MMC. It would be an extension to C that supports vector types natively, but only for a limited set of lengths and types, like the virtual vectors supported by MMM. It would not support arbitrary length vectors like other data-parallel languages; using vectors as long as the largest register length in the set of targets is all that is needed in order to make efficient programs. Applications that work on larger data sets can divide it into sections equal to the virtual register length without a loss in performance.

MMC would also be different from other data-parallel languages in that it would not try to map all vector operations into standard language operators. Instead it would use intrinsic functions to access a library of virtual instructions. The virtual instructions for each target would be defined as macros in a library file, much like MMM macros are defined today. The emulation of complex instructions and long registers would be done the same way as in MMM.

Developing MMC requires defining the language syntax for declaration and manipulation of vector variables. Then it needs a source-to-source translator for each target archtecture. The source-to-source translator would take MMC as an input, and use a macro library to generate C code with intrinsics for each target architecture. It might be possible to create a retargetable translator based on the macro libraries. This MMC translator could be complemented with a permutation compiler and an improved shift emulator, as discussed in the previous section.

I believe that the MMC language proposed above would be a practical solution. It would solve the limitations, and preserve the flexibility and proven performance of MMM. With an elegant, flexible infrastructure like MMC it makes sense to use it to write all optimized programs, instead of writing target-specific code. Even if one decides to implement different algorithms for different target architectures, they can still be written in MMC. This would allow them to be ported to future target platforms very easily. If one needs a particular instruction that is not available in the virtual instruction set, it can be added. This way the library of virtual instructions would grow over time, and adapt to new applications.

Appendix A

VIRTUAL INSTRUCTION SET DEFINITION

This appendix contains a complete list of the MMM macros supported for the selected set of target architectures. The instructions grouped by instruction type. For each macro, the tables below show the intrinsics they map to in each architecture, or the instructions involved in the emulation. The *_mm_* prefix was dropped from all MMX, SSE and SSE2 intrinsics to save space.

A.1 Vector Declaration

Table A.1
MMM vector declaration macros

MMM Macro	TriMedia	MMX+SSE	SSE2	AltiVec
DECLARE_I8x16	int	_m64	_m128i	vector char
DECLARE_U8x16	unsigned int	_m64	_m128i	vector unsigned char
DECLARE_I16x8	int	_m64	_m128i	vector short
DECLARE_U16x8	unsigned int	_m64	_m128i	vector unsigned short
DECLARE_I32x4	int	_m64	_m128i	vector int
DECLARE_U32x4	unsigned int	_m64	_m128i	vector unsigned int
DECLARE_F32x4	float	_m128	_m128	vector float
DECLARE_CONST_I16x8x4	short[][]	__declspec (align(16)) short[][]	__declspec (align(16)) short[][]	vector short[]

A.2 Set Instructions

Table A.2
MMM set instructions

MMM Macro	TriMedia	MMX+SSE	SSE2	AltiVec
SET_I8x16 SET_U8x16	Emulate with =, <<, |	set_pi8	set_epi8	=
SET_I16x8 SET_U16x8	Emulate with =, <<, |	set_pi16	set_epi16	=
SET_I32x4 SET_U32x4	=	set_pi32	set_epi32	=
SET_F32x4	=	set_ps	set_ps	=
SET1_I8x16 SET1_U8x16	Emulate with =, <<, |	set1_pi8	set1_epi8	=
SET1_I16x8 SET1_U16x8	Emulate with =, <<, |	set1_pi16	set1_epi16	=
SET1_I32x4 SET1_U32x4	=	set1_pi32	set1_epi32	=
SET1_F32x4	=	set1_ps	set1_ps	=
CLEAR_I8x16 CLEAR_U8x16 CLEAR_I16x8 CLEAR_U16x8 CLEAR_I32x4 CLEAR_U32x4	=	setzero_si64	setzero_si128	=
CLEAR_F32x4	=	setzero_ps	setzero_ps	=
COPY_I8x16 COPY_U8x16 COPY_I16x8 COPY_U16x8 COPY_I32x4 COPY_U32x4 COPY_F32x4	=	=	=	=

A.3 Load and Store Instructions

Aligned instructions are suffixed with *_A_* and unaligned with *_U_*.

Table A.3
MMM load and store instructions

MMM Macro	TriMedia	MMX+SSE	SSE2	AltiVec
LOAD_A_I8x16 LOAD_A_U8x16 LOAD_A_I16x8 LOAD_A_U16x8 LOAD_A_I32x4 LOAD_A_U32x4	v=*p	v=*p	load_si128	vec_ld
LOAD_A_F32x4	v=*p	load_ps	load_ps	vec_ld
STORE_A_I8x16 STORE_A_U8x16 STORE_A_I16x8 STORE_A_U16x8 STORE_A_I32x4 STORE_A_U32x4	*p=v	*p=v	store_si128	vec_st
STORE_A_F32x4	*p=v	store_ps	store_ps	vec_st
PREPARE_LOAD_ ALIGNMENT	Compute shift amounts	Do nothing	Do nothing	vec_lvsl
PREPARE_STORE_ ALIGNMENT	Compute shift amounts	Do nothing	Do nothing	vec_lvsr
LOAD_U_I8x16 LOAD_U_U8x16 LOAD_U_I16x8 LOAD_U_U16x8 LOAD_U_I32x4 LOAD_U_U32x4	Emulate with *p, <<, >>, |	v=*p	loadu_si128	Emulate with vec_ld, vec_perm
LOAD_U_F32x4	Emulate with *p, <<, >>, |	loadu_ps	loadu_ps	Emulate: vec_ld, vec_perm
LOAD_ADJ_I8x16 LOAD_ADJ_U8x16 LOAD_ADJ_I16x8 LOAD_ADJ_U16x8 LOAD_ADJ_I32x4 LOAD_ADJ_U32x4	Emulate with *p, <<, >>, |	v1=*p v2=*(p+1)	loadu_si128	Emulate with vec_ld, vec_perm
LOAD_ADJ_F32x4	Emulate with *p, <<, >>, |	loadu_ps	loadu_ps	Emulate: vec_ld, vec_perm
STORE_U_I8x16 STORE_U_U8x16	Emulate with *p,	*p=v	storeu_si128	Emulate with

STORE_U_I16x8 STORE_U_U16x8 STORE_U_I32x4 STORE_U_U32x4	<<, >>, |, &			vec_st, vec_perm, vec_sel
STORE_U_F32x4	Emulate with *p, <<, >>, |, &	storeu_ps	storeu_ps	Emulate with vec_st, vec_perm, vec_sel
STORE_MASKED_I8x16 STORE_MASKED_U8x16	Emulate with *p, |, &, ~	maskmove_si64	maskmoveu_si128	Emulate: vec_st, vec_sel

A.4 Rearrangement Instructions

The *_H_* suffix stands for high and *_L_* for low in *interleave* instructions. The *_x_* in the *broadcast* macros is the element index that is to be copied.

Table A.4
MMM rearrangement instructions

MMM Macro	TriMedia	MMX+SSE	SSE2	AltiVec
INTERLEAVE_H_I8x16 INTERLEAVE_H_U8x16	MERGEMSB	unpackhi_pi8	unpackhi_epi8	vec_mergeh
INTERLEAVE_H_I16x8 INTERLEAVE_H_U16x8	PACK16MSB	unpackhi_pi16	unpackhi_epi16	vec_mergeh
INTERLEAVE_H_I32x4 INTERLEAVE_H_U32x4	=	unpackhi_pi32	unpackhi_epi32	vec_mergeh
INTERLEAVE_H_F32x4	=	unpackhi_ps	unpackhi_ps	vec_mergeh
INTERLEAVE_L_I8x16 INTERLEAVE_L_U8x16	MERGELSB	unpacklo_pi8	unpacklo_epi8	vec_mergel
INTERLEAVE_L_I16x8 INTERLEAVE_L_U16x8	PACK16LSB	unpacklo_pi16	unpacklo_epi16	vec_mergel
INTERLEAVE_L_I32x4 INTERLEAVE_L_U32x4	=	unpacklo_pi32	unpacklo_epi32	vec_mergel
INTERLEAVE_L_F32x4	=	unpacklo_ps	unpacklo_ps	vec_mergel
BROADCAST_x_I8x16 BROADCAST_x_U8x16	Emulate: =, <<, |	Emulate: lsl, or, shuffle	Emulate: lsl, or, shuffle	vec_splat
BROADCAST_x_I16x8 BROADCAST_x_U16x8	Emulate: =, <<, |	shuffle_pi16	Emulate with shuffle, shufflelo	vec_splat
BROADCAST_x_I32x4 BROADCAST_x_U32x4	=	shuffle_pi16	shuffle_epi32	vec_splat
BROADCAST_x_F32x4	=	shuffle_ps	shuffle_ps	vec_splat
BROADCAST_PAIR_x_ I16x8, U16x8	Same as BROADCAST_{I/U}32x4 but typecasted to {I/U}16x8			
BROADCAST_PAIR_x_ I8x16, U8x16	Same as BROADCAST_{I/U}16x8 but typecasted to {I/U}8x16			
PERMUTE_I16x8_ 02134657	Emulate with =, PACK16MSB, PACK16LSB	shuffle_pi16	shufflelo_epi16	vec_perm
PERMUTE_I16x8_ 01237654	Emulate: =, ROLI	shuffle_pi16	shufflehi_epi16	vec_perm

A.5 Conversion Instructions

The macro names are *CVT_{dst type}_{src type}*, where the source and destination types can be vectors or scalars. This covers *pack* and *extend* operations too; they are conversions between vectors with different partition types. Conversions with truncation have the suffix *_T_*, the ones with saturation have *_S_*, and the ones with unspecified reduction type have *_N_*. The *_H_* suffix stands for high and *_L_* means low in *extend* operations.

Table A.5
MMM conversion instructions

MMM Macro	TriMedia	MMX+SSE	SSE2	AltiVec
CVT_I32x4_I32 CVT_U32x4_U32	=	cvtsi32_si64	cvtsi32_si128	vec_lde, vec_splat
CVT_I32_I32x4 CVT_U32_U32x4	=	cvtsi64_si32	cvtsi128_si32	vec_ste, vec_splat
CVT_F32x4_I32x4	(float)	cvtpi32x2_ps	cvtepi32_ps	vec_ctf
CVT_I32x4_F32x4	(int)	cvtt_ps2pi	cvttps_epi32	vec_cts
PACK_T_I16x8 PACK_T_U16x8	MERGEDUAL16LSB	Emulate with &, packs	Emulate with &, packs	vec_pack
PACK_T_I32x4 PACK_T_U32x4	PACK16LSB	Emulate with &, packs	Emulate with &, packs	vec_pack
PACK_S_I16x8	Emulate with DUALICLIPI, MERGEDUAL16LSB	packs_pi16	packs_epi16	vec_packs
PACK_S_U16x8	Emulate with DUALUCLIPI, MERGEDUAL16LSB	packs_pu16	packus_epi16	vec_packs
PACK_S_I32x4	Emulate with ICLIPI, PACK16LSB	packs_pi32	packs_epi32	vec_packs
PACK_N_I16x8	MERGEDUAL16LSB	packs_pi16	packs_epi16	vec_pack
PACK_N_U16x8	MERGEDUAL16LSB	packs_pu16	packus_epi16	vec_pack
PACK_N_I32x4 PACK_N_U32x4	PACK16LSB	packs_pi32	packs_epi32	vec_pack
EXTEND_H_I8x16	Emulate with MERGEMSB, DUALASR	Emulate with unpackhi, sra	Emulate with unpackhi, sra	vec_unpackh
EXTEND_H_U8x16	MERGEMSB	unpackhi_pi8	unpackhi_epi8	vec_unpackh
EXTEND_H_I16x8	Emulate with PACK16MSB, SEX16	Emulate with unpackhi, srai	Emulate with unpackhi, srai	vec_unpackh

EXTEND_H_U16x8	PACK16MSB	unpackhi_ pi16	unpackhi_ epi16	vec_unpackh
EXTEND_L_I8x16	Emulate: ROLI, MERGEMSB, DUALASR	Emulate with unpacklo, sl, sra	Emulate with unpacklo, sl, sra	vec_unpackl
EXTEND_L_U8x16	Emulate with MERGEMSB, SRLI	unpacklo_pi8	unpacklo_epi8	vec_unpackl
EXTEND_L_I16x8	SEX16	Emulate with unpacklo, sl, sra	Emulate with unpacklo, sl, sra	vec_unpackl
EXTEND_L_U16x8	PACK16LSB	unpacklo_ pi16	unpacklo_ epi16	vec_unpackl

A.6 Bit-wise Logic Instructions

Table A.6
MMM bit-wise logic instructions

MMM Macro	TriMedia	MMX+SSE	SSE2	AltiVec
AND_I8x16, U8x16 AND_I16x8, U16x8 AND_I32x4, U32x4	&	and_si64	and_si128	vec_and
AND_F32x4	&	and_ps	and_ps	vec_and
ANDN_I8x16, U8x16 ANDN_I16x8, U16x8 ANDN_I32x4, U32x4	BITANDINV	andnot_si64	andnot_si128	vec_andc
ANDN_F32x4	BITANDINV	andnot_ps	andnot_ps	vec_andc
OR_I8x16, U8x16 OR_I16x8, U16x8 OR_I32x4, U32x4	|	or_si64	or_si128	vec_or
OR_F32x4	|	or_ps	or_ps	vec_or
XOR_I8x16, U8x16 XOR_I16x8, U16x8 XOR_I32x4, U32x4	^	xor_si64	xor_si128	vec_xor
XOR_F32x4	^	xor_ps	xor_ps	vec_xor
SEL_I8x16, U8x16 SEL_I16x8, U16x8 SEL_I32x4, U32x4	Emulate with &, |, BITANDINV	Emulate with &, |, andnot	Emulate with &, |, andnot	vec_sel
SEL_F32x4	Emulate with &, |, BITANDINV	Emulate with &, |, andnot	Emulate with &, |, andnot	vec_sel

A.7 Shift Instructions

Macros with the *_I_* suffix take immediate shift amounts.

Table A.7
MMM shift instructions

MMM Macro	TriMedia	MMX+SSE	SSE2	AltiVec
SLL_I8x16 SLL_U8x16	Emulate with <<, &, |	Emulate with sll, and, or	Emulate with sll, and, or	vec_sl
SLL_I16x8 SLL_U16x8	Emulate with <<, &, |	sll_pi16	sll_epi16	vec_sl
SLL_I32x4 SLL_U32x4	<<	sll_pi32	sll_epi32	vec_sl
SLL_I_I8x16 SLL_I_U8x16	Emulate with LSLI, &, |	Emulate with slli, and, or	Emulate with slli, and, or	vec_sl(vec_splat())
SLL_I_I16x8 SLL_I_U16x8	Emulate with LSLI, &, |	slli_pi16	slli_epi16	vec_sl(vec_splat())
SLL_I_I32x4 SLL_I_U32x4	LSLI	slli_pi32	slli_epi32	vec_sl(vec_splat())
SRL_I8x16 SRL_U8x16	Emulate with >>, &, |	Emulate with srl, and, or	Emulate with srl, and, or	vec_sr
SRL_I16x8 SRL_U16x8	Emulate with >>, &, |	srl_pi16	srl_epi16	vec_sr
SRL_I32x4 SRL_U32x4	<<	srl_pi32	srl_epi32	vec_sr
SRL_I_I8x16 SRL_I_U8x16	Emulate with LSRI, &, |	Emulate with srli, and, or	Emulate with srli, and, or	vec_sr(vec_splat())
SRL_I_I16x8 SRL_I_U16x8	Emulate with LSRI, &, |	srli_pi16	srli_epi16	vec_sr(vec_splat())
SRL_I_I32x4 SRL_I_U32x4	LSRI	srli_pi32	srli_epi32	vec_sr(vec_splat())
SRA_I8x16 SRA_U8x16	Emulate with DUALASR, &, |	Emulate with sra, and, or	Emulate with sra, and, or	vec_sra
SRA_I16x8 SRA_U16x8	DUALASR	sra_pi16	sra_epi16	vec_sra
SRA_I32x4 SRA_U32x4	>>	sra_pi32	sra_epi32	vec_sra
SRA_I_I8x16 SRA_I_U8x16	Emulate with DUALASR, &, |	Emulate with srai, and, or	Emulate with srai, and, or	vec_sra(vec_splat())
SRA_I_I16x8 SRA_I_U16x8	DUALASR	srai_pi16	srai_epi16	vec_sra(vec_splat())
SRA_I_I32x4 SRA_I_U32x4	ASRI	srai_pi32	srai_epi32	vec_sra(vec_splat())

ROL_I8x16 ROL_U8x16	Emualte with ROL, >>, &, |	Emulate with sll, srl, &, |	Emulate with sll, srl, &, |	vec_rl
ROL_I16x8 ROL_U16x8	Emualte with ROL, &, |	Emulate with sll, srl, &, |	Emulate with sll, srl, &, |	vec_rl
ROL_I32x4 ROL_U32x4	ROL	Emulate with sll, srl, &, |	Emulate with sll, srl, &, |	vec_rl
ROL_I_I8x16 ROL_I_U8x16	Emualte with ROLI, &, |	Emulate with slli, srli, &, |	Emulate with slli, srli, &, |	vec_rl(vec_splat())
ROL_I_I16x8 ROL_I_U16x8	Emualte with ROLI, &, |	Emulate with slli, srli, &, |	Emulate with slli, srli, &, |	vec_rl(vec_splat())
ROL_I_I32x4 ROL_I_U32x4	ROLI	Emulate with slli, srli, &, |	Emulate with slli, srli, &, |	vec_rl(vec_splat())

A.8 Floating-Point Arithmetic Instructions

Table A.8
MMM floating-point arithmetic instructions

MMM Macro	TriMedia	MMX+SSE	SSE2	AltiVec
ADD_F32x4	+	add_ps	add_ps	vec_add
SUB_F32x4	-	sub_ps	sub_ps	vec_sub
MULT_F32x4	*	mul_ps	mul_ps	vec_madd
MULT_ADD_F32x4	Emulate with *, +	Emulate with mul, +	Emulate with mul, +	vec_madd
DIV_F32x4	/	div_ps	div_ps	Emulate with vec_madd, vec_re
MIN_F32x4	FMIN	min_ps	min_ps	vec_min
MAX_F32x4	FMAX	max_ps	max_ps	vec_max
SQRT_F32x4	FSQRT	sqrt_ps	sqrt_ps	Emulate with vec_rsqrte, vec_re
REC_F32x4	/	rcp_ps	rcp_ps	vec_re
RSQRT_F32x4	Emulate with /, FSQRT	rsqrt_ps	rsqrt_ps	vec_rsqrte

A.9 Integer Arithmetic Instructions

Operations with modulo handling of overflow append *_M_*; those with saturation append *_S_*, and unspecified behavior under overflow append *_N_*. Low and high multiplications are identified with *_L_* and *_H_* respectively.

Table A.9
MMM integer arithmetic instructions

MMM Macro	TriMedia	MMX+SSE	SSE2	AltiVec
ADD_M_I8x16 ADD_M_U8x16	Emulate with +, &, ^	add_pi8	add_epi8	vec_add
ADD_M_I16x8 ADD_M_U16x8	Emulate with +, &, ^	add_pi16	add_epi16	vec_add
ADD_M_I32x4 ADD_M_U32x4	+	add_pi32	add_epi32	vec_add
ADD_S_I8x16	Emulate with +, DUALICLIPI, >>, <<, &, |	adds_pi8	adds_epi8	vec_adds
ADD_S_U8xx6	Emulate with +, DUALUCLIPI, >>, <<, &, |	adds_pu8	adds_epu8	vec_adds
ADD_S_I16x8	DSPIDUALADD	adds_pi16	adds_epi16	vec_adds
ADD_S_U16x8	Emulate with DSPUADD, <<, >>, &, |	adds_pu16	adds_epu16	vec_adds
ADD_N_I8x16 ADD_N_U8x16	+	add_pi8	add_epi8	vec_add
ADD_N_I16x8 ADD_N_U16x8	+	add_pi16	add_epi16	vec_add
ADD_N_I32x4 ADD_N_U32x4	+	add_pi32	add_epi32	vec_add
SUB_M_I8x16 SUB_M_U8x16	Emulate with -, &, ^	sub_pi8	sub_epi8	vec_sub
SUB_M_I16x8 SUB_M_U16x8	Emulate with -, &, ^	sub_pi16	sub_epi16	vec_sub
SUB_M_I32x4 SUB_M_U32x4	-	sub_pi32	sub_epi32	vec_sub
SUB_S_I8x16	Emulate with -, DUALICLIPI, >>, <<, &, |	subs_pi8	subs_epi8	vec_subs
SUB_S_U8xx6	Emulate with -, DUALUCLIPI, >>, <<, &, |	subs_pu8	subs_epu8	vec_subs
SUB_S_I16x8	DSPIDUALSUB	subs_pi16	subs_epi16	vec_subs
SUB_S_U16x8	Emulate with DSPUSUB, <<, >>, &, |	subs_pu16	subs_epu16	vec_subs
SUB_N_I8x16 SUB_N_U8x16	Emulate with -, &, ^	sub_pi8	sub_epi8	vec_sub
SUB_N_I16x8 SUB_N_U16x8	DSPIDUALSUB	sub_pi16	sub_epi16	vec_sub
SUB_N_I32x4 SUB_N_U32x4	-	sub_pi32	sub_epi32	vec_sub
MULT_L_I16x8	Emulate with	mullo_pi16	mullo_epi16	vec_mladd

	*, >>, <<, &, |			
MULT_L_ADD_M_I16x8	Emulate with *, >>, <<, &, |, +, ^	Emulate with mullo, add	Emulate with mullo, add	vec_mladd
MULT_L_ADD_N_I16x8	Emulate with *, >>, <<, &, |, DSPIDUALADD	Emulate with mullo, add	Emulate with mullo, add	vec_mladd
MULT_H_I16x8	Emulate with *, IMULM, &, PACK16MSB	mulhi_pi16	mulhi_epi16	vec_madds
MULT_H_ADD_S_I16x8	Emulate with *, IMULM, &, PACK16MSB, DSPIDUALADD	Emulate with mulhi, adds	Emulate with mulhi, adds	vec_madds
MULT_ADDPAIRS_I16x8	ifir16	madd_pi16	madd_epi16	vec_msum
MULT_ADDPAIRS_ADD_M_I16x8	Emulate with IFIR16, &, ^	Emulate with madd, add	Emulate with madd, add	vec_msum
MULT_ADDPAIRS_ADD_S_I16x8	Emulate with IFIR16, DSPIDUALADD	Emulate with madd, adds	Emulate with madd, adds	vec_msums
MULT_ADDPAIRS_ADD_N_I16x8	Emulate with IFIR16, DSPIDUALADD	Emulate with madd, add	Emulate with madd, add	vec_msum
AVG_U8x16	QUADAVG	avg_pu8	avg_epu8	vec_avg
AVG_U16x8	Emulate with &, +, >>, |	avg_pu16	avg_epu16	vec_avg
MIN_U8x16	QUADUMIN	min_pu8	min_epu8	vec_min
MIN_I16x8	Emulate with IMIN, &, |	min_pi16	min_epi16	vec_min
MAX_U8x16	QUADUMAX	max_pu8	max_epu8	vec_max
MAX_I16x8	Emulate with IMAX, &, |	max_pi16	max_epi16	vec_max
CLIP_I16x8	DUALICLIPI	Emulate with min	Emulate with min	Emulate with vec_min, vec_splat
SAD2_U8x16	UME8UU	sad_pu8	sad_epu8	Emulate with sum (max-min)
SAD2_ADD_M_U8x16	Emulate with UME8UU, +	Emulate with sad, add	Emulate with sad, add	Emulate with sum (max-min)
SUM2_U32x4	+	Emulate with +, _m_to_int	Emulate with add_epi32, srli_si128	vec_sums

A.10 Comparison Instructions

The suffix *EQ* stands for equal, *NEQ* for not equal, *GT* for greater than, *GTE* for greater than or equal, *LT* for less than, and *LTE* for less than or equal.

Table A.10
MMM comparison instructions

MMM Macro	TriMedia	MMX+SSE	SSE2	AltiVec
CMP_EQ_I8x16 CMP_EQ_U8x16	Emulate with MUX, ==, &, |	cmpeq_pi8	cmpeq_epi8	vec_cmpeq
CMP_EQ_I16x8 CMP_EQ_U16x8	Emulate with MUX, ==, &, |	cmpeq_pi16	cmpeq_epi16	vec_cmpeq
CMP_EQ_I32x4 CMP_EQ_U32x4	Emulate with MUX, ==	cmpeq_pi32	cmpeq_epi32	vec_cmpeq
CMP_EQ_F32x4	Emulate with MUX, ==	cmpeq_ps	cmpeq_ps	vec_cmpeq
CMP_GT_I8x16 CMP_GT_U8x16	Emulate with MUX, >, &, |	cmpgt_pi8	cmpgt_epi8	vec_cmpgt
CMP_GT_I16x8 CMP_GT_U16x8	Emulate with MUX, >, &, |	cmpgt_pi16	cmpgt_epi16	vec_cmpgt
CMP_GT_I32x4 CMP_GT_U32x4	Emulate with MUX, >	cmpgt_pi32	cmpgt_epi32	vec_cmpgt
CMP_GT_F32x4	Emulate with MUX, >	cmpgt_ps	cmpgt_ps	vec_cmpgt
CMP_GTE_F32x4	Emulate with MUX, >=	cmpge_ps	cmpge_ps	vec_cmpge
CMP_LT_I8x16 CMP_LT_U8x16	Emulate with MUX, <, &, |	Emulate with cmpgt, andnot	cmplt_epi8	vec_cmplt
CMP_LT_I16x8 CMP_LT_U16x8	Emulate with MUX, <, &, |	Emulate with cmpgt, andnot	cmplt_epi16	vec_cmplt
CMP_LT_I32x4 CMP_LT_U32x4	Emulate with MUX, <	Emulate with cmpgt, andnot	cmplt_epi32	vec_cmplt
CMP_LT_F32x4	Emulate with MUX, <	cmplt_ps	cmplt_ps	vec_cmplt
CMP_LTE_F32x4	Emulate with MUX, <=	cmple_ps	cmple_ps	vec_cmple
CMP_NEQ_F32x4	Emulate with MUX, !=	cmpneq_ps	cmpneq_ps	vec_andc, vec_cmpeq

Appendix B

MMM LIBRARY IMPLEMENTATIONS

The sections below show the actual implementation of the MMM macro libraries for the four different target architectures. The libraries implement the portion of the virtual instruction set that is used by the example programs only.

B.1 TriMedia TM1300

```
/************************************************************
* mmm_tm.h
*
*    This file includes Multi-Media Macro library definitions
*    for the TriMedia TM1300 architecture.
*
*    This library was developed by Juan Carlos Rojas
*    as part of his PhD research at Northeastern University.
*
************************************************************/

#ifndef __MMM_TM__
#define __MMM_TM__

#include "custom_defs.h"

/*
** Precise Basic Types
*/

#define INT8    char
#define INT16   short
#define INT32   int
#define UINT8   unsigned char
#define UINT16  unsigned short
#define UINT32  unsigned int

/*
** Vector Declarations
*/

#define DECLARE_I16x8(var) \
    int var##_0; \
    int var##_1; \
    int var##_2; \
    int var##_3;
```

```
#define DECLARE_U8x16(var) \
    unsigned int var##_0; \
    unsigned int var##_1; \
    unsigned int var##_2; \
    unsigned int var##_3;

#define DECLARE_I32x4(var) \
    int var##_0; \
    int var##_1; \
    int var##_2; \
    int var##_3;

#define DECLARE_U32x4(var) \
    unsigned int var##_0; \
    unsigned int var##_1; \
    unsigned int var##_2; \
    unsigned int var##_3;


#define DECLARE_CONST_I16x8x4(var, c11, c12, c13, c14, c15, c16, c17, c18, \
                                   c21, c22, c23, c24, c25, c26, c27, c28, \
                                   c31, c32, c33, c34, c35, c36, c37, c38, \
                                   c41, c42, c43, c44, c45, c46, c47, c48) \
    INT16 var[4][8] = {c11, c12, c13, c14, c15, c16, c17, c18, \
                       c21, c22, c23, c24, c25, c26, c27, c28, \
                       c31, c32, c33, c34, c35, c36, c37, c38, \
                       c41, c42, c43, c44, c45, c46, c47, c48};

/*
** Set Instructions
*/

#define SET1_I16x8(var, c) \
    var##_0 = var##_1 = var##_2 = var##_3 = (c << 16) | c;

#define SET1_I32x4(var, c) \
    var##_0 = var##_1 = var##_2 = var##_3 = c;

#define CLEAR_U32x4(var) \
    var##_0 = var##_1 = var##_2 = var##_3 = 0;

#define COPY_U8x16(dst, src) \
    dst##_0 = src##_0; \
    dst##_1 = src##_1; \
    dst##_2 = src##_2; \
    dst##_3 = src##_3;

/*
** Load and Store Instructions
*/

#define LOAD_A_I16x8(var, ptr) \
    var##_0 = *((int *) (ptr)); \
    var##_1 = *(((int *) (ptr))+1); \
    var##_2 = *(((int *) (ptr))+2); \
    var##_3 = *(((int *) (ptr))+3);
```

```
#define LOAD_A_U8x16(var, ptr) \
    var##_0 = *((int *) (ptr)); \
    var##_1 = *(((int *) (ptr))+1); \
    var##_2 = *(((int *) (ptr))+2); \
    var##_3 = *(((int *) (ptr))+3);

#define STORE_A_I16x8(ptr, var) \
    *((int *) (ptr)) = var##_0; \
    *(((int *) (ptr))+1) = var##_1; \
    *(((int *) (ptr))+2) = var##_2; \
    *(((int *) (ptr))+3) = var##_3;

/* Static re-alignment values */
static int mmm_tm_shift_left_1;
static int mmm_tm_shift_right_1;
static int mmm_tm_shift_left_2;
static int mmm_tm_shift_right_2;

#define PREPARE_LOAD_ALIGNMENT(index, ptr) \
    mmm_tm_shift_right_##index = (((int) (ptr)) & 0x3)<<3; \
    mmm_tm_shift_left_##index = 32 - mmm_tm_shift_right_##index;

#define LOAD_U_U8x16(var, ptr, index) \
    var##_0 = ((*(((int *) (ptr))+1)) << mmm_tm_shift_left_##index) | \
              (*(((unsigned int *) (ptr))) >> mmm_tm_shift_right_##index); \
    var##_1 = ((*(((int *) (ptr))+2)) << mmm_tm_shift_left_##index) | \
              (*(((unsigned int *) (ptr))+1) >> mmm_tm_shift_right_##index); \
    var##_2 = ((*(((int *) (ptr))+3)) << mmm_tm_shift_left_##index) | \
              (*(((unsigned int *) (ptr))+2) >> mmm_tm_shift_right_##index); \
    var##_3 = ((*(((int *) (ptr))+4)) << mmm_tm_shift_left_##index) | \
              (*(((unsigned int *) (ptr))+3) >> mmm_tm_shift_right_##index);

#define LOAD_ADJ_U8x16(var1, var2, ptr, index1, index2) \
    var1##_0 = ((*(((int *) (ptr))+1)) << mmm_tm_shift_left_##index1) | \
               (*(((unsigned int *) (ptr))) >> mmm_tm_shift_right_##index1); \
    var1##_1 = ((*(((int *) (ptr))+2)) << mmm_tm_shift_left_##index1) | \
               (*(((unsigned int *) (ptr))+1) >> mmm_tm_shift_right_##index1); \
    var1##_2 = ((*(((int *) (ptr))+3)) << mmm_tm_shift_left_##index1) | \
               (*(((unsigned int *) (ptr))+2) >> mmm_tm_shift_right_##index1); \
    var1##_3 = ((*(((int *) (ptr))+4)) << mmm_tm_shift_left_##index1) | \
               (*(((unsigned int *) (ptr))+3) >> mmm_tm_shift_right_##index1); \
    var2##_0 = ((*(((int *) (ptr))+1)) << mmm_tm_shift_left_##index2) | \
               (*(((unsigned int *) (ptr))) >> mmm_tm_shift_right_##index2); \
    var2##_1 = ((*(((int *) (ptr))+2)) << mmm_tm_shift_left_##index2) | \
               (*(((unsigned int *) (ptr))+1) >> mmm_tm_shift_right_##index2); \
    var2##_2 = ((*(((int *) (ptr))+3)) << mmm_tm_shift_left_##index2) | \
               (*(((unsigned int *) (ptr))+2) >> mmm_tm_shift_right_##index2); \
    var2##_3 = ((*(((int *) (ptr))+4)) << mmm_tm_shift_left_##index2) | \
               (*(((unsigned int *) (ptr))+3) >> mmm_tm_shift_right_##index2);

/*
** Rearrangement Instructions
*/

#define BROADCAST_PAIR_0_I16x8(dst, src) \
    dst##_0 = src##_0; \
    dst##_1 = src##_0; \
    dst##_2 = src##_0; \
    dst##_3 = src##_0;

#define BROADCAST_PAIR_1_I16x8(dst, src) \
    dst##_0 = src##_1; \
```

```
    dst##_1 = src##_1; \
    dst##_2 = src##_1; \
    dst##_3 = src##_1;

#define BROADCAST_PAIR_2_I16x8(dst, src) \
    dst##_0 = src##_2; \
    dst##_1 = src##_2; \
    dst##_2 = src##_2; \
    dst##_3 = src##_2;

#define BROADCAST_PAIR_3_I16x8(dst, src) \
    dst##_0 = src##_3; \
    dst##_1 = src##_3; \
    dst##_2 = src##_3; \
    dst##_3 = src##_3;

#define PERMUTE_I16x8_02134657(dst, src) \
{ \
    int tmp; \
    tmp = PACK16LSB(src##_1, src##_0); \
    dst##_1 = PACK16MSB(src##_1, src##_0); \
    dst##_0 = tmp; \
    tmp = PACK16LSB(src##_3, src##_2); \
    dst##_3 = PACK16MSB(src##_3, src##_2); \
    dst##_2 = tmp; \
}

#define PERMUTE_I16x8_01237654(dst, src) \
{ \
    int tmp; \
    dst##_0 = src##_0; \
    dst##_1 = src##_1; \
    tmp = ROLI(16, src##_3); \
    dst##_3 = ROLI(16, src##_2); \
    dst##_2 = tmp; \
}

/*
** Conversion Instructions
*/

#define PACK_N_I32x4(dst, src1, src2) \
    dst##_0 = PACK16LSB(src1##_1, src1##_0); \
    dst##_1 = PACK16LSB(src1##_3, src1##_2); \
    dst##_2 = PACK16LSB(src2##_1, src2##_0); \
    dst##_3 = PACK16LSB(src2##_3, src2##_2);

/*
** Shift Instructions
*/

#define SRA_I_I16x8(dst, src, amount) \
    dst##_0 = DUALASR(src##_0, amount); \
    dst##_1 = DUALASR(src##_1, amount); \
    dst##_2 = DUALASR(src##_2, amount); \
    dst##_3 = DUALASR(src##_3, amount);
```

```
#define SRA_I_I32x4(dst, src, amount) \
    dst##_0 = ASRI(amount, src##_0); \
    dst##_1 = ASRI(amount, src##_1); \
    dst##_2 = ASRI(amount, src##_2); \
    dst##_3 = ASRI(amount, src##_3);

/*
** Integer Arithmetic Instructions
*/

#define ADD_N_I16x8(dst, src1, src2) \
    dst##_0 = DSPIDUALADD(src1##_0, src2##_0); \
    dst##_1 = DSPIDUALADD(src1##_1, src2##_1); \
    dst##_2 = DSPIDUALADD(src1##_2, src2##_2); \
    dst##_3 = DSPIDUALADD(src1##_3, src2##_3);

#define ADD_N_I32x4(dst, src1, src2) \
    dst##_0 = src1##_0 + src2##_0; \
    dst##_1 = src1##_1 + src2##_1; \
    dst##_2 = src1##_2 + src2##_2; \
    dst##_3 = src1##_3 + src2##_3;

#define SUB_N_I16x8(dst, src1, src2) \
    dst##_0 = DSPIDUALSUB(src1##_0, src2##_0); \
    dst##_1 = DSPIDUALSUB(src1##_1, src2##_1); \
    dst##_2 = DSPIDUALSUB(src1##_2, src2##_2); \
    dst##_3 = DSPIDUALSUB(src1##_3, src2##_3);

#define SUB_N_I32x4(dst, src1, src2) \
    dst##_0 = src1##_0 - src2##_0; \
    dst##_1 = src1##_1 - src2##_1; \
    dst##_2 = src1##_2 - src2##_2; \
    dst##_3 = src1##_3 - src2##_3;

#define MULT_H_I16x8(dst, src1, src2) \
    dst##_0 = PACK16MSB(IMULM(src1##_0 & 0xFFFF0000, src2##_0 & 0xFFFF0000), \
                        SEX16(src1##_0) * SEX16(src2##_0)); \
    dst##_1 = PACK16MSB(IMULM(src1##_1 & 0xFFFF0000, src2##_1 & 0xFFFF0000), \
                        SEX16(src1##_1) * SEX16(src2##_1)); \
    dst##_2 = PACK16MSB(IMULM(src1##_2 & 0xFFFF0000, src2##_2 & 0xFFFF0000), \
                        SEX16(src1##_2) * SEX16(src2##_2)); \
    dst##_3 = PACK16MSB(IMULM(src1##_3 & 0xFFFF0000, src2##_3 & 0xFFFF0000), \
                        SEX16(src1##_3) * SEX16(src2##_3));

#define MULT_H_ADD_N_I16x8(dst, src1, src2, src3) \
    dst##_0 = DSPIDUALADD(PACK16MSB(IMULM(src1##_0 & 0xFFFF0000, \
                                          src2##_0 & 0xFFFF0000), \
                            SEX16(src1##_0) * SEX16(src2##_0)), src3##_0); \
    dst##_1 = DSPIDUALADD(PACK16MSB(IMULM(src1##_1 & 0xFFFF0000, \
                                          src2##_1 & 0xFFFF0000), \
                            SEX16(src1##_1) * SEX16(src2##_1)), src3##_1); \
    dst##_2 = DSPIDUALADD(PACK16MSB(IMULM(src1##_2 & 0xFFFF0000, \
                                          src2##_2 & 0xFFFF0000), \
                            SEX16(src1##_2) * SEX16(src2##_2)), src3##_2); \
    dst##_3 = DSPIDUALADD(PACK16MSB(IMULM(src1##_3 & 0xFFFF0000, \
                                          src2##_3 & 0xFFFF0000), \
                            SEX16(src1##_3) * SEX16(src2##_3)), src3##_3);
```

```
#define MULT_ADDPAIRS_I16x8(dst, src1, src2) \
    dst##_0 = IFIR16(src1##_0, src2##_0); \
    dst##_1 = IFIR16(src1##_1, src2##_1); \
    dst##_2 = IFIR16(src1##_2, src2##_2); \
    dst##_3 = IFIR16(src1##_3, src2##_3);

#define MULT_ADDPAIRS_ADD_N_I16x8(dst, src1, src2, src3) \
    dst##_0 = IFIR16(src1##_0, src2##_0) + src3##_0; \
    dst##_1 = IFIR16(src1##_1, src2##_1) + src3##_1; \
    dst##_2 = IFIR16(src1##_2, src2##_2) + src3##_2; \
    dst##_3 = IFIR16(src1##_3, src2##_3) + src3##_3;

#define AVG_U8x16(dst, src1, src2) \
    dst##_0 = QUADAVG(src1##_0, src2##_0); \
    dst##_1 = QUADAVG(src1##_1, src2##_1); \
    dst##_2 = QUADAVG(src1##_2, src2##_2); \
    dst##_3 = QUADAVG(src1##_3, src2##_3);

#define SAD2_ADD_M_U8x16(dst, src1, src2, src3) \
    dst##_0 = UME8UU(src1##_0, src2##_0) + UME8UU(src1##_1, src2##_1) + src3##_0;
\
    dst##_2 = UME8UU(src1##_2, src2##_2) + UME8UU(src1##_3, src2##_3) + src3##_2;

#define SUM2_U32x4(dst, src) \
    dst = src##_0 + src##_2;


/*
** Miscelaneous
*/
#define MALLOC_ALIGN16(size) _cache_malloc(size)

#define END_OPTIMIZED()

#endif /* __MMM_TM__ */
```

B.2 MMX + SSE

```
/*********************************************************
* mmm_sse.h
*
*    This file includes Multi-Media Macro library definitions
*    for Intel MMX & SSE instruction sets.
*
*    This library was developed by Juan Carlos Rojas
*    as part of his PhD research at Northeastern University.
*
*********************************************************/

#ifndef __MMM_SSE__
#define __MMM_SSE__

#include <xmmintrin.h>

/*
** Precise Basic Types
*/

#define INT8    char
#define INT16   short
#define INT32   long
#define UINT8   unsigned char
#define UINT16  unsigned short
#define UINT32  unsigned long

/*
** Vector Declarations
*/

#define DECLARE_I16x8(var) \
    __m64 var##_0;         \
    __m64 var##_1;

#define DECLARE_U8x16(var) \
    __m64 var##_0;         \
    __m64 var##_1;

#define DECLARE_I32x4(var) \
    __m64 var##_0;         \
    __m64 var##_1;

#define DECLARE_U32x4(var) \
    __m64 var##_0;         \
    __m64 var##_1;
```

```
#define DECLARE_CONST_I16x8x4(var, c11, c12, c13, c14, c15, c16, c17, c18, \
                                   c21, c22, c23, c24, c25, c26, c27, c28, \
                                   c31, c32, c33, c34, c35, c36, c37, c38, \
                                   c41, c42, c43, c44, c45, c46, c47, c48) \
    __declspec(align(16)) INT16 var[4][8] = {c11, c12, c13, c14, c15, c16, c17, c18, \
                                             c21, c22, c23, c24, c25, c26, c27, c28, \
                                             c31, c32, c33, c34, c35, c36, c37, c38, \
                                             c41, c42, c43, c44, c45, c46, c47, c48};

/*
** Set Instructions
*/

#define SET1_I16x8(var, c) \
    var##_0 = _mm_set1_pi16(c); \
    var##_1 = var##_0;

#define SET1_I32x4(var, c) \
    var##_0 = _mm_set1_pi32(c); \
    var##_1 = var##_0;

#define CLEAR_U32x4(var) \
    var##_0 = _mm_setzero_si64(); \
    var##_1 = _mm_setzero_si64();

#define COPY_U8x16(dst, src) \
    dst##_0 = src##_0; \
    dst##_1 = src##_1;

/*
** Load and Store Instructions
*/

#define LOAD_A_I16x8(var, ptr) \
    var##_0 = *((__m64 *) (ptr)); \
    var##_1 = *(((__m64 *) (ptr))+1);

#define LOAD_A_U8x16(var, ptr) \
    var##_0 = *((__m64 *) (ptr)); \
    var##_1 = *(((__m64 *) (ptr))+1);

#define STORE_A_I16x8(ptr, var) \
    *((__m64 *) (ptr)) = var##_0; \
    *(((__m64 *) (ptr))+1) = var##_1;

#define PREPARE_LOAD_ALIGNMENT(index, ptr)

#define LOAD_U_U8x16(var, ptr, index) \
    var##_0 = *((__m64 *) (ptr)); \
    var##_1 = *(((__m64 *) (ptr))+1);

#define LOAD_ADJ_U8x16(var1, var2, ptr, index1, index2) \
    var1##_0 = *((__m64 *) (ptr)); \
    var1##_1 = *(((__m64 *) (ptr))+1); \
    var2##_0 = *((__m64 *) (ptr+1)); \
    var2##_1 = *(((__m64 *) (ptr+1))+1);
```

```
/*
** Rearrangement Instructions
*/

#define BROADCAST_PAIR_0_I16x8(dst, src) \
    dst##_0 = _m_pshufw(src##_0, 0x44); \
    dst##_1 = dst##_0;

#define BROADCAST_PAIR_1_I16x8(dst, src) \
    dst##_0 = _m_pshufw(src##_0, 0xEE); \
    dst##_1 = dst##_0;

#define BROADCAST_PAIR_2_I16x8(dst, src) \
    dst##_0 = _m_pshufw(src##_1, 0x44); \
    dst##_1 = dst##_0;

#define BROADCAST_PAIR_3_I16x8(dst, src) \
    dst##_0 = _m_pshufw(src##_1, 0xEE); \
    dst##_1 = dst##_0;

#define PERMUTE_I16x8_02134657(dst, src) \
    dst##_0 = _m_pshufw(src##_0, 0xD8); \
    dst##_1 = _m_pshufw(src##_1, 0xD8);

#define PERMUTE_I16x8_01237654(dst, src) \
    dst##_0 = src##_0; \
    dst##_1 = _m_pshufw(src##_1, 0x1B);

/*
** Conversion Instructions
*/

#define PACK_N_I32x4(dst, src1, src2) \
    dst##_0 = _m_packssdw(src1##_0, src1##_1); \
    dst##_1 = _m_packssdw(src2##_0, src2##_1);

/*
** Shift Instructions
*/

#define SRA_I_I16x8(dst, src, amount) \
    dst##_0 = _mm_srai_pi16(src##_0, amount); \
    dst##_1 = _mm_srai_pi16(src##_1, amount);

#define SRA_I_I32x4(dst, src, amount) \
    dst##_0 = _mm_srai_pi32(src##_0, amount);\
    dst##_1 = _mm_srai_pi32(src##_1, amount);

/*
** Integer Arithmetic Instructions
*/

#define ADD_N_I16x8(dst, src1, src2) \
    dst##_0 = _mm_add_pi16(src1##_0, src2##_0); \
    dst##_1 = _mm_add_pi16(src1##_1, src2##_1);

#define ADD_N_I32x4(dst, src1, src2) \
    dst##_0 = _mm_add_pi32(src1##_0, src2##_0);\
    dst##_1 = _mm_add_pi32(src1##_1, src2##_1);

#define SUB_N_I16x8(dst, src1, src2) \
    dst##_0 = _mm_sub_pi16(src1##_0, src2##_0); \
```

```
    dst##_1 = _mm_sub_pi16(src1##_1, src2##_1);

#define SUB_N_I32x4(dst, src1, src2) \
    dst##_0 = _mm_sub_pi32(src1##_0, src2##_0);\
    dst##_1 = _mm_sub_pi32(src1##_1, src2##_1);

#define MULT_H_I16x8(dst, src1, src2) \
    dst##_0 = _mm_mulhi_pi16(src1##_0, src2##_0); \
    dst##_1 = _mm_mulhi_pi16(src1##_1, src2##_1);

#define MULT_H_ADD_N_I16x8(dst, src1, src2, src3) \
    dst##_0 = _mm_add_pi16(_mm_mulhi_pi16(src1##_0, src2##_0), src3##_0); \
    dst##_1 = _mm_add_pi16(_mm_mulhi_pi16(src1##_1, src2##_1), src3##_1);

#define MULT_ADDPAIRS_I16x8(dst, src1, src2) \
    dst##_0 = _m_pmaddwd(src1##_0, src2##_0);\
    dst##_1 = _m_pmaddwd(src1##_1, src2##_1);

#define MULT_ADDPAIRS_ADD_N_I16x8(dst, src1, src2, src3) \
    dst##_0 = _mm_add_pi32(src3##_0, _m_pmaddwd(src1##_0, src2##_0));\
    dst##_1 = _mm_add_pi32(src3##_1, _m_pmaddwd(src1##_1, src2##_1));

#define AVG_U8x16(dst, src1, src2) \
    dst##_0 = _m_pavgb(src1##_0, src2##_0); \
    dst##_1 = _m_pavgb(src1##_1, src2##_1);

#define SAD2_ADD_M_U8x16(dst, src1, src2, src3) \
    dst##_0 = _mm_add_pi32(_m_psadbw(src1##_0, src2##_0), src3##_0); \
    dst##_1 = _mm_add_pi32(_m_psadbw(src1##_1, src2##_1), src3##_1);

#define SUM2_U32x4(dst, src) \
    dst = _m_to_int(src##_0) + _m_to_int(src##_1);


/*
** Miscelaneous
*/
#ifdef __INTEL_COMPILER
    #define MALLOC_ALIGN16(size) _mm_malloc(size, 16)
#else
    #define MALLOC_ALIGN16(size) _aligned_malloc(size, 16)
#endif

#define END_OPTIMIZED() \
    _mm_empty();

#endif /* __MMM_SSE__ */
```

B.3 SSE2

```
/*************************************************************
* mmm_sse.h
*
*    This file includes Multi-Media Macro library definitions
*    for Intel MMX & SSE instruction sets.
*
*    This library was developed by Juan Carlos Rojas
*    as part of his PhD research at Northeastern University.
*
*************************************************************/

#ifndef __MMM_SSE2__
#define __MMM_SSE2__

#include <emmintrin.h>

/*
** Precise Basic Types
*/
#define INT8    char
#define INT16   short
#define INT32   long
#define UINT8   unsigned char
#define UINT16  unsigned short
#define UINT32  unsigned long

/*
** Vector Declarations
*/
#define DECLARE_I16x8(var) \
    __m128i var;

#define DECLARE_U8x16(var) \
    __m128i var;

#define DECLARE_I32x4(var) \
    __m128i var;

#define DECLARE_U32x4(var) \
    __m128i var;

#define DECLARE_CONST_I16x8x4(var, c11, c12, c13, c14, c15, c16, c17, c18, \
                                   c21, c22, c23, c24, c25, c26, c27, c28, \
                                   c31, c32, c33, c34, c35, c36, c37, c38, \
                                   c41, c42, c43, c44, c45, c46, c47, c48) \
    __declspec(align(16)) INT16 var[4][8] = {c11, c12, c13, c14, c15, c16, c17, c18, \
                                             c21, c22, c23, c24, c25, c26, c27, c28, \
                                             c31, c32, c33, c34, c35, c36, c37, c38, \
                                             c41, c42, c43, c44, c45, c46, c47, c48};

/*
** Set Instructions
*/

#define SET1_I16x8(var, c) \
    var = _mm_set1_epi16(c);
#define SET1_I32x4(var, c) \
```

```
    var = _mm_set1_epi32(c);

#define CLEAR_U32x4(var) \
    var = _mm_xor_si128(var, var);

#define COPY_U8x16(dst, src) \
    dst = src;

/*
** Load and Store Instructions
*/
#define LOAD_A_I16x8(var, ptr) \
    var = _mm_load_si128((__m128i *) (ptr));

#define LOAD_A_U8x16(var, ptr) \
    var = _mm_load_si128((__m128i *) (ptr));

#define STORE_A_I16x8(ptr, var) \
    _mm_store_si128((__m128i *) (ptr), var);

#define PREPARE_LOAD_ALIGNMENT(index, offset)

#define LOAD_U_U8x16(var, ptr, index) \
    var = _mm_loadu_si128((__m128i *) (ptr));

#define LOAD_ADJ_U8x16(var1, var2, ptr, index1, index2) \
    var1 = _mm_loadu_si128((__m128i *) (ptr)); \
    var2 = _mm_loadu_si128((__m128i *) (ptr + 1));

/*
** Rearrangement Instructions
*/
#define BROADCAST_PAIR_0_I16x8(dst, src) \
    dst = _mm_shuffle_epi32(src, 0x00);

#define BROADCAST_PAIR_1_I16x8(dst, src) \
    dst = _mm_shuffle_epi32(src, 0x55);

#define BROADCAST_PAIR_2_I16x8(dst, src) \
    dst = _mm_shuffle_epi32(src, 0xAA);

#define BROADCAST_PAIR_3_I16x8(dst, src) \
    dst = _mm_shuffle_epi32(src, 0xFF);

#define PERMUTE_I16x8_02134657(dst, src) \
    dst = _mm_shufflelo_epi16(_mm_shufflehi_epi16(src, 0xD8), 0xD8);

#define PERMUTE_I16x8_01237654(dst, src) \
    dst = _mm_shufflehi_epi16(src, 0x1B);

/*
** Conversion Instructions
*/
#define PACK_N_I32x4(dst, src1, src2) \
    dst = _mm_packs_epi32(src1, src2);

/*
** Shift Instructions
*/
#define SRA_I_I16x8(dst, src, amount) \
    dst = _mm_srai_epi16(src, amount);
```

```
#define SRA_I_I32x4(dst, src, amount) \
    dst = _mm_srai_epi32(src, amount);

/*
** Integer Arithmetic Instructions
*/
#define ADD_N_I16x8(dst, src1, src2) \
    dst = _mm_add_epi16(src1, src2);

#define ADD_N_I32x4(dst, src1, src2) \
    dst = _mm_add_epi32(src1, src2);

#define SUB_N_I16x8(dst, src1, src2) \
    dst = _mm_sub_epi16(src1, src2);

#define SUB_N_I32x4(dst, src1, src2) \
    dst = _mm_sub_epi32(src1, src2);

#define MULT_H_I16x8(dst, src1, src2) \
    dst = _mm_mulhi_epi16(src1, src2);

#define MULT_H_ADD_N_I16x8(dst, src1, src2, src3) \
    dst = _mm_add_epi16(_mm_mulhi_epi16(src1, src2), src3);

#define MULT_ADDPAIRS_I16x8(dst, src1, src2) \
    dst = _mm_madd_epi16(src1, src2);

#define MULT_ADDPAIRS_ADD_N_I16x8(dst, src1, src2, src3) \
    dst = _mm_add_epi32(_mm_madd_epi16(src1, src2), src3);

#define AVG_U8x16(dst, src1, src2) \
    dst = _mm_avg_epu8(src1, src2);

#define SAD2_ADD_M_U8x16(dst, src1, src2, src3) \
    dst = _mm_add_epi32(src3, _mm_sad_epu8(src1, src2));

#define SUM2_U32x4(dst, src) \
    dst = _mm_cvtsi128_si32(_mm_add_epi32(src, _mm_srli_si128(src, 8)));

/*
** Miscelaneous
*/
#ifdef __INTEL_COMPILER
    #define MALLOC_ALIGN16(size) _mm_malloc(size, 16)
#else
    #define MALLOC_ALIGN16(size) _aligned_malloc(size, 16)
#endif

#define END_OPTIMIZED()


#endif /* __MMM_SSE2__ */
```

B.4 AltiVec

```
/*************************************************************
* mmm_altivec.h
*
*    This file includes Multi-Media Macro library definitions
*    for the AltiVec instruction set.
*
*    This library was developed by Juan Carlos Rojas
*    as part of his PhD research at Northeastern University.
*
*************************************************************/

#ifndef __MMM_ALTIVEC__
#define __MMM_ALTIVEC__

/*
** Precise Basic Types
*/

/* Precise basic types */
#define INT8    signed char
#define INT16   signed short
#define INT32   signed int
#define UINT8   unsigned char
#define UINT16  unsigned short
#define UINT32  unsigned int

/*
** Vector Declarations
*/

#define DECLARE_I16x8(var) \
    vector INT16 var;

#define DECLARE_U8x16(var) \
    vector UINT8 var;

#define DECLARE_I32x4(var) \
    vector INT32 var;

#define DECLARE_U32x4(var) \
    vector UINT32 var;

#define DECLARE_CONST_I16x8x4(var, c11, c12, c13, c14, c15, c16, c17, c18, \
                                   c21, c22, c23, c24, c25, c26, c27, c28, \
                                   c31, c32, c33, c34, c35, c36, c37, c38, \
                                   c41, c42, c43, c44, c45, c46, c47, c48) \
    vector INT16 var[4] = {(vector INT16) (c11, c12, c13, c14, c15, c16, c17, c18), \
                           (vector INT16) (c21, c22, c23, c24, c25, c26, c27, c28), \
                           (vector INT16) (c31, c32, c33, c34, c35, c36, c37, c38), \
                           (vector INT16) (c41, c42, c43, c44, c45, c46, c47, c48)};
```

```
/*
** Set Instructions
*/

#define SET1_I16x8(dst, c) \
    dst = (vector INT16) (c);

#define SET1_I32x4(dst, c) \
    dst = (vector INT32) (c);

#define CLEAR_U32x4(dst) \
    dst = (vector UINT32)(0);

#define COPY_U8x16(dst, src) \
    dst = src;

/*
** Load and Store Instructions
*/

#define LOAD_A_I16x8(var, ptr) \
    var = vec_ld(0, (vector INT16 *) (ptr));

#define LOAD_A_U8x16(var, ptr) \
    var = vec_ld(0, (vector UINT8 *) (ptr));

#define STORE_A_I16x8(ptr, var) \
    vec_st(var, 0, (vector INT16 *) (ptr));

/* Static re-alignemt vectors */
static vector UINT8 mmm_align_vector1;
static vector UINT8 mmm_align_vector2;

#define PREPARE_LOAD_ALIGNMENT(index, ptr) \
    mmm_align_vector##index = vec_lvsl(0, ptr);

#define LOAD_U_U8x16(var, ptr, index) \
    var = vec_perm(vec_ld(0, (vector UINT8 *) (ptr)), \
                   vec_ld(0, ((vector UINT8 *) (ptr)) +
1),mmm_align_vector##index);

#define LOAD_ADJ_U8x16(var1, var2, ptr, index1, index2) \
    var1 = vec_perm(vec_ld(0, (vector UINT8 *) (ptr)), \
                    vec_ld(0, ((vector UINT8 *) (ptr)) +
1),mmm_align_vector##index1); \
    var2 = vec_perm(vec_ld(0, (vector UINT8 *) (ptr)), \
                    vec_ld(0, ((vector UINT8 *) (ptr)) +
1),mmm_align_vector##index2);
```

```
/*
** Rearrangement Instructions
*/

#define BROADCAST_PAIR_0_I16x8(dst, src) \
    dst = (vector INT16)(vec_splat((vector INT32)(src), 0));

#define BROADCAST_PAIR_1_I16x8(dst, src) \
    dst = (vector INT16)(vec_splat((vector INT32)(src), 1));

#define BROADCAST_PAIR_2_I16x8(dst, src) \
    dst = (vector INT16)(vec_splat((vector INT32)(src), 2));

#define BROADCAST_PAIR_3_I16x8(dst, src) \
    dst = (vector INT16)(vec_splat((vector INT32)(src), 3));

#define PERMUTE_I16x8_02134657(dst, src) \
    dst = vec_perm(src, src, (vector UINT8) \
          (0, 1, 4, 5, 2, 3, 6, 7, 8, 9, 12, 13, 10, 11, 14, 15));

#define PERMUTE_I16x8_01237654(dst, src) \
    dst = vec_perm(src, src, (vector UINT8) \
          (0, 1, 2, 3, 4, 5, 6, 7, 14, 15, 12, 13, 10, 11, 8, 9));

/*
** Conversion Instructions
*/

#define PACK_N_I32x4(dst, src1, src2) \
    dst = vec_pack(src1, src2);

/*
** Shift Instructions
*/

#define SRA_I_I16x8(dst, src, amount) \
    dst = vec_sra(src, (vector UINT16) (amount));

#define SRA_I_I32x4(dst, src, amount) \
    dst = vec_sra(src, (vector UINT32) (amount));

/*
** Integer Arithmetic Instructions
*/

#define ADD_N_I16x8(dst, src1, src2) \
    dst = vec_add(src1, src2);

#define ADD_N_I32x4(dst, src1, src2) \
    dst = vec_add(src1, src2);

#define SUB_N_I16x8(dst, src1, src2) \
    dst = vec_sub(src1, src2);

#define SUB_N_I32x4(dst, src1, src2) \
    dst = vec_sub(src1, src2);

#define MULT_H_I16x8(dst, src1, src2) \
    dst = vec_madds(src1, vec_sra(src2, (vector UINT16) (1)), (vector INT16)
(0));

#define MULT_H_ADD_N_I16x8(dst, src1, src2, src3) \
```

```
    dst = vec_madds(src1, vec_sra(src2, (vector UINT16) (1)), src3);

#define MULT_ADDPAIRS_I16x8(dst, src1, src2) \
    dst = vec_msum(src1, src2, (vector INT32) (0));

#define MULT_ADDPAIRS_ADD_N_I16x8(dst, src1, src2, src3) \
    dst = vec_msum(src1, src2, src3);

#define AVG_U8x16(dst, src1, src2) \
    dst = vec_avg(src1, src2);

#define SAD2_ADD_M_U8x16(dst, src1, src2, src3) \
    dst = (vector UINT32) vec_sum2s((vector INT32) \
          vec_sum4s(vec_sub(vec_max(src1, src2), vec_min(src1, src2)), \
          (vector UINT32)(0)), (vector INT32) src3);

#define SUM2_U32x4(dst, src) \
    vec_ste((vector UINT32) vec_splat(vec_sums((vector INT32) src, \
          (vector INT32)(0)), 3), 0, &dst);

/*
** Miscelaneous
*/
#define MALLOC_ALIGN16(size) malloc(size)

#define END_OPTIMIZED()


#endif /* __MMM_ALTIVEC__ */
```

Appendix C

MMM EXAMPLE PROGRAMS

This appendix includes the source code of the portable example programs written in MMM. Section C.1 is the 8x8 IDCT, C.2 is the 16x16 L_1-Distance, and C.3 is the 16x16 L_1-Distance with interpolation.

C.1 8x8 IDCT

```
/*************************************************************
* idct_mmm.c
*
*   This file includes an implementation of 8x8 Inverse Discrete
*   Cosine Transform using Multi-Media Macro libraries.
*
*   This file in intended to be compiled for
*   any of the following target architectures:
*       - Intel SSE2
*       - Intel MMX + SSE
*       - TriMedia TM1300
*       - AltiVec
*
*   This program, and Multi-Media Macro libraries were developed
*   by Juan Carlos Rojas as part of his PhD research at
*   Northeastern University.
*
*************************************************************/

#ifdef SSE2
    #include "mmm_sse2.h"
#endif
#ifdef SSE
    #include "mmm_sse.h"
#endif
#ifdef TRIMEDIA
    #include "mmm_tm.h"
#endif
#ifdef ALTIVEC
    #include "mmm_altivec.h"
#endif

/* Coefficient constants for horizontal IDCT */
/* They using 15 bits of fractional precision */
#define C1C1 31521 /* Cos(1*pi/16)*Cos(1*pi/16) << 15 */
#define C1C2 29692 /* Cos(1*pi/16)*Cos(2*pi/16) << 15 */
#define C1C3 26722 /* Cos(1*pi/16)*Cos(3*pi/16) << 15 */
```

```
#define C1C4 22725 /* Cos(1*pi/16)*Cos(4*pi/16) << 15 */
#define C1C5 17855 /* Cos(1*pi/16)*Cos(5*pi/16) << 15 */
#define C1C6 12299 /* Cos(1*pi/16)*Cos(6*pi/16) << 15 */
#define C1C7 6270  /* Cos(1*pi/16)*Cos(7*pi/16) << 15 */

#define C2C2 27969 /* Cos(2*pi/16)*Cos(2*pi/16) << 15 */
#define C2C3 25172 /* Cos(2*pi/16)*Cos(3*pi/16) << 15 */
#define C2C4 21407 /* Cos(2*pi/16)*Cos(4*pi/16) << 15 */
#define C2C5 16819 /* Cos(2*pi/16)*Cos(5*pi/16) << 15 */
#define C2C6 11585 /* Cos(2*pi/16)*Cos(6*pi/16) << 15 */
#define C2C7 5906  /* Cos(2*pi/16)*Cos(7*pi/16) << 15 */

#define C3C3 22654 /* Cos(3*pi/16)*Cos(3*pi/16) << 15 */
#define C3C4 19266 /* Cos(3*pi/16)*Cos(4*pi/16) << 15 */
#define C3C5 15137 /* Cos(3*pi/16)*Cos(5*pi/16) << 15 */
#define C3C6 10426 /* Cos(3*pi/16)*Cos(6*pi/16) << 15 */
#define C3C7 5315  /* Cos(3*pi/16)*Cos(7*pi/16) << 15 */

#define C4C4 16384 /* Cos(4*pi/16)*Cos(4*pi/16) << 15 */
#define C4C5 12873 /* Cos(4*pi/16)*Cos(5*pi/16) << 15 */
#define C4C6 8867  /* Cos(4*pi/16)*Cos(6*pi/16) << 15 */
#define C4C7 4520  /* Cos(4*pi/16)*Cos(7*pi/16) << 15 */

/* Coefficient constants for vertical IDCT */
/* They use 16 bits of fractional precision */
#define TAN1 (UINT16) 13036 /* Tan(1*pi/16) << 16 */
#define TAN2 (UINT16) 27146 /* Tan(2*pi/16) << 16 */
#define TAN3 (UINT16) 43790 /* Tan(3*pi/16) << 16 */
#define COS4 (UINT16) 46341 /* Cos(4*pi/16) << 16 */

/* Arrays of constants */

/* Operator M8 coefficients in 2x4 groups, scaled by C1 */
DECLARE_CONST_I16x8x4(ConstM_C1,
                      C1C4, C1C2, C1C4, C1C6, C1C4, -C1C6, C1C4, -C1C2,
                      C1C4, C1C6, -C1C4, -C1C2, -C1C4, C1C2, C1C4, -C1C6,
                      C1C1, C1C3, C1C3, -C1C7, C1C5, -C1C1, C1C7, -C1C5,
                      C1C5, C1C7, -C1C1, -C1C5, C1C7, C1C3, C1C3, -C1C1)

/* Operator M8 coefficients in 2x4 groups, scaled by C2 */
DECLARE_CONST_I16x8x4(ConstM_C2,
                      C2C4, C2C2, C2C4, C2C6, C2C4, -C2C6, C2C4, -C2C2,
                      C2C4, C2C6, -C2C4, -C2C2, -C2C4, C2C2, C2C4, -C2C6,
                      C1C2, C2C3, C2C3, -C2C7, C2C5, -C1C2, C2C7, -C2C5,
                      C2C5, C2C7, -C1C2, -C2C5, C2C7, C2C3, C2C3, -C1C2);

/* Operator M8 coefficients in 2x4 groups, scaled by C3 */
DECLARE_CONST_I16x8x4(ConstM_C3,
                      C3C4, C2C3, C3C4, C3C6, C3C4, -C3C6, C3C4, -C2C3,
                      C3C4, C3C6, -C3C4, -C2C3, -C3C4, C2C3, C3C4, -C3C6,
                      C1C3, C3C3, C3C3, -C3C7, C3C5, -C1C3, C3C7, -C3C5,
                      C3C5, C3C7, -C1C3, -C3C5, C3C7, C3C3, C3C3, -C1C3);

/* Operator M8 coefficients in 2x4 groups, scaled by C4 */
DECLARE_CONST_I16x8x4(ConstM_C4,
                      C4C4, C2C4, C4C4, C4C6, C4C4, -C4C6, C4C4, -C2C4,
                      C4C4, C4C6, -C4C4, -C2C4, -C4C4, C2C4, C4C4, -C4C6,
                      C1C4, C3C4, C3C4, -C4C7, C4C5, -C1C4, C4C7, -C4C5,
                      C4C5, C4C7, -C1C4, -C4C5, C4C7, C3C4, C3C4, -C1C4);
```

```
/*************************************************************
* ROW_IDCT - 1D IDCT of row
*
* Inputs:
*   pSrc         - Pointer to input array in memory
*   pConst       - Pointer to array of constants
*
* Output:
*   Y            - Result vector
*
* Uses:
*   X, XP, XB, MP, ME, MO, A1, A2, Temp, ConstRound12Bit
*
* Description:
*   Computes the 1D Inverse Discrete Cosine Transform
*   of an 8-element vector of 16-bit signed elements.
*
*   The output is scaled by a factor of four, which is compensated in the
*   column idct.  This helps preserve accuracy.
*
*************************************************************/

#define ROW_IDCT(Y, pSrc, pConst);                                  \
{                                                                   \
    /* Load input row */                                            \
    LOAD_A_I16x8(X, pSrc);                                          \
                                                                    \
    /* Permute input to order 02134657*/                            \
    PERMUTE_I16x8_02134657(XP, X);                                  \
                                                                    \
    /* Extract elements 0 & 2, and repeat them 4 times */          \
    BROADCAST_PAIR_0_I16x8(XB, XP);                                 \
                                                                    \
    /* Multiply by coefficients in operator M8 and add results */   \
    LOAD_A_I16x8(Temp, &pConst[0]);                                 \
    MULT_ADDPAIRS_I16x8(MP, XB, Temp);                              \
                                                                    \
    /* Extract elements 4 & 6, and repeat them 4 times */          \
    BROADCAST_PAIR 2_I16x8(XB, XP);                                 \
                                                                    \
    /* Multiply by coefficients in operator M8 and add results */   \
    /* Sum top 4 rows of M8 */                                      \
    LOAD_A_I16x8(Temp, &pConst[1]);                                 \
    MULT_ADDPAIRS_ADD_N_I16x8(ME, XB, Temp, MP);                    \
                                                                    \
    /* Extract elements 1 & 3, and repeat them 4 times */          \
    BROADCAST_PAIR_1_I16x8(XB, XP);                                 \
                                                                    \
    /* Multiply by coefficients in operator M8 and add results */   \
    LOAD_A_I16x8(Temp, &pConst[2]);                                 \
    MULT_ADDPAIRS_I16x8(MP, XB, Temp);                              \
                                                                    \
    /* Extract elements 5 & 7, and repeat them 4 times */          \
    BROADCAST_PAIR_3_I16x8(XE, XP);                                 \
                                                                    \
    /* Multiply by coefficients in operator M8 and add results */   \
    /* Sum bottom 4 rows of M8 */                                   \
    LOAD_A_I16x8(Temp, &pConst[3]);                                 \
    MULT_ADDPAIRS_ADD_N_I16x8(MO, XB, Temp, MP);                    \
                                                                    \
    /* Add rounding amount */                                       \
    ADD_N_I32x4(ME, ME, ConstRound12Bit);                           \
```

```
                                                                    \
    /* Operator A8 */                                               \
    ADD_N_I32x4(A1, ME, MO);                                        \
    SUB_N_I32x4(A2, ME, MO);                                        \
                                                                    \
    /* Shift out the lower bits. */                                 \
    SRA_I_I32x4(A1, A1, 12);                                        \
    SRA_I_I32x4(A2, A2, 12);                                        \
                                                                    \
    /* Pack as 16-bit */                                            \
    PACK_N_I32x4(Y, A1, A2);                                        \
                                                                    \
    /* Correct order of last 4 values */                            \
    PERMUTE_I16x8_01237654(Y, Y);                                   \
}


/*************************************************************
* Idct8x8
*
* Inputs:
*   pSrc         - Pointer to input array in memory
*
* Output:
*   pDst         - Pointer to output array in memory
*
* Description:
*   Computes the 2D Inverse Discrete Cosine Transform of an 8x8 block
*   of 16-bit signed elements.
*
*************************************************************/

void Idct8x8 ( INT16 *pSrc, INT16 *pDst)
{
    /*
    ** Intermediate variables for horizontal IDCT
    */
    DECLARE_I16x8(X)    /* Input row */
    DECLARE_I16x8(XP)   /* Input row permuted */
    DECLARE_I16x8(XB)   /* Two columns of row repeated 4 times */
    DECLARE_I32x4(MP)   /* Partial results of operator M */
    DECLARE_I32x4(ME)   /* Result of operator M, even part */
    DECLARE_I32x4(MO)   /* Result of operator M, odd part */
    DECLARE_I32x4(A1)   /* Partial results of operator A */
    DECLARE_I32x4(A2)

    DECLARE_I16x8(Y0)   /* Row IDCT outputs */
    DECLARE_I16x8(Y1)
    DECLARE_I16x8(Y2)
    DECLARE_I16x8(Y3)
    DECLARE_I16x8(Y4)
    DECLARE_I16x8(Y5)
    DECLARE_I16x8(Y6)
    DECLARE_I16x8(Y7)

    DECLARE_I16x8(Temp) /* Auxiliary */

    /*
    ** Intermediate variables for vertical IDCT
    */
    DECLARE_I16x8(B0)   /* Output of operator B8^-1 */
    DECLARE_I16x8(B1)
```

```
DECLARE_I16x8(B2)
DECLARE_I16x8(B3)
DECLARE_I16x8(B4)
DECLARE_I16x8(B5)
DECLARE_I16x8(B6)
DECLARE_I16x8(B7)

DECLARE_I16x8(E0)   /* Output of operator E8^-1 */
DECLARE_I16x8(E1)
DECLARE_I16x8(E2)
DECLARE_I16x8(E3)
DECLARE_I16x8(E4)
DECLARE_I16x8(E5)
DECLARE_I16x8(E6)
DECLARE_I16x8(E7)

DECLARE_I16x8(F5)   /* Output of operator F8^-1 */
DECLARE_I16x8(F6)

/* Contant vectors */
DECLARE_I16x8(ConstTan1)
DECLARE_I16x8(ConstTan2)
DECLARE_I16x8(ConstTan3)
DECLARE_I16x8(ConstCos4)
DECLARE_I16x8(ConstRound5Bit)
DECLARE_I16x8(ConstRound5BitCorr)
DECLARE_I16x8(ConstCorr)
DECLARE_I32x4(ConstRound12Bit)

/* Set constant vectors */
SET1_I16x8(ConstTan1, TAN1)
SET1_I16x8(ConstTan2, TAN2)
SET1_I16x8(ConstTan3, TAN3)
SET1_I16x8(ConstCos4, COS4)
SET1_I16x8(ConstRound5Bit, 0x10)
SET1_I16x8(ConstRound5BitCorr, 0xF)
SET1_I16x8(ConstCorr, 0x1)
SET1_I32x4(ConstRound12Bit, 0x800)

/*
** Horizontal IDCT
*/
ROW_IDCT(Y3, (pSrc + 3 * 8), ConstM_C3);
ROW_IDCT(Y5, (pSrc + 5 * 8), ConstM_C3);
ROW_IDCT(Y1, (pSrc + 1 * 8), ConstM_C1);
ROW_IDCT(Y7, (pSrc + 7 * 8), ConstM_C1);
ROW_IDCT(Y2, (pSrc + 2 * 8), ConstM_C2);
ROW_IDCT(Y6, (pSrc + 6 * 8), ConstM_C2);
ROW_IDCT(Y0, (pSrc + 0 * 8), ConstM_C4);
ROW_IDCT(Y4, (pSrc + 4 * 8), ConstM_C4);

/*
** Vertical IDCT
*/

/* Operator B8^-1 */
ADD_N_I16x8(B0, Y0, Y4)
SUB_N_I16x8(B1, Y0, Y4)
MULT_H_ADD_N_I16x8(B2, Y6, ConstTan2, Y2)
MULT_H_I16x8(Temp, Y2, ConstTan2)
SUB_N_I16x8(B3, Temp, Y6)
MULT_H_ADD_N_I16x8(B4, Y7, ConstTan1, Y1)
```

```
MULT_H_I16x8(Temp, Y1, ConstTan1)
SUB_N_I16x8(B5, Temp, Y7)
MULT_H_ADD_N_I16x8(Temp, Y5, ConstTan3, Y5)
ADD_I16x8(B6, Temp, Y3)
MULT_H_ADD_N_I16x8(Temp, Y3, ConstTan3, Y3)
SUB_N_I16x8(B7, Y5, Temp)

/* Operator E8^-1 */
ADD_N_I16x8(E0, B0, B2)
ADD_N_I16x8(E0, E0, ConstRound5Bit)
SUB_N_I16x8(E3, B0, B2)
ADD_N_I16x8(E3, E3, ConstRound5BitCorr)
ADD_N_I16x8(E1, B1, B3)
ADD_N_I16x8(E1, E1, ConstRound5Bit)
SUB_N_I16x8(E2, B1, B3)
ADD_N_I16x8(E2, E2, ConstRound5BitCorr)
ADD_N_I16x8(E4, B4, B6)
ADD_N_I16x8(E4, E4, ConstCorr)
SUB_N_I16x8(E5, B4, B6)
SUB_N_I16x8(E6, B5, B7)
ADD_N_I16x8(E6, E6, ConstCorr)
ADD_N_I16x8(E7, B5, B7)

/* Operator F8^-1 */
ADD_N_I16x8(Temp, E5, E6)
MULT_H_ADD_N_I16x8(F5, Temp, ConstCos4, Temp)
ADD_N_I16x8(F5, F5, ConstCorr)
SUB_N_I16x8(Temp, E5, E6)
MULT_H_ADD_N_I16x8(F6, Temp, ConstCos4, Temp)
ADD_N_I16x8(F6, F6, ConstCorr)

/* Operator A8^-1 */
/* Y0 */
ADD_N_I16x8(Temp, E0, E4)
SRA_I_I16x8(Temp, Temp, 5);
STORE_A_I16x8((pDst + 0*8), Temp);
/* Y7 */
SUB_N_I16x8(Temp, E0, E4)
SRA_I_I16x8(Temp, Temp, 5);
STORE_A_I16x8((pDst + 7*8), Temp);
/* Y1 */
ADD_N_I16x8(Temp, E1, F5)
SRA_I_I16x8(Temp, Temp, 5);
STORE_A_I16x8((pDst + 1*8), Temp);
/* Y6 */
SUB_N_I16x8(Temp, E1, F5)
SRA_I_I16x8(Temp, Temp, 5);
STORE_A_I16x8((pDst + 6*8), Temp);
/* Y2 */
ADD_N_I16x8(Temp, E2, F6)
SRA_I_I16x8(Temp, Temp, 5);
STORE_A_I16x8((pDst + 2*8), Temp);
/* Y5 */
SUB_N_I16x8(Temp, E2, F6)
SRA_I_I16x8(Temp, Temp, 5);
STORE_A_I16x8((pDst + 5*8), Temp);
/* Y3 */
ADD_N_I16x8(Temp, E3, E7)
SRA_I_I16x8(Temp, Temp, 5);
STORE_A_I16x8((pDst + 3*8), Temp);
/* Y4 */
SUB_N_I16x8(Temp, E3, E7)
```

```
    SRA_I_I16x8(Temp, Temp, 5);
    STORE_A_I16x8((pDst + 4*8), Temp);

    END_OPTIMIZED();
}
```

C.2 16x16 L_1-Distance

Shortcut paths are supported when SHORTCUT_PATH is defined.

```
/************************************************************
* l1_dist_mmm.c
*
*   Implementation of L1-Distance of 16x16 blocks,
*   with and without interpolation, using Multi-Media Macro libraries.
*
*   This file is intended to be compiled for any of the following
*   target architectures:
*       - Intel MMX and SSE
*       - TriMedia TM1300
*       - AltiVec
*
*   This program, and Multi-Media Macro libraries were developed
*   by Juan Carlos Rojas as part of his PhD research at
*   Northeastern University.
*
************************************************************/

#ifdef SSE2
    #include "mmm_sse2.h"
#endif
#ifdef SSE
    #include "mmm_sse.h"
#endif
#ifdef TRIMEDIA
    #include "mmm_tm.h"
#endif
#ifdef ALTIVEC
    #include "mmm_altivec.h"
#endif

/* Use the following define to support shortcut paths. */
/*#define SHORTCUT_PATH */

/************************************************************
* L1Dist16x16
*
* Inputs:
*   pRef, pIn    - Addresses of input blocks
*   RowPitch     - Distance (in bytes) of vertically adjacent pixels
*   Limit        - Stop if sum exceeds this value
*
* Output:
*   Sum          - Accumulated SAD for this block
*
* Description:
*   Computes the L1-Distance (sum of absolute differences) between two 16*16
*   blocks of 8-bit unsigned integers. Block pIn is assumed to
*   be aligned to 16-byte boundaries, pRef may not be.
*
************************************************************/

#define SAD_ROW(dst, pRef, pIn, index)       \
```

```
                                                \
    /* Load next row of each input array */     \
    LOAD_U_U8x16(R1, pRef, index)               \
    LOAD_A_U8x16(I, pI)                         \
                                                \
    /* Accumulate SAD of this row */            \
    SAD2_ADD_M_U8x16(dst, R1, I, dst)


UINT32 L1Dist16x16(UINT8 *pRef, UINT8 *pIn, int RowPitch, int Limit)
{
    DECLARE_U8x16(R1)   /* Holds one row of reference block */
    DECLARE_U8x16(I)    /* Holds one row of input block */

    DECLARE_U32x4(Sad)  /* Vector with two partial sums */
    UINT32 Sum;         /* Integer result */

    CLEAR_U32x4(Sad)

    PREPARE_LOAD_ALIGNMENT(1, pRef)

    SAD_ROW(Sad, pRef + 0*RowPitch, pIn + 0*RowPitch, 1)
    SAD_ROW(Sad, pRef + 1*RowPitch, pIn + 1*RowPitch, 1)
    SAD_ROW(Sad, pRef + 2*RowPitch, pIn + 2*RowPitch, 1)
    SAD_ROW(Sad, pRef + 3*RowPitch, pIn + 3*RowPitch, 1)
    SAD_ROW(Sad, pRef + 4*RowPitch, pIn + 4*RowPitch, 1)
    SAD_ROW(Sad, pRef + 5*RowPitch, pIn + 5*RowPitch, 1)
    SAD_ROW(Sad, pRef + 6*RowPitch, pIn + 6*RowPitch, 1)
    SAD_ROW(Sad, pRef + 7*RowPitch, pIn + 7*RowPitch, 1)

#ifdef SHORTCUT_PATH
    SUM2_U32x4(Sum, Sad)
    if (Sum > Limit) {
        END_OPTIMIZED()
        return Sum;
    }
#endif

    SAD_ROW(Sad, pRef + 8*RowPitch, pIn + 8*RowPitch, 1)
    SAD_ROW(Sad, pRef + 9*RowPitch, pIn + 9*RowPitch, 1)
    SAD_ROW(Sad, pRef +10*RowPitch, pIn +10*RowPitch, 1)
    SAD_ROW(Sad, pRef +11*RowPitch, pIn +11*RowPitch, 1)
    SAD_ROW(Sad, pRef +12*RowPitch, pIn +12*RowPitch, 1)
    SAD_ROW(Sad, pRef +13*RowPitch, pIn +13*RowPitch, 1)
    SAD_ROW(Sad, pRef +14*RowPitch, pIn +14*RowPitch, 1)
    SAD_ROW(Sad, pRef +15*RowPitch, pIn +15*RowPitch, 1)

    /* Add partial sums*/
    SUM2_U32x4(Sum, Sad)

    END_OPTIMIZED()

    return Sum;
}
```

C.3 16x16 L_1-Distance with Interpolation

```
/************************************************************
* L1Dist16x16_InterpXY
*
* Inputs:
*   pRef, pIn    - Addresses of input blocks
*   RowPitch     - Distance (in bytes) of vertically adjacent pixels
*   Limit        - Stop if sum exceeds this value
*
* Output:
*   Sum          - Accumulated SAD for this block
*
* Description:
*   Performs half-pixel horizontal and vertical interpolation of pRef,
*   a 16x16 block of 8-bit unsigned integers, and computes the L1-Distance
*   (sum of absolute differences) between it and pIn,
*   another block of the same size.
*   Block pIn is assumed to be word-aligned, pRef may not be.
*
************************************************************/

#define SAD_INTERP_ROW(dst, pRef, pIn, index1, index2)   \
    COPY_U8x16(R1, R2)                                   \
    LOAD_ADJ_U8x16(R2, R3, pRef, index1, index2)         \
    AVG_U8x16(R2, R2, R3) /* Interpolate horizontally */ \
    AVG_U8x16(R1, R1, R2) /* Interpolate vertically */   \
    LOAD_A_U8x16(I, pIn)                                 \
    SAD2_ADD_M_U8x16(dst, R1, I, dst)


int L1Dist16x16_InterpXY(UINT8 *pRef, UINT8 *pIn, int RowPitch, int Limit)
{
    DECLARE_U8x16(R1)   /* Holds one row of reference block */
    DECLARE_U8x16(R2)
    DECLARE_U8x16(R3)
    DECLARE_U8x16(B)    /* Holds one row of input block */

    DECLARE_U32x4(Sad)  /* Vector with two partial sums */
    UINT32 Sum;         /* Integer result */

    CLEAR_U32x4(Sad)

    PREPARE_LOAD_ALIGNMENT(1, pRef)
    PREPARE_LOAD_ALIGNMENT(2, pRef+1)

    /* Load first row */
    LOAD_ADJ_U8x16(R2, R3, pRef, 1, 2)

    /* Interpolate horizontally */
    AVG_U8x16(R2, R2, R3)

    SAD_INTERP_ROW(Sad, pRef + 1*RowPitch, pIn + 0*RowPitch, 1, 2)
    SAD_INTERP_ROW(Sad, pRef + 2*RowPitch, pIn + 1*RowPitch, 1, 2)
    SAD_INTERP_ROW(Sad, pRef + 3*RowPitch, pIn + 2*RowPitch, 1, 2)
    SAD_INTERP_ROW(Sad, pRef + 4*RowPitch, pIn + 3*RowPitch, 1, 2)
    SAD_INTERP_ROW(Sad, pRef + 5*RowPitch, pIn + 4*RowPitch, 1, 2)
    SAD_INTERP_ROW(Sad, pRef + 6*RowPitch, pIn + 5*RowPitch, 1, 2)
    SAD_INTERP_ROW(Sad, pRef + 7*RowPitch, pIn + 6*RowPitch, 1, 2)
```

```
#ifdef SHORTCUT_PATH
    SUM2_U32x4(Sum, Sad)
    if (Sum > Limit) {
        END_OPTIMIZED()
        return Sum;
    }
#endif

    SAD_INTERP_ROW(Sad, pRef + 8*RowPitch, pIn + 7*RowPitch, 1, 2)
    SAD_INTERP_ROW(Sad, pRef + 9*RowPitch, pIn + 8*RowPitch, 1, 2)
    SAD_INTERP_ROW(Sad, pRef +10*RowPitch, pIn + 9*RowPitch, 1, 2)
    SAD_INTERP_ROW(Sad, pRef +11*RowPitch, pIn +10*RowPitch, 1, 2)
    SAD_INTERP_ROW(Sad, pRef +12*RowPitch, pIn +11*RowPitch, 1, 2)
    SAD_INTERP_ROW(Sad, pRef +13*RowPitch, pIn +12*RowPitch, 1, 2)
    SAD_INTERP_ROW(Sad, pRef +14*RowPitch, pIn +13*RowPitch, 1, 2)
    SAD_INTERP_ROW(Sad, pRef +15*RowPitch, pIn +14*RowPitch, 1, 2)
    SAD_INTERP_ROW(Sad, pRef +16*RowPitch, pIn +15*RowPitch, 1, 2)

    /* Add partial sums*/
    SUM2_U32x4(Sum, Sad)

    END_OPTIMIZED()

    return Sum;
}
```

GLOSSARY

3DNow! Multimedia extensions for the AMD K6 2 and later processors.

3DNow! Professional. Multimedia extensions for the AMD Athlon XP and later processors. It is a combination of Enhanced 3DNow! and SSE extensions.

AltiVec. Multimedia extensions for the Motorola PowerPC G4 processor.

Enhanced 3DNow! Multimedia extensions for the AMD Athlon and later processors.

H.263. Video compression standard by the International Telecommunications Union.

IDCT. Inverse Discrete Cosine Transform.

Intrinsics. Extensions to the C language that serve to indicate specific machine instructions to the compiler.

FIR. Finite-impulse response filter.

FFT. Fast Fourier transform.

GPP. General-purpose processor.

MMX. Multimedia extensions to the Intel Pentium, AMD K6 and later processors.

MPEG(2). Video compression standard by the Moving Pictures Experts Group.

Multimedia instruction set. Includes instructions that operate in parallel on parts of the registers and complex instructions designed for multimedia applications.

Multimedia programs. Programs that process video and/or audio information. For example, video compression.

SAD. Sum of absolute differences.

Scalar processor. A processor whose registers represent a single value at a time.

SIMD. Single instruction, multiple data paradigm of parallel processing.

Speedup. Ratio of optimized execution speed to the unoptimized one. Speedup = unoptimized execution time / optimized execution time.

SSE. Streaming SIMD extensions for the Intel Pentium III, Itanium, AMD Athlon XP and later processors. It is a complement to MMX extensions.

SSE2. Streaming SIMD extensions 2 for the Intel Pentium 4 processors.

VIS. Visual Instruction Set. Multimedia extensions for Sun UltraSparc processors.

BIBLIOGRAPHY

[1] William Chen, et al. "Native Signal Processing on the UltraSparc in the Ptolemy Environment," *Conference Record of the Thirtieth Asilomar Conference on Signals, Systems & Computers*, Pacific Grove, Calif., 1996, pp. 1368-72.

[2] Parthasarathy Ranaganathan, Sarita Adve and Norman P. Jouppi. "Performance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions," *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, 1999, pp. 124-35.

[3] Yi-Shin Tung, Chia-Chiang Ho and Ja-Ling Wu. "MMX-based DCT and MC Algorithms for Real-Time Pure Software MPEG Decoding," *Proceedings: IEEE International Conference on Multimedia Computing and Systems*, Florence, 1999, pp. 357-62.

[4] Ville Lappalanien. "Performance Analysis of Intel MMX Technology for an H.263 Video Encoder," *Proceedings: ACM Multimedia 98*, Bristol, England, 1998, pp. 309-14.

[5] Ravi Bhargava, et al. "Evaluating MMX Technology Using DSP and Multimedia Applications," *Proceedings of the IEEE Symposium on Microarchitecture (MICRO-31)*, Dallas, 1998, pp. 37-46.

[6] Berna Erol, Faouzi Kossentini and Hussein Alnuweiri. "Implementation of a Fast H.263+ Encoder/Decoder," *Conference Record of the Thirty-Second Asilomar Conference on Singals, Systems & Computers*, Pacific Grove, Calif., 1998, pp. 462-6.

[7] Huy Nguyen, and Lizzy Kurian John. "Exploiting SIMD Parallelism in DSP and Multimedia Algorithms Using the AltiVec Technology," *Proceedings of the International Conference on Supercomputing (ICS)*, 1999, pp. 11-20.

[8] Sebot Julien, and Nathalie Drach-Temam. "Memory Bandwidth: the True Bottleneck for SIMD Multimedia Performance on a Superscalar Processor," *European Conference on Parallel Computing (EUROPAR) 2001*, Manchester, England, 2001.

[9] N. Sreraman, and R. Govindarajan. "A Vectorizing Compiler for Multimedia Extensions," *International Journal of Parallel Programming*, no. 4, vol. 28, 2000, pp. 363-400.

[10] Markus Lorenz, Lars Wehmeyer and Thorsten Dräger. "Energy Aware Compilation for DSPs with SIMD Instructions," *Languages, Compilers, and Tools for Embedded Systems (LCTES'02) and Software and Compilers for Embedded Systems (SCOPES'02)*, Berlin, 2002.

[11] Samuel Larsen, and Saman Amarasinghe. "Exploiting Superword Level Parallelism with Multimedia Instruction Sets," *Proceedings of the SIGPLAN'00 Conference on Programming Language Design and Implementation*, Vancouver, 2000.

[12] Rainer Leupers. "Code Selection for Media Processors with SIMD Instructions," *Design Automation and Test in Europe (DATE) Conference Proceedings,* 2000, pp. 4-8.

[13] Aart Bik, et al. "Experiments with Automatic Vectorization for the Pentium 4 Processor," *9th Workshop on Compilers for Parallel Computers*, Edinburgh, Scottland, 2001.

[14] Aart Bik, et al. "Efficient Exploitation of Parallelism on Pentium III and Pentium IV Processor-Based Systems," *Intel Technology Journal*, Q1 2001 Issue, http://intel.com/technology/itj/q12001/articles/art_6.htm (current May 2003).

[15] Codeplay. *VectorC {PC} Overview,* http://www.codeplay.com/vectorc/index_pc.html (current May 2003).

[16] Andreas Krall, and Sylvain Lelait. "Compilation Techniques for Multimedia Processors," *International Journal of Parallel Programming,* no. 4, vol. 28, 2000, pp. 347-61.

[17] Gerald Cheong, and Monica Lam. "An Optimizer for Multimedia Instruction Sets," *Proceedings of the Second SUIF Compiler Workshop*, Stanford, 1997.

[18] Michael Metcalf, and John Reid. *Fortran 90 Explained.* Oxford: Oxford University Press, 1990.

[19] Vipin Kumar, et al. *Introduction to Parallel Computing: Design and Analysis of Algorithm,* Benjamin/Cummins, Redwood City, Calif., 1994.

[20] Paul Cockshott. *Vector Pascal, an Array Language.* Jan. 2002, http://www.dcs.gla.ac.uk/~wpc/reports/compilers/compilerindex/vp-ver2.pdf (current May 2003).

[21] Randall J. Fisher, and Henry G. Dietz. "Compiling for SIMD Within a Register," *Lecture Notes in Computer Science*, vol. 1656, Springer, Berlin, 1998, pp. 292-304.

[22] Randall J. Fisher, and Henry G. Dietz. "The Scc Compiler: SWARing at MMX and 3DNow!," Lecture Notes in Computer Science, vol. 1863, Springer, pp. 399.

[23] ISO/IEC WDTR 18037. *Extensions for the Programming Language C to Support Embedded Processors,* http://std.dkuug.dk/JTC1/SC22/WG14/www/docs/n972.pdf (current May 2003).

[24] Franz Franchetti, and Markus Püschel. "A SIMD Vectorizing Compiler for Digital Signal Processing Algorithms," *Proceedings International Parallel and Distributed Processing Symposium (IPDPS),* 2002.

[25] Franz Franchetti, and Markus Püschel. "Short Vector Code Generation and Adaptation for DSP Algorithms," *Proceedings International Conference on Acoustics, Speech, and Signal Processing (ICASSP),* 2003.

[26] Intel Corporation. *Intel C++ Compiler User's Guide*, http://www.intel.com/software/products/compilers/techtopics/c_ug_lnx.pdf (current May 2003).

[27] Jack Dongarra, et al. "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software,* no. 16, vol. 1, 1990, pp.1-17.

[28] Intel Corporation. *Intel Integrated Performance Primitives for Intel Pentium Processors and Intel Itanium Architectures,* http://www.intel.com/software/products/ipp/ipp30/ (current May 2003).

[29] David Shwartz, et al. *VSIPL 1.01 API,* http://www.vsipl.org/CD/vsiplv1p01_final1.pdf (current May 2003).

[30] Guy Blelloch, et al. *CVL: A C Vector Library: Manual: Version 2.1*, http://www.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/public/code/nesl/doc/cvl.ps (current May 2003).

[31] Matteo Frigo, and Steven G. Johnson. "FFTW: An Adaptive Software Architecture for the FFT," *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 3, 1998, pp. 1381-4.

[32] Matteo Frigo, and Steven G. Johnson. *FFTW: for Version 3.0-beta2*, http://www.fftw.org/fftw3.pdf (current May 2003).

[33] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. "Automated Empirical Optimization of Software and the ATLAS Project," *Parallel Computing,* no. 27, vols. 1-2, 2000, pp. 3-35.

[34] Daniel Zucker, and Ruby Lee. "Achieving Subword Parallelism by Software Reuse of the Floating-Point Data Path," *SPIE Proceedings 3021: Multimedia Hardware Architectures*, San Jose, Calif., 1997, pp. 51-64.

[35] Philips Semiconductors. *TriMedia TM1300 Data Book*. In *Philips TriMedia Documentation Set: SDE Version 2.1, CD-ROM, Oct. 1999*.

[36] Motorola, Inc. *AltiVec Technology: Programming Interface Manual*. Rev. 0, Jun. 1999, http://e-www.motorola.com/brdata/PDFDB/docs/ALTIVECPIM.pdf (current May 2003).

[37] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*. Vol. 2, *Instruction Set Reference,* 2003, ftp://download.intel.com/design/Pentium4/manuals/24547111.pdf (current May 2003).

[38] AMD. *AMD Extensions to the 3DNow! and MMX Instruction Sets Manual,* Mar. 2000, http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/22466.pdf (current May 2003).

[39] Sun Microsystems. *VIS Instruction Set User's Manual,* May 2001, http://www.sun.com/processors/vis/download/vsdk/visuserg.pdf (current May 2003).

[40] Chris Basoglu, Woobin Lee, and John O'Donnell. "The Equator MAP-CA DSP: An End-To-End Broadband Signal Processor VLIW," *IEEE Transactions on Circuits and Systems for Video Technology,* no. 12, vol. 8, 2000.

[41] IEEE Standard 1180-1990. IEEE Standard Specifications for the Implementations of 8X8 Inverse Discrete Cosine Transform.

[42] MPEG Software Simulation Group (MSSG). *MPEG-2 Encoder / Decoder, Version 1.2,* http://www.mpeg.org/MSSG/ (current May 2003).

[43] Intel Corporation. *Using Streaming SIMD Extensions 2 (SSE2) to Implement an Inverse Discrete Cosine Transform, Version 2.0,* Application Note AP-945, July 2000, http://cedar.intel.com/media/pdf/appnotes/sse2/w_idct.pdf. Source code: http://cedar.intel.com/media/pdf/appnotes/sse2/w_idct.zip (current May 2003).

[44] Intel Corporation. *A Fast Precise 8x8 DCT A Fast Precise Implementation of 8x8 Discrete Cosine Transform Using the Streaming SIMD Extensions and MMX Instructions, Version 1.0,* Application Note AP-922, Apr. 1999, http://cedar.intel.com/media/pdf/appnotes/ap922/ap922.pdf (current May 2003).

[45] Motorola, Inc. *2D Inverse Discrete Cosine Transform,* http://e-www.motorola.com/collateral/AVEC_2DICOSTRANS.zip (current May 2003).

[46] Philips Semiconductors. *Case Studies.* Book 2, Part D, Chapter 12 of *Philips TriMedia Documentation Set: SDE Version 2.1,* 1999, CD-ROM.

[47] Intel Corporation. *Using Streaming SIMD Extensions in a Motion Estimation Algorithm for MPEG Encoding, Version 1.2,* Application Note AP-818, Jan. 1999, http://cedar.intel.com/media/pdf/appnotes/ap818/motion_e.pdf. Source code: http://cedar.intel.com/media/pdf/appnotes/ap818/samples.zip (current May 2003).

[48] Intel Corporation. *Block-Matching in Motion Estimation Algorithms Using Streaming SIMD Extensions 2 (SSE2): Version 2.0,* Application Note AP-940, July 2000, http://cedar.intel.com/media/pdf/appnotes/sse2/w_me_alg.pdf. Source code: http://cedar.intel.com/media/pdf/appnotes/sse2/w_motion_est.zip (current May 2003).

[49] Motorola, Inc. *Sum of Absolute Differences,* http://e-www.motorola.com/collateral/AVEC_SAD.zip (current May 2003).

[50] Apple Computer Corp. *Project Builder Programming Examples,* http://developer.apple.com/hardware/ve/downloads/altivecPBExample.sit.hqx (current May 2003).

[51] Wen-Hsiung Chen, Harrison Smith and S.C.Fralick. "A Fast Computational Algorithm for the Discrete Cosine Transform," *IEEE Transactions on Communications*, no. 9, vol. 25, 1977, pp. 1004-9.

[52] Christoph Loeffler, Adriaan Ligtenberg and George S. Moschytz. "Practical Fast 1-D DCT Algorithms with 11 Multiplications," *International Conference on Acoustics, Speech, and Signal Processing*, Glasgow, Scotland, 1989, pp. 988-91.