GENERAL-PURPOSE SIMD WITHIN A REGISTER: PARALLEL PROCESSING ON CONSUMER MICROPROCESSORS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Randall James Fisher

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2003

In memory

of my father, Robert, who sparked and supported my interests in many things, and

my grandmother, Ethel, who helped me find beauty in the world around me.

I miss you both very much.

For Shelbi and Bobby, and the rest of my family and friends

ACKNOWLEDGMENTS

I would like to thank my committee: My adviser, Hank Dietz, for his guidance and for keeping things interesting; his co-chair, Leah Jamieson, for stepping in when she really had more than enough things to worry about; and Ed Delp and Zhiyuan Li for their patience and support. I would also like to thank the ECE graduate coordinator, Daniel Elliott, for the same.

I would also like to thank several people for their assistance in this research and preparation of this dissertation. Tim Mattox always provided a willing sounding board, as did my other colleagues, Soohong Kim, Gayathri Krishnamurthy, Ray Hoare, Ekechi Nwokah, Will Cohen, and Rick Kennell (I apologize if I have misspelled anyone's name). Rick was also helpful during the preparation of this document. Matt Golden was especially helpful with bureaucratic matters, as was Andy Hughes with formatting and procedural issues.

Dondi Bogusky deserves special recognition for allowing me to talk him into putting Linux on the G4 system in his office so that I could use it. My sister-in-law, Rhonda, also deserves credit for her help with Internet access and for providing caffeine over the summer.

I would especially like to thank Ed Bos, without whose help I would not have made it through my undergraduate years, and Tom DeMarse, who kept me sane during my years at Purdue. Of course, it was his fault that I came here to begin with. I would also like to thank all the other friends I have made at Michigan State and Purdue for, well, for being my friends.

Finally, I would like to thank my mother, Merrill, and my brother, John, for their love and support throughout my college career and life.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

fxb	A partitioned entity with f fields of b bits each
fxbs	A partitioned entity with f signed fields of b bits each
fxbu	A partitioned entity with f unsigned fields of b bits each
fxbf	A partitioned entity with f floating point fields of b bits each
Part	Indicates a partitioned operand
Scalar	Indicates a partitioned operand with identical field values
Element	Indicates one field of a partitioned operand
Single	Indicates a partitionable register taken as a single unpartitioned value
Immed	Indicates an immediate operand encoded in the instruction itself
Acc	Indicates that the result will be added to the accumulator
Acc Init	Indicates that the result will be stored in the accumulator
Acc Diff	Indicates that the difference will be added to the accumulator
Acc Sub	Indicates that the result will be subtracted from the accumulator

ABSTRACT

Fisher, Randall James. Ph.D., Purdue University, May, 2003. General-Purpose SIMD Within A Register: Parallel Processing On Consumer Microprocessors. Major Professors: Henry G. Dietz and Leah H. Jamieson.

Recent extensions to microprocessor instruction sets are intended to speed-up multimedia algorithms by allowing SIMD parallel processing over multiple data fields within each processor register. These extensions, while effectively supporting hand-coding of some multimedia tasks, do not directly support a high-level parallel programming model. Unfortunately, the extensions vary widely across different processor families, making portability difficult to achieve. Even within one set of extensions, each operation is supported only for certain field widths, and the widths supported are different for different operations. This thesis will define a general-purpose SWAR (SIMD Within A Register) programming model. This model will be implemented for multiple target architectures: initially as compatible libraries, then as optimizing compilers accepting a simple high-level parallel language. The new SWAR libraries and compiler technology should enable a much wider range of applications to achieve speed-up through SIMD execution using COTS microprocessors.

1. INTRODUCTION

1.1 Motivation

Modern commodity microprocessors employ a limited form of parallel processing in order to speed up multimedia algorithms. While these modified architectures are similar to certain traditional parallel processing models, they have unique and varied constraints on how they can be used. Traditional models of parallel processing are based on more powerful architectures and thus do not account for these constraints. To better reflect the capabilities and limitations of these new architectures, and to bridge the gaps between them, a new abstract model is required. We call this new processing model SWAR (SIMD Within A Register).

1.1.1 A Brief Introduction to Processing Models

To understand why previous abstract models are not sufficient, we need to have an understanding of these models and their purposes. Flynn's classification of processing systems [1] is useful in this endeavor, and we will use it to help denote the various processing models in this discussion. While we will often treat them as being interchangeable, computer architectures and the languages used to program them may actually be based on different processing models. In this discussion, we will differentiate between architectural and programming models as necessary. Also, these models are presented in an order that is not necessarily chronological, but should highlight their salient properties.

Sequential processors execute a single instruction on a single set of scalar operands at any given time. To reflect this fact, Flynn named this processing model SISD (Single Instruction stream, Single Data stream). This model is the basis for most computers including the first microprocessor systems. While SISD systems are sufficient for many of the computing problems we encounter on a daily basis, they are too slow to be used to solve very large problems in a reasonable amount of time. A desire to improve upon this situation led to the development of new architectures and processing models.

Pipelined processors are SISD machines in which each instruction is executed in a single processing unit with multiple stages. The processor is set up like an assembly line with each stage performing one part of the total work needed to complete the instruction. An instruction can occupy only one stage of the pipeline at any given time, leaving the remaining stages available to other instructions. Thus, multiple instructions from an instruction stream can be in the pipeline simultaneously.

In mathematics, a vector is a single-dimensional, multi-element object. *Vector programming models* help programmers express operations on vectors more concisely than do scalar models. Many of these operations are applied to each of the vector's elements independently or cumulatively. For example, adding two vectors is equivalent to adding their elements in a pairwise manner. Vector programming models allow such operations to be expressed as a single operation on a vector rather than as a series of scalar operations on the vector's elements.

Vector processors were developed to minimize the costs associated with performing vector operations. They capitalize on the fact that most vector operations are repeated over many elements. For these operations, some of the pipeline execution stages need only be performed once for the entire vector. Thus, vector processors reduce execution time by removing redundancy in the execution of identical element-wise operations.

The simplest vector processors execute repetitive vector operations by sequentially running the vector elements through an ALU which performs an identical operation on each element. *Pipelined vector processors* allow multiple ALUs to be chained together to form an execution pipeline similar to that of a pipelined SISD processor. This increases the performance of the processor by allowing multiple vector operations to not only share control stages but also to overlap in time.

While these vector processors can achieve significant speedup, they fail to fully exploit the available parallelism of vector code. This is because they perform each operation on only one set of corresponding elements at a time. Thus, in some sense, they are actually just improved SISD machines. To obtain better performance, processing models were developed in which work is performed on multiple parts of a problem simultaneously (i.e. in parallel). This is known as *parallel processing.*

These new processing models were more closely matched to the large, scientific problems which high-performance systems were intended to address than were the scalar models upon which sequential and simple vector processors were based. These problems included the modeling of physical phenomena such as weather and nuclear reactions and the analysis of observed data such as satellite photographs.

In these problems, physical environments or entities are represented by large data sets. For example, each datum may represent the value of some physical property at one of thousands of points within an environment at some given time. At each point, the predicted future value of this property is a function of its current value and its value at each of the neighboring points in multiple directions. Thus, solving these problems typically requires not only large amounts of computational power but also timely access to both local and neighboring point data.

Parallel processors are systems which are based on parallel processing models. These systems consist of multiple processing units which operate on multiple instruction streams simultaneously. Typically, these processing units are connected to form an array via one or more communications networks. These *interconnection networks*, which are sometimes referred to simply as the *interconnect*, allow point data to be passed between neighboring processing units in one or more dimensions. Thus, these systems were designed to be appropriate targets for large-scale scientific problems.

There are two major forms of parallelism which these systems exploit. *Control parallelism* refers to the separation of a problem into multiple independent sections which can be executed simultaneously. *Data parallelism* refers to problems with a regular nature in which the same series of operations must be applied to multiple sets of data. Different processing models and architectures were developed to exploit these differing forms of parallelism.

MIMD (Multiple Instruction stream, Multiple Data stream) is a parallel processing model that was developed as a means of exploiting control parallelism in large problems. The computational nodes of a MIMD processor each execute a series of instructions which may differ from that of the other nodes. This allows each node to execute an independent section of the problem.

MIMD processors can simultaneously run multiple unrelated sections of code or multiple copies of identical code. This allows various programming models to be used to program these systems. For example, the *MIMD programming model* is based on the assumption that the problem is divided into pieces that may need to be synchronized occasionally, but are otherwise completely independent. The *SPMD programming model* (Single Program, Multiple Data) is similar, but is based on the assumption that the independent pieces are identical.

While MIMD processing is quite versatile, there is a cost associated with this flexibility due to the replication of both computational and control hardware. This makes MIMD relatively expensive. Other processing models were developed as a means of avoiding this cost while still benefitting from some form of parallelism.

One such model was SIMD (Single Instruction stream, Multiple Data stream), which was developed as a relatively inexpensive means of exploiting data parallelism. This is done by applying each operation simultaneously to as many data points as possible. Thus, a single instruction stream is executed on multiple data streams.

SIMD systems can be divided into vector-based and array-based systems. *Vector SIMD processors*, also called *vector parallel processors*, are single-dimensional SIMD processors designed to operate on vector data objects. *SIMD array processors* are SIMD architectures whose PEs are connected in shapes of two or more dimensions.

Vector SIMD processors execute repetitive vector element operations in a simultaneous fashion. With these processors, data is loaded into a set of vector registers which hold some fixed number of elements. Operations are then performed on some or all of these elements simultaneously. This allows the processor to take advantage of the data parallelism inherent in vector processing to achieve higher performance than non-parallel vector processors.

While vector processors shorten the time required to solve certain classes of problems, they are not well-adapted to solving large multi-dimensional problems efficiently. Array processors are better suited to these problems because they allow arrays to be processed with their coordinate systems intact. That is, these processors allow data from neighboring points in space to be stored in neighboring processing units.

A typical SIMD system has a single *control unit*, usually abbreviated CU, and an array of multiple processing units which are often called *processing elements* (PEs). The CU is responsible for reading a single stream of instructions from memory, decoding these instructions into control signals, and issuing the control signals to the PE array. Each PE executes the operation defined by the control signals on its own data stream. This data stream may be from a shared memory, but is usually from a memory which the PE holds privately.

Using a single controller makes SIMD systems inexpensive compared to the more general MIMD architectures in which the control unit is replicated for each of the PEs. Yet, for data parallel problems, SIMD retains the benefits of parallel processing associated with MIMD, thus giving it a higher performance to cost ratio.

One drawback of SIMD programming models is that they are severely limited when compared to MIMD models because every processor must execute exactly the same instruction simultaneously. This limits them to SPMD-style programs which are executed with every instruction synchronized.

This also makes the handling of high-level language control constructs, such as `if` statements, difficult. Typical SIMD systems have special hardware to turn PEs on and off (or equivalently, to block the side-effects of execution) depending on the local conditions of the PE. If this hardware is not present, the executed program must be modified to nullify the effects of code that should not have been executed.

1.1.2 Multimedia Extensions

Several programming and machine models have been developed to improve performance over traditional SISD computers. These were well-developed by the 1990s when manufacturers of commodity SISD microprocessors began experimenting with non-SISD architectures for multimedia processing.

Early work in this area focused on enhancing processors with on-chip graphical hardware. This was typically in the form of a handful of instructions for speeding common graphics operations. This included operations such as interpolating the position of non-end points on a line when only the endpoints were known and testing for the visibility of objects to determine if they should be drawn on the screen. These efforts were very limited, and not intended for general-purpose computing. However, they used methods that were later employed in implementing more general multimedia extensions.

In the 1990s, several manufacturers of commodity microprocessors began expanding their instruction set architectures with *multimedia extensions*. These were intended to speedup data parallel algorithms used in graphical and audio processing while keeping the amount of architectural modification required to implement them at a minimum. Of the processing models mentioned, the closest match to these goals was the vector parallel subset of SIMD. Thus, the designers of these multimedia extensions implemented them as sets of SIMD-like instructions.

When executed, these instructions are performed on multiple streams of data residing in a single CPU register. Thus, these extended architectures implement a form of SIMD processing. However, they differ from previous SIMD architectures because they have only one central processing unit (CPU) whose operation has been altered to act like a CU with a set of PEs, rather than an actual set of PEs driven by a single, separate control unit.

This means that the entire set of PEs shares the CPU's single data path. Data can only be moved in and out of the PEs in the equivalent of block form from a single shared memory. Thus, a memory access moves a block of consecutive bits between a set of neighboring PEs and a single word in memory. This restriction is a significant limitation compared with typical SIMD architectures, which could load data from independent addresses or from private memories.

Data communication is also significantly different because there is often no equivalent to the communications networks employed in typical SIMD systems. Often `SHIFT` and `ROTATE` instructions are the only means available to move data between these pseudo-PEs. One communication type used in later SIMD architectures is a vector-indexed communication. This allows each PE to access data stored by some other PE, independent of the actions of the remaining PEs. Few multimedia architectures can perform such a generalized communication.

While not exactly SIMD, these SIMD-like extensions serve their intended purpose by allowing assembly language programmers to capture some of the potential speedup due to the data parallel nature of the targeted algorithms. Unfortunately, few of these extensions were designed with the intention of developing a complete processing model.

Usually, the registers and control logic used to implement these extensions needed to be enhanced to allow SIMD-like processing. This required considerable investment in the redesign and modification of the existing architecture. To minimize this investment while maximizing its perceived benefits, each of the extension sets has been targeted to support the multimedia algorithms that are believed to be most often used on its host platform. Thus, these extensions have limited functionality and tend to support only those data types and sizes which are normally used in multimedia.

Because of the variation in the architectures and the algorithms which are typically run on them, the instructions and data sizes supported often differ substantially between extension sets. Even within a single extension set, an instruction may exist to perform a particular operation on one size of data, but not on another size. This was intentionally done, based on the assumption that some operations are performed often on certain types of multimedia data, but rarely on others.

These variations and limitations are the primary problem with multimedia extensions, and limit their usefulness substantially. As a result, these extensions are sufficient for hand-coding architecture-specific, SIMD-parallel, multimedia operations at the assembly level, but are less useful beyond this scope. Variations between extension sets make code portability difficult, and the lack of consistent support for differing data sizes often forces format conversions between successive parallel operations. Finally, these extensions simply do not support certain data sizes and operations which may be useful to applications programmers in the future.

1.1.3 My Thesis

I believe that the set of applications which can benefit from these extensions is unknown and not limited to multimedia algorithms and data types. Also, that it is likely that multimedia extensions will continue to evolve, with some growing into more general systems and others dying out. Thus, not only will programmers need to be able to port code from one architecture to another, they will also want their code to take advantage of future capabilities without having to be rewritten for each new architecture.

Current programming models are either target-specific, based directly on some target's multimedia extensions, or based on programming models which do not match the capabilities of these architectures. These models are also unnecessarily limited to currently common data types and sizes. This ultimately limits their usefulness to those types of applications which we are able to foresee in the near future, and also prevents programmers from expressing algorithms which are best suited to non-standard data precisions. To move beyond the current situation, a general-purpose programming model for the form of SIMD processing described above should be developed.

This form of parallelism, in which a single CPU register holds multiple data items that are operated on in a SIMD manner, is referred to as “microparallelism” by Alpern, Carter, and Gatlin [2], and belongs to a class of operation known as “sub-word processing.” We will reserve the former term for any form of parallelism performed within a single register, including concepts such as single-register VLIW, and the latter term to mean any form of processing data which resides in less than a full machine word (e.g. byte operations on a 32-bit machine).

Thus, we shall consider the SIMD form of parallelism that this thesis addresses to be a subset of both microparallelism and sub-word processing. We refer to this form of processing as *SWAR* (SIMD Within A Register) [3].

While the limitations of multimedia extensions make it difficult to develop a consistent, portable, general-purpose SWAR programming model, they are not fatal. In fact, a generalized programming model can be developed which can target standard processor families with no SWAR-like extensions whatsoever.

It is my goal in this research to create a SWAR processing model which extends beyond the limits of current models, and to lay the groundwork for continued development of this form of parallel processing.

1.2 Related Work

When this work was first proposed in 1997 [4], we were unaware of any other groups pursuing a high-level approach to general-purpose SWAR processing. Known support for SWAR processing was limited to assembly-level programming tools and high-level multimedia libraries. Since then, the situation has changed with various groups now performing related work.

While some of this work is similar to that presented in this thesis, to our knowledge there are still no other groups which take as broad an approach to SWAR processing as the one presented here. In this section, we discuss related work in the context of the pursuit of a general-purpose SWAR processing model.

These efforts can be separated into four primary types: software-only methods for SWAR processing, non-compiler tools which assist the programmer in the use of multimedia instructions, pre-written libraries which make use of multimedia instructions, and compiler support for SWAR processing. Some of this support was discussed in [5]. That work is updated and expanded here.

1.2.1 Software-only Methods

In his Doctoral Dissertation to the Royal Melbourne Institute of Technology [6], Mark Spieth presented the Single Processor Single Instruction Multiple Data processing model. This model is similar to that of SWAR, but is limited in several ways.

The primary goal of the research was to "explore the feasibility of the software only solution to the parallel implementation of arithmetic operations in single processors." This was a less ambitious goal than that proposed here which includes the use of SWAR hardware, expansion of the model to arbitrary data sizes, and the development of a fully portable programming model and related compiler technology.

The work by Spieth is a more complete theoretical treatment of the subset of the SWAR work dealing with the processing of packed standard integer data using software techniques on unenhanced hardware, primarily as it relates to image processing.

In his thesis, Spieth explored various representations of numeric information and provided a mathematical framework of packed number representations. The primary method explored was *aliasing*, in which the sign bit of each register data field is conceptually extended into the upper fields of the register and combined with the data in those fields. This causes the lower field data to affect the bit patterns stored in the upper fields. An unaliasing step is required to extract individual field data from the register.

Algorithms were provided for performing the operations Spieth considered to be valid for SPSIMD processing. These include addition, subtraction, constant multiplication and division, bit shifts, Boolean (i.e. bitwise logical), and conditionals within which are included minimum, maximum, and absolute value operations. This is a limited set compared to that of the SWAR model.

These algorithms were evaluated mathematically to determine the effects of aliasing on their operation and performance. It was found that aliasing places limits on the domains of the operands of these operations. Calculations of the theoretical speedup of these algorithms were also provided. These appear to be compared to software implementations of the same operation on unpacked data rather than against possible hardware implementations.

Spieth also examined the removal of the restrictions of the SPSIMD paradigm. These are the restriction of operation domains to prevent overflow from occurring and the restriction of result precisions to those of the source operands. Removal of the first restriction would allow the operand domain to encompass a larger range of values. Removal of the second restriction would allow intermediate calculations to increase in precision.

In the discussion of this examination, Spieth described *split word processing* where packed data is "split" into multiple packed words which each contain a subset of the packed data. This includes techniques that were discussed early in 1997 by Professor Dietz [3] and which are used extensively within the Scc compiler discussed later in this thesis. One of these techniques is the *virtual spacer technique* for implementing arithmetic operations that may overflow. Another is the general method of temporarily promoting packed data to a greater intermediate precision, performing operations at this precision, then repacking the data into its original precision.

Spieth found that removing the restrictions of the SPSIMD model using split word processing was effective, but subject to overhead, memory interface speed, and the set of assumptions one could make about the operands.

Tests of the effectiveness of the SPSIMD model were performed on several hand-coded image processing algorithms. This was done by comparing the results obtained using the SPSIMD version with those obtained for rolled and unrolled looped, sequential implementations as baselines. This was done on five different machines, running four different operating systems, and compiled with GCC or Borland C using their full set of optimizations.

Spieth also briefly discussed other criteria for evaluating SPSIMD processing including cost, convenience, and suitability. He specifically mentioned that he believed that the development of compiler extensions would improve the situation by providing packed data structures and parallel operations. This is one of the goals of my work and is beyond the scope of Spieth's.

A performance comparison of the methods used by Spieth versus those used in the compiler implementation described in this research would be an interesting future work. Also, Spieth's work should be further explored for possible alternative compiler implementations of SWAR operations which are not supported by hardware.

1.2.2 Non-compiler Tools

The lowest level of support for the use of multimedia extensions includes tools such as profilers and debuggers. Neither of these is in the realm of a programming model and can safely be ignored, but we will briefly mention some examples to convey a sense of their utility.

The VTune optimization package from Intel [7] provides programmers with performance tuning tools which analyze source code and offer advice for using Intel's multimedia extensions to improve it. This would typically be used in an ad hoc manner with programmers performing a coding cycle of writing code, profiling, then rewriting the code to try to get better performance. For some time, this was the only significant means of support provided by Intel for its multimedia extensions.

NuMega Technologies' SoftICE for Windows 95 and SoftICE for Windows NT [8] are debuggers which allow the disassembly of MMX instructions. These allow the programmer to use any available method of generating code which contains multimedia instructions, then debug or profile the resulting assembly code. It is likely that most multimedia-aware C/C++ compilation packages now include a debugger and/or integrated disassembler.

1.2.3 Libraries

Pre-written libraries provide a high-level interface to a target's multimedia instructions. These libraries are usually both application- and target- specific, and perform common high-level operations which are comprised of multiple hardware instructions. They provide a means for applications programmers to exploit a target's multimedia extensions without being concerned with the details of the architecture; however, they typically do not address the issues of generality or portability.

Several application-specific libraries have been developed for MMX, including signal processing [9], image processing [10], speech recognition [11], and speech to text libraries[12]. A set of "Performance Libraries", to which the above libraries may belong, are included with Intel's Fortran and C++ compilers. These libraries are not intended to provide a general-purpose programming model, and support only specific data sizes.

Apple has adapted its core math libraries to make use of Motorola's AltiVec [13] extensions. They plan to rewrite their other libraries for this purpose in the future.

Sun Microsystems provides a C library called "mediaLib" [14] for the VIS extension set. mediaLib can be freely downloaded in binary form for certain platforms after a required licensing and non-disclosure agreement [15] is electronically accepted. Documentation for mediaLib is freely downloadable, and indicates that mediaLib is a high-level library which offers support for basic 8-, 16-, and 32-bit operations, as well as advanced functions such as FFTs.

The libSIMD project [16] is an attempt to define a portable math library for "commonly-used algorithms" across SIMD-enhanced and unenhanced architectures. The goal is to support "trigonometric, complex number, quaternion and FFT operations" on scalar, vector, and matrix objects. Functions are expected to be implemented using inline assembly code to access multimedia instructions and C code for portability to unenhanced architectures.

While plans for libSIMD are broad, its functionality is currently limited, consisting primarily of floating-point operations. Vectors and matrices appear to be limited to single fragment or sub-fragment lengths. The function listings in the documentation refer to 2-vectors, 3-vectors, and 4-vectors, while matrix functions operate on 2x2, 3x3, and 4x4 matrices.

libSIMD function arguments are objects in memory and results are stored to memory. Unless the compiler is able to perform optimizations across these procedures, possibly via inlining, then the memory access overhead will be too great to achieve significant speedup. Our decision to concentrate on a compiler rather than a general library was partially due to this fact.

The primary benefit of the libSIMD library would be portability of code between various multimedia-enhanced and unenhanced targets. However, this aspect seems to be insufficiently developed at this time as libSIMD is currently targeted only to AMD's 3DNow! extension set. This should change in the future as the author targets other multimedia extensions.

1.2.4 Compiler Support for SWAR

Current compiler support for SWAR processing consists primarily of various methods for exploiting multimedia extensions. This support falls into five major categories:

- *Inline assembly and compiler intrinsics.* This type of support gives the programmer low-level access to the instructions in the target's multimedia extension set. This allows the programmer to use multimedia instructions, but with a minimum of compiler support. Programmers must maintain type and partitioning information themselves and choose the correct intrinsic to use based on this knowledge. In some cases, the compiler is able to optimize the resulting low-level code.

- *Classes or types which represent a fragment.* Compiler support of this type is also limited to low-level access, but type and partitioning information is tracked for the programmer via the type or class system of the source language. This information may be used by the compiler to ensure that the correct assembly instruction is executed based on the partitioning of the fragment operands.

- *Automatic vectorization of loops.* This type of support provides an abstract model which hides the use of extended instructions. With this type of support, well-known techniques are used to parallelize loops in existing code. The primary disadvantage is that loops must conform to certain forms for the compiler to recognize that they are parallelizable.

- *Automatic vectorization of basic blocks.* This type of support also provides an abstract model which hides the use of extended instructions. Here, code in a basic block is combined into operations on fragments. This is a more general approach than vectorization of loops because the code does not have to be in loop form to be vectorized. The primary disadvantage is the amount of work and space required to combine the code into vector operations.

- *Languages with first-class vector objects.* This type of support also provides an abstract model which hides the use of extended instructions. Here, the structure and semantics of the language indicate which operations can be automatically parallelized. This is more restrictive than automatic parallelization of basic blocks, but provides a concise method for describing vector operations.

We will now look at each of these categories in turn, and describe some of the related work which has been, or is being, conducted along these lines.

Inline Assembly and Compiler Intrinsics

Inline assembly is low-level code for the target machine which is inserted into high-level language source code. This code is typically emitted directly into the assembly code generated by the high-level language compiler. This lets programmers use assembly language instructions of whose existence the compiler is unaware. In many cases this is the only form of support that the compiler provides for the use of extended instruction sets.

Compiler intrinsics are built-in functions which provide a function-call-like high-level interface to the target's machine instructions. Generally, these are trivial to implement and are usually just preprocessor macros which hide inlined assembly code which is used to execute a single instruction. These intrinsics are intended to provide access to instructions that the programmer would not otherwise be able to use, but generally do not provide functionality beyond the limits of the extended instruction set.

Inline assembly and compiler intrinsics operate at too low a level to be considered for a portable general-purpose SWAR processing model. However, this is often the starting point for other forms of support, so we will briefly survey some of the commercial compilers which support the use of multimedia instructions via intrinsics and/or macros.

Both Intel's Fortran [17] and C++ [18] compilers supply a set of intrinsics for their MMX, SSE, and SSE2 instruction sets. These intrinsics provide a means of describing the application of these instructions to objects in memory. The compiler is then responsible for register allocation and optimization of the resulting code.

Microsoft's Visual C++ version 5.0 compiler [19] also provides inline assembly support for MMX instructions as well the ability to disassemble code containing these instructions.

Metrowerks' CodeWarrior [20, 21] compiler provides inline assembly support for both MMX and AMD's 3DNow! instructions. This is one of several compilers of this product line which are targeted to different architectures. At least one version, CodeWarrior for Mac OS Professional Edition [22], supports AltiVec, although it isn't clear how.

Q Software Solutions LCC-Win32 compiler [23] also provides intrinsic support for MMX and 3DNow!. This compiler is an extension of the lcc compiler created by Fraser and Hanson for their text on compiler design [24].

The VectorC{PC} [25] C/C++ compiler by codeplay, Ltd., provides inline assembly support and intrinsics for the MMX, 3DNow!, and SSE extension sets. This compiler is intended primarily for the development of graphics-intensive games.

Green Hills Software makes an optimizing C/C++ compiler [26] which supports Motorola's AltiVec via a set of high-level intrinsics.

The VIS Software Developer's Kit (VSDK) [27] includes a set of macros for using Sun's VIS extensions. VSDK can be freely downloaded in binary form for certain platforms after a required licensing and non-disclosure agreement [28] is electronically accepted. The documentation for VSDK is part of the licensed package.

According to [29], C compilers which provide access via macros for Hewlett-Packard's MAX-2 extensions, Sun's VIS extensions, and the multimedia instructions of the MicroUnity and and Philips' Trimedia architectures have been available since the mid-1990's. The authors had suggested that a set of industry standard macros be developed. To the best of my knowledge, this has never been done.

Classes or Types which Represent a Fragment

A vector *fragment* is the amount of parallel data than can reside in a single multimedia-enhanced register. Conceptually, long vectors of data can be broken into multiple smaller vectors which fit into a register. It is these small vectors that we refer to as a fragment.

Object-oriented classes or simple type definitions which represent a fragment can provide a first-class feel to these objects and the operations on them. To do this, class definitions include functions which overload common operators with parallel versions of the operation. Conversely, the use of non-class type definitions generally requires a modification to both the high-level language and the associated compiler to support parallel operations on these objects.

Several compilers support the use of multimedia extensions via class or type definitions. Usually, these fragment-based models are built on top of a set of intrinsics and support only the operations and partitionings native to the target's multimedia extension set. The following is a brief survey of a few of the compilers that provide this form of support.

The Intel C++ compiler includes class libraries for operating on MMX, SSE, and SSE2 fragments.

Free Pascal [30, 31] includes predefined array types for MMX and 3DNow!, and extends Pascal through what are essentially compiler directives to allow some first-class operations on these types.

Oxford Micro Devices' C compiler for its A236 Parallel Video DSP chip [32], which has instructions similar to MMX, provides predefined `struct` types for describing fragments. Arithmetic and comparison operations on these types are performed on a single fragment of data.

Motorola has developed an extension of the C programming language which includes a new "vector" type to represent a single AltiVec fragment. This extension is not intended to be portable to other architectures, and requires a modified version of the GNU C compiler [33], GCC, which generates AltiVec instructions to perform operations on these "vector" objects.

Green Hills Software's optimizing C/C++ compiler [26] also supports AltiVec via Motorola's "vector" extensions.

Automatic Vectorization of Loops

Under strict conditions, and usually with hints from the programmer, some compilers are able to vectorize simple data-parallel loops. This support is in the early stages and is limited in the data types and operations that can occur in the body of the loop, although more advance techniques are under development. This development can be expected to follow that of Fortran loop manipulation and vectorization.

Intel's Fortran and C++ compilers provide automatic loop vectorization targeting the MMX, SSE [34], and SSE2 extension sets.

Metrowerks' CodeWarrior compiler provides vectorization for Intel's MMX and also for AMD's 3DNow! extensions [20]. Metrowerks is now owned by Motorola, so one would expect that support for Motorola's AltiVec extensions would be forthcoming. According to [13] this support is currently under development.

Green Hills Software's [26] C/C++ compiler supports AltiVec via automatic vectorization of loops.

Codeplay's VectorC{PC} C/C++ compiler performs automatic vectorization for MMX, 3DNow!, Enhanced 3DNow!, SSE, and SSE2 targets [35]. A separate version of this compiler targets the vector units of the Sony PlayStation2 [25].

The Portland Group's Workstation compilers for Fortran 77 [36], Fortran 90, C, and C++ [37] use a common core which supports automatic vectorization of loops for SSE-based targets.

Veridian Systems VAST/Parallel restructuring Fortran and C/C++ preprocessors [38] perform automatic loop vectorization and reordering as a front-end to a native compiler. Currently, these preprocessors only target the AltiVec multimedia extension set.

The VAST preprocessors have a long history, dating back to the mid-1980s when the Vector and Array Syntax Translator by Pacific Sierra Research Corporation was used to vectorize Fortran 200 code for the CDC Cyber 205 [39].

The VAST-F/AltiVec Fortran Preprocessor [40] "replaces vectorized Fortran loops with calls to VAST-generated C functions containing vector instructions." The VAST-C/AltiVec C Preprocessor [41] "automatically replaces loops in C programs with inline vector extensions (as defined by Motorola)."

These preprocessors generate C code in a manner similar to that of the Scc SWARC compiler discussed later in this work, but depend on Motorola's modified version of the GNU C compiler discussed previously.

According to [13], Absoft is also working on automatic vectorization of loops for Apple's Velocity Engine implementation of AltiVec. Their Pro Fortran compilers for Mac O/S 9 [42] and PPC/Linux [43], however, support AltiVec via precompiled Fortran 90/95 intrinsics and optimized benchmark and application-specific libraries. Automatic vectorization is only supported for the PPC/Linux version, and seems to be supported via Veridian's VAST-F/Vector preprocessors.

The VSUIF project at the University of Toronto [44] was conducted in the mid-to-late 1990's to add support for vector microprocessors to the SUIF compiler [45]. The goal of this project was to provide a high-level language programming model for using these architectures.

This compiler vectorizes loop-oriented, high-level language code into assembly code for the target architecture. The original target was the Torrent architecture [46, 47] which was then under development at the University of California at Berkeley. The designers planned to target Sun's UltraSPARC with VIS afterward, and a separate research effort was underway to create a SPARC code generator for SUIF [48]. This was intended to provide the back-end for VIS targets.

At the time [44] was written, DeVries and Lee had achieved some success vectorizing moderately complex code. They were still working on the handling of breaks and a method of classifying functions to determine if they would affect the vectorizability of loops when called. This work was to be validated using the UCB Torrent simulator before work to target the UltraSPARC was to begin.

We are unaware of the ultimate disposition of this work, although DeVries' Master's thesis is based on its implementation and performance [49]. While this project was intended to provide high-level support for vector processing, including SWAR targets, it takes the loop vectorization approach and does not treat vectors as first-class objects.

Automatic Vectorization of Basic Blocks

A more general approach to automatic vectorization is to search basic blocks for code which can be parallelized via the use of multimedia extensions. This allows not only loops to be vectorized, but also unrelated scalar code. This approach is also more general than parallelizing code based on first-class vector objects, because the statements which are automatically combined into vector fragments are not necessarily related.

Thus, this method is able to exploit a larger amount of parallelism than any other discussed. However, as with loop-vectorizing compilers, a compiler which vectorizes basic blocks is placed in the position of having to detect parallelism which is not explicitly described in the high-level language. This complex task requires a significant amount of time and space, more so than any other method of parallelization discussed here.

There are two groups known to be performing research in this area. The first is at the Massachusetts Institute of Technology's Laboratory for Computer Science. The other is at the University of Dortmund.

Work at MIT's Laboratory for Computer Science centers around what they term *Superword Level Parallelism* (SLP) [50]. This is defined as "short SIMD parallelism in which the operands and results of SIMD operations are packed in a storage location" [51].

The goal is to vectorize high-level sequential code throughout a basic block by detecting sets of single-valued *isomorphic statements* (statements which have the same expression structure) and collecting them into a series of vector fragment operations. This "SLP algorithm" is proposed as an alternative to the vectorization of looped code. In fact, the SLP compiler unrolls loops in order to generate isomorphic sequential code that can be parallelized in this manner. The SLP detection algorithm is described in [51] and elaborated on in [52].

A later report [53] presents a simplified alternative to the SLP vectorizing algorithm; however, this algorithm exploits only a subset of the parallelism that the SLP detection algorithm can. Results presented in this report were based on the percentage of dynamic instructions eliminated from sequential benchmarks. These were calculated for the 128-bit AltiVec architecture and for larger hypothetical architectures via SUIF. Apparently, no actual timing information was gathered.

This project is based on the vectorization of pre-existing sequential code which may be marked-up with compiler hints to indicate the presence of hard-to-detect parallelism. As such, it does not conform to the SWAR vector programming model. However, it is probably a good complement to the SWAR model in that it seeks to find parallelizable expressions which are more general than SWAR vectors. Conceptually, one could fragment vector and array code, and apply the SLP detection algorithm to extract parallelism from the remaining scalar code.

Work at the University of Dortmund centers around "code selection" for media and embedded processors. The goal of this work is similar to that of the MIT group.

A compiler technique introduced in [54] and briefly described in [55] uses a data-flow graph (DFG) as an architecture-independent intermediate representation of a high-level language (i.e. C) source. This DFG is then walked using a pattern-matching algorithm which pre-assigns instructions to the parts of the tree. Branches which can be covered by a single one of the target's SWAR instructions are tracked. When the entire graph is covered, instructions are actually assigned with the use of SWAR instructions maximized.

The authors seem to be unaware of similar work performed in the parallel processing area. In [54] it is claimed that "SIMD instructions are so far not really exploited by compilers for media processors. Taking advantage of such instructions is only possible, if processor-specific assembly routines or compiler intrinsics are used, resulting in low portability of software." This is despite the fact that the Scc compiler for the target-independent SWARC language was freely available for about two years before these papers were published and contemporary compilers such as Metrowerks' CodeWarrior [20] were capable of performing automatic vectorization of simple C language loops for multimedia-based targets.

Languages with First-Class Vector Objects

Languages which provide first-class vector objects allow multi-fragment objects to be defined and operated on as a single entity. This has several benefits. First, it allows the programmer to express vector operations in a more concise manner than inline assembly, fragment-based types and classes, or automatically vectorized scalar code. Second, it allows portability between architectures by hiding their differences, such as supported partitionings and register sizes, from the programmer. Third, it allows the compiler to deal with issues such as code optimization rather than parallelism detection.

Existing compilers for languages which support first-class vector and array objects, such as Fortran 90, have been targeted to architectures which have multimedia extensions, but it is not clear that any of these convert first-class vector or array operations into multimedia instructions. For example, the literature for the Veridian Systems VAST-F/AltiVec Fortran preprocessor [40] never mentions any such support although loop vectorization is discussed.

We are aware of only one other research effort which specifically takes this approach to supporting SWAR architectures. This is the Vector Pascal project at the University of Glasgow. Vector Pascal [56] is an extension of the Pascal language to support first-class operations on vector and array objects targeted to multimedia-enhanced architectures.

In Vector Pascal, unary and binary operations can be performed on complete arrays or their subsections. Certain higher-level functions, such as `sqrt`, `abs`, and `sin`, are intrinsic to the language and can also operate on these objects.

Binary operations include modular and saturated addition and subtraction, other modular arithmetic operations such as multiplication, division, and exponentiation, and various other types of operations such as comparisons, shifts, and logicals. These operations assume an implied identity value if one is not given. This applies to operations on set expressions as well as numeric ones. For example, the Vector Pascal expression `/a` is equivalent in meaning to the expression `1/a` for any value `a`.

For each of the binary operators there is an associated reduction operator. This applies the binary operation along the last dimension of its operand. These reductions reduce the rank of the operand by one with the exception of the scalar case in which they have no effect.

Objects of different rank can be operated on in mixed expressions with the restriction that, except for reductions, each variable in the expression must have rank less than or equal to that of the lvalue to which the expression's value will be assigned. Operands which have lower rank are replicated to match the rank of the lvalue. Operands of higher rank must be reduced in rank via one or more reduction operations.

User-defined functions which operate on a scalar object are automatically extended to apply to an array object of the same type in an element-wise manner. This mechanism allows the programmer to write functions that operate on both scalars and arrays of various sizes without having to parameterize the dimensions of its formal parameters.

One important aspect of SIMD programming that appears to be missing from Vector Pascal is the proper handling of parallel objects in the language's control constructs. No mention is made concerning if, or how, conditional constructs such as `if` statements and loops are handled when their conditional expressions are non-scalar. This is a significant issue which should be addressed in the design of a high-level SIMD language.

The Vector Pascal compiler uses the ILCG [57] code generation system in which a target description language is used to denote the specifics of the target architecture. The initial targets were the Intel 486 and Pentium with MMX. Currently Vector Pascal targets the "Intel 486, Pentium with MMX, and P3 and also the AMD K6." [56] It should be noted that these are all IA32-based architectures.

1.2.5 Summary

Software-only methods, such as Spieth's, cannot compete with those which take advantage of available SIMD instruction set extensions and can thus be rejected in most cases. These methods do, however, provide a level of portability between targets which cannot currently be obtained using multimedia extension sets only.

Low-level, high-performance libraries are closely related to their target architectures. These are often written to be inlined by a compiler and can thus be easily optimized. However, they do not provide portability between architectures and are thus insufficient for our model.

High-level libraries tend to be application-specific, intended to perform particular algorithms or operations for well-known problems. While typically having reasonably portable interfaces, these libraries are not intended for use in general-purpose algorithms and are usually too specialized for our purposes.

As a general rule, high-performance in library code comes at the price of non-portability. Thus, it is difficult, but not impossible, to develop a portable, high-performance, general-purpose library. Developing such a library would entail making a trade-off between these two competing factors.

Inline assembly and compiler intrinsics are directly related to their associated architectures, and thus operate at too low a level to be considered for a portable general-purpose programming model. However, they can be useful for code generation as they tend to ease the integration of unsupported hardware instructions into preexisting compilers.

Classes and types which represent a word-sized fragment of vector data also operate at too low a level to be considered for a general-purpose programming model. These are directly related to their associated hardware architectures, encoding the size of their registers, and often only provide access to the available hardware instructions. Thus, they generally do not present a portable programming model. This is not to say that classes and new types cannot provide a portable level of abstraction, only that current systems tend not to use these methods to their best advantage.

Compilers which perform automatic vectorization of scalar loops and basic blocks tend to be overly limited in their current capabilities. Most of the current set of vectorizing compilers are only capable of vectorizing simple loops that would be more succinctly expressed as first-class vector operations. More complex loops, those that cannot be expressed as vector operations, are typically too complex for these compilers to handle.

As current compiler writers learn more about, or reinvent, the work done in the high performance computing community over the last few decades, these compilers will become better at generating vectorized code from scalar sources. However, we should be developing programming models that make it easier to express complex operations, not high-performance compilers which optimize source code based on the wrong architectural model.

As part of the development of a new general-purpose SWAR programming model, the subject of this thesis, we have chosen to design a language with first-class vector objects because we believe this offers the best opportunity for performance gains over a large range of applications and target architectures.

Unlike any of the related work, this language allows both the precision of the data and the number of elements to differ from those supported by the hardware. It also provides a full, portable set of vector operations which are independent of the extended instructions available on any particular target. This language, SWARC, will be discussed later in this work, and is, to the best of my knowledge, the only language which adheres to this generalized model.

1.3 Scope of Work

In this thesis, a new abstract model of parallel computation is developed which better reflects the capabilities and limitations of modern SWAR architectures than do current computational models. An example language based on this model is presented, as is a compiler for this language which uses various techniques to optimize code for these architectures. Performance metrics are also developed and employed to evaluate these implementations. This work should provide a starting point for future research and the development of practical programming languages for SWAR processing.

1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 is a study of the multimedia extension sets available in commodity general-purpose microprocessors. Chapter 3 presents the general-purpose SWAR processing model. Chapter 4 describes the SWARC language which is based on the SWAR processing model and a proof-of-concept implementation of a SWARC compiler called Scc. Chapter 5 presents various evaluations of the defined SWAR model, the SWARC language, and the Scc compiler.

2. ANALYSIS OF CURRENT MULTIMEDIA EXTENSION SETS

A new abstract model of parallel computation is needed which will better reflect the capabilities and limitations of modern SWAR architectures than do current computational models. In order to develop a new model which adequately accounts for the capabilities and limitations of current SWAR architectures, it is necessary to have an understanding of the range of functionality which they support.

These architectures were created when commercial developers of microprocessors redesigned them to improve their performance on multimedia applications. This was done by extending their standard instruction sets with new sets of "multimedia instructions" which operate in a SIMD manner on parallel sections of their system data paths.

Each extension set was tailored to support the algorithms and applications which its designers believed to be most important to their clientele. Early extensions tended to be limited to instructions which perform operations that are frequently used in their particular target applications, and were not intended to present a complete parallel programming model to their users. Thus they failed to provide sufficient support for a viable SWAR processing model.

Because of the variation in their applications, the extensions meant to support them varied widely. However, some of these applications differ only in scope or quality, with the underlying algorithms being equivalent. Consequently, while each extension set is unique, its functionality may have aspects which are similar or equivalent to those of other extensions.

Later extensions are more complete, often including improvements which address problems with their ancestors' designs. Thus, a type of evolution is in play which may ultimately lead to relatively stable and complete sets of multimedia instructions. Unfortunately, current extension sets still have limitations.

The range of support provided by these extensions still varies widely. The scope of these extensions also differs, with some including a large number of SWAR operations, while others include only a few. Support is still limited to data of standard sizes, and is still not consistent across these sizes. Also, instructions necessary for proper SIMD operation are often lacking or limited.

The primary goal of this phase of research was to determine the capabilities and limitations of the multimedia extension families which are available on current COTS (commodity, off-the-shelf) processors [4]. This analysis will be used as a basis for the design and implementation of the general-purpose SWAR programming model undertaken in later phases of the research. This is necessary to ensure that the developed model fairly reflects the common capabilities of current architectures.

This analysis should also be useful when deciding how an architecture's enhancements will be used within an implementation of the generalized model, and should foster insight into the possibility of code optimization based on a target architecture's enhanced features.

Data collection and organization was carried out over the last few years by myself. The data is derived primarily from programming manuals pertaining to the various extension sets and their related architectures. Other sources of information included journal articles and promotional literature, but manuals were used whenever possible as they are generally the most reliable sources.

An early survey of multimedia extensions was presented by Kelley and Postiff in [58]. That paper also discusses issues related to the circuit implementation of multimedia extensions. A limited table of multimedia extensions was presented by Dubey in [59]. This was apparently developed at about the time of my thesis proposal [4], but I was unaware of it until recently. Unless noted, neither of these was used as a source of information for the following analysis.

In this section, several current extension sets are briefly introduced, along with some older ones which have interesting features. In the following section, a set of tables is presented which describe the SWAR instructions available to programmers using these extension sets.

The multimedia extension sets analyzed in this chapter are: Digital Equipment Corporation's Motion Video Instructions (MVI) [60]; Hewlett-Packard Company's PA-RISC 1.1 Multimedia Acceleration Extensions (MAX-1) [61], and PA-RISC 2.0 Multimedia Acceleration Extensions (MAX-2) [62, 63]; Silicon Graphics MIPS-V [64] and MIPS Digital Media Extension (MDMX) [65, 66]; Motorola, Incorporated's AltiVec [67, 68]; Sun Microsystems, Incorporated's Visual Instruction Set (VIS) [69, 70]; Intel Corporation's [71, 72] MMX, which is also implemented by Advanced Micro Devices, Incorporated [73] and Cyrix Corporation [74]; AMD's 3DNow! [75], Enhanced 3DNow! (E3DNow!) [76], and 3DNow! Professional (3DNow!Pro); Cyrix's Extended MMX (EMMX) [77]; and Intel's Streaming SIMD Extensions (SSE) [78] and Streaming SIMD Extensions 2 (SSE2) [78].

MVI

The Motion Video Instructions (MVI) were originally developed by Digital Equipment Corporation for their Alpha microprocessor architecture in about 1996. This was "motivated by the desire to perform high quality software motion video encoding using the prevalent ISO/ITU video compression standards." [79].

MVI was clearly not an attempt to develop a high-level SWAR programming model, and is in fact more closely related to the graphical extensions included in the Intel i860 or Motorola 88110 processors than to other extensions studied.

MVI consists of a minimal set of instructions that perform graphical operations such as calculating pixel differences and finding the larger or smaller of two values. These instructions operate on data residing in the Alpha's standard 64-bit integer register set. This makes the standard integer instructions available to the SWAR programmer.

Digital was bought by Compaq Computer Corporation, which was subsequently bought-out by Hewlett-Packard. The Alpha architecture and the MVI extensions have been passed along as well.

PA-RISC MAX-1.0

The original version of Hewlett Packard's Multimedia Acceleration eXtensions (MAX-1.0) were intended to accelerate the decompression of video data for real-time display without resorting to special-purpose hardware.

The basic design process was described by chief architect Ruby Lee as "...finding the most frequent operations, breaking them down into simple primitives, and accelerating their execution." [61] This process resulted in a small set of general-purpose instructions which performed basic arithmetic operations, and allowed these extensions to be used for purposes beyond those for which they were designed.

MAX-1.0 was originally implemented on the 32-bit PA-RISC 1.1 architecture PA-7100LC [80, 61] which was introduced in 1994. Primitive arithmetic and shift-and-arithmetic operations were performed by the 7100LC's two integer ALUs on the 16-bit subwords of the processor's 32-bit integer registers. This allowed two MAX instructions to be executed with every clock cycle at peak speed.

MAX-1.0 was superseded by the MAX-2.0 extension set with the introduction of the PA-RISC 2.0 architecture. In each of the tables, these are combined under the MAX heading unless there are instructions which are only in MAX-2.0. In this case, the there is a column for each of the two versions, and those listed in MAX-1.0 are available in both.

PA-RISC MAX-2.0

As with MAX-1.0, Hewlett-Packard Company's MAX-2.0 extensions [62] were designed to accelerate multimedia processing without using special-purpose hardware. MAX-2.0 was developed with the goal of introducing "instructions that provide significant performance improvement with insignificant impact on the area, cycle-time, and design time of the PA-RISC processor." [81] A good description of the thoughts of the HP designers can be found on page 1-6 of [82].

MAX-2 was first implemented on the 64-bit PA-8000 microprocessor [83, 84] in 1995 and is considered to be an integral part of the PA-RISC 2.0 architecture [82]. It is a superset of MAX-1 which it extends to support 64-bit architectures and instructions for controlling data alignment and layout. These include simple parallel shifts, "mix" instructions which interleave the fields of two operands, and an instruction which permutes the fields of a register. These instructions were chosen to significantly accelerate media processing while still being useful for general-purpose processing [63].

MAX-2 uses the integer general registers, integer ALUs, and shift merge units (SMUs) of the PA-8000. The two integer ALUs are similar to those of the 7100LC. The two SMUs perform basic parallel shifting operations, the merging functions which interleave two operands, and the generalized permute operation. This allows up to four MAX-2 instructions to be executed simultaneously. The integer pathways were chosen to minimize the amount of modification required and allow the use of preexisting integer instructions such as extractions.

MAX-2 is currently available in PA-RISC 2.0-based servers such as HP's rp8400 series. With Hewlett-Packard's acquisitions of Compaq and Digital, and the recent move toward support for Intel-based systems, the future of the PA-RISC architecture, and thus MAX, is in question. It remains to be seen if they will continue to be supported.

MIPS-V Paired-Single

The MIPS-V instruction set adds support for partitioned operations on pairs of single-precision floating-point data to the MIPS-IV instruction set. Pages 7-10 of [66] contain an overview of these extensions, and detailed descriptions of the instructions are provided in [64].

This extension set was intended to support applications related to graphics and signal processing, such as "3D [*sic*] geometry processing, oil and gas, and manufacturing applications." [85] It does this via a reasonable set of floating-point arithmetic instructions, a rich set of conditional tests, and data alignment and layout operations. This makes the MIPS-V "paired-single" extensions useful for a variety of applications.

MIPS-V was announced in 1996 [86], and was to be introduced with the H1 generation of processors following the R12000. These were scheduled for production in the first half of 1999 [85]. At some point, MIPS changed its focus to the development of processor cores for application specific markets, and the architectures were reorganized. It is not clear to me if MIPS-V was ever actually implemented as a standalone entity. The current MIPS64 architecture is MIPS-V compatible; however, the paired-single extensions are an optional feature [87].

MDMX

The MIPS Digital Media Extension (MDMX) was announced at the same time as the MIPS-V paired-single extensions [86]. It was intended to provide support for "video, audio, and graphics pixel processing by introducing vectors of small integers." Pages 11-19 of [66] contain an overview of these extensions and detailed descriptions of the instructions are provided in [65]. MDMX is one of several "Application Specific Extensions" to the MIPS-V architecture. Its presence implies availability of the MIPS-V paired-single extensions.

In regards to general-purpose parallel processing, a paragraph from the MIPS Digital Media Extension definition [65] is telling:

> The MIPS MDMX is not intended for general purpose computing. Software support for the MDMX is via shared libraries (DSOs) and assembly language only. Compiler support is neither implied nor planned.

One of the unique features of MDMX is a 192-bit “accumulator”, which is primarily used as the target for repetitive applications of cumulative instructions. It is divided into fields which are three times as wide as the data being operated on. For example, for a data size of 16-bits the accumulator consists of four fields of 48-bits each.

Another of MDMX’s strengths lies in the variation it allows for the second source vector of its instructions. Almost all MDMX instructions allow this source to be a partitioned register, an immediate value, or a scalar which the instruction replicates. This allows a single immediate or field value to be “broadcast” to each of the fields, and also allows mixed operations between partitioned values and scalars. Thus, this feature makes MDMX quite versatile.

As with the MIPS-V paired-single extensions, MDMX was to be implemented in the H1 generation of MIPS processors [85]. However, it is not clear to me that MDMX ever was actually implemented, although similar instructions exist in the MIPS-64 [87] and MIPS-3D [88] architectures. Its unique qualities make MDMX worth studying in any case.

MIPS-3D

The MIPS-3D graphics extension to the MIPS64 architecture was introduced sometime around the year 2000. It is an application-specific extension “intended for 64-bit consumer applications that need three-dimensional graphics but require minimal implementation costs for low-power or System-on-Chip (SOC) solutions.” [88] As an extension, MIPS-3D is implemented as an optional core that can be incorporated into an application-specific processor design.

MIPS-3D uses the MIPS64 floating-point unit and operates on “paired-single” floating-point data. It consists of 13 instructions for absolute value calculation, advanced arithmetic operations such as reciprocal approximation, reductions, data conversion, and aggregate conditionals.

Having only learned of this extension recently, I have decided not to discuss it to any significant depth at this time. However, by adding support for reductions and aggregate conditionals, it address two of the primary deficiencies in current SWAR extensions.

AltiVec

Motorola Incorporated’s AltiVec [68] extension to the PowerPC architecture was developed in the late 1990s and incorporated into the MPC7400 processor [89] in 1999. It was developed to support high-performance computing and high-bandwidth networking applications such as array processing, Internet routers, and video processing systems [67].

AltiVec includes integer and floating-point SWAR instructions. These are executed by a special-purpose vector processing unit which operates on data stored in a set of 32 128-bit vector registers. Its completeness and ability to operate on both integer and floating-point data make AltiVec one of the better designed extension sets from a parallel processing stand-point.

The PowerPC architecture was jointly developed by Motorola, Apple Computer Incorporated and International Business Machines Corporation. However, Motorola has been the primary developer of AltiVec, with Apple a major consumer, and IBM declining to participate in the effort. AltiVec is a well-defined, general-purpose set of extensions which is likely to have continued use in high-performance and embedded systems in the future.

VIS

Sun's VIS [70, 69, 90] instruction set was intended to support networked applications such as video conferencing, data encryption, and collaborative software and also scientific applications such as systems modeling and image processing.

VIS is best suited to handling 16- and 32-bit data, with some support for 8-bit pixel data. The instructions included tend to be special-purpose and limited in the data precisions supported. For example, a fairly large set of multiplications is available, but these are all mixed-precision operations that operate on 8- and 16-bit operands. By contrast, there is no support for the addition or subtraction of 8-bit data at all.

One of the design goals for VIS was allow good data flow between memory and the floating-point registers. This is supported with a reasonable set of loads and stores including block accesses and masked stores. These improve throughput and support SIMD processing. This may be VIS's greatest strength.

VIS was implemented in 1995 with the 64-bit, first-generation V9 architecture UltraSPARC-I processor TrGrNo:95. The UltraSPARC-I had a single pair of fully-pipelined graphics add and multiply units. VIS was subsequently implemented in the UltraSPARC-II, a second-generation V9 processor with two floating-point/graphics units [91].

A somewhat extended version, referred to as VIS 2.0 is available in current processors such as the UltraSPARC III Cu [92]. The version discussed in this thesis is now called VIS 1.0.

MMX

The MMX extension set, was designed by Intel Corporation and introduced in 1996 in later Pentium (Pentium with MMX) processors [93, 71]. MMX was cloned by Advanced Micro Devices, Incorporated [73], Cyrix Corporation [74], and others such as Rise Technology Company [94].

It was originally "...designed to enhance performance of advanced media and communication applications" [72] while retaining "full compatibility with existing operating systems and software." [93] An overview of the MMX extensions is provided in [72], and detailed descriptions of the instructions are available in [95]. A short summary, including cycle counts, is available in [93].

MMX operates on integer data stored in the CPU's floating-point (FP) registers. These cannot be used for floating-point operations while MMX is in use. Also, the IA-32's standard integer instructions cannot be used on the data stored in these registers. In this sense, MMX is less useful than extensions which operate on their standard integer registers.

The MMX extensions provide a fairly wide range of support for a high-level parallel programming model; however, they are limited to 8-, 16-, and 32- bit SWAR operations which are not implemented consistently across these field sizes. There are also no reduction operations nor minimum or maximum instructions which could be used for emulating unsupported saturation arithmetic operations. Despite these limitations, MMX is one of the more complete sets of SWAR extensions and has become a permanent feature of Intel IA-32 architecture processors with a large number of other extensions built on top of it.

SSE

Intel's Streaming SIMD Extensions (SSE) [78] serve two purposes. First, they fill in some of the missing pieces of MMX. Second, they add a set of 32-bit floating-point SWAR instructions which operate on a new set of eight 128-bit registers. With these extensions, the Intel architecture is divided into three sections: the basic IA32 architecture, the integer SWAR MMX, and the floating-point SWAR SSE.

SSE is very complete, but lacks 64-bit support and leaves the Intel IA-32 architecture with two different SWAR register sets for different types of data. In this respect

AltiVec is better, and has more registers to work with. However, SSE has better memory handling and the ability to move data between registers.

SSE was introduced with the Pentium III architecture in 1999 and continues to be part of the IA-32 architecture.

SSE2

Intel's Streaming SIMD Extensions 2 (SSE2) is a set of integer instructions primarily intended to provide MMX equivalent functionality to data stored in the 128-bit SSE register set. SSE2 also includes 64-bit floating-point extensions to SSE and various integer extensions to MMX. These are intended to fill-in gaps in the earlier extension sets to make them more complete.

Combined, SSE and SSE2 form the most powerful set of SWAR extensions currently available. They allow both integer and floating-point data to be stored and operated on in the same register set. This, and their comprehensive support for data of standard precision, places the SSE/SSE2 pair on par with Motorola's AltiVec extensions.

SSE2 was implemented with Intel's Pentium 4 (previously code-named Willamette [96]), and is now a permanent feature of the IA-32 architectural line. The future of SSE2 depends on whether this 32-bit line of processors remains viable given Intel's development of the IA-64 architecture and on the extent to which its functionality is incorporated into this newer architecture.

3DNow!

AMD's 3DNow! [75] expands the MMX instruction set by filling in some of its gaps and by including a set of 32-bit floating-point instructions. This was intended to support "floating-point-intensive and multimedia applications", and was expected to improve frame rates for high-resolution graphics, modeling of physical environments, three-dimensional imaging, and video and audio playback quality.

3DNow! uses the same registers as MMX. This allows mixed-mode expressions to be evaluated easily. It also allows the MMX polymorphic operations to be applied to floating-point data for masking or extraction purposes.

3DNow! adds basic arithmetic, comparison, and maximum/minimum operations for floating-point data, as well as more advanced mathematical operations such as reciprocals and square roots. It also includes instructions for converting between integer and floating-point formats and instructions for cache prefetching.

3DNow! was first implemented on the K6-2 processor in 1998, a two-pipeline processor with separate MMX ALU units, but shared 3DNow! resources. It has subsequently been implemented on the K6-III and current Athlon processors.

Enhanced 3DNow!

AMD's Athlon extensions to 3DNow! and MMX [76], which we will refer to as Enhanced 3DNow! or E3DNow!, was intended to provide better support for DVD-quality audio and video streaming and digital signal processing than did these earlier extension sets.

E3DNow! fills gaps in the MMX and 3DNow! extension sets. It extends 3DNow! with a few instructions for floating-point accumulation, type conversion, and double-word swaps. It extends MMX with a large set of instructions. These perform various arithmetic operations, cache-bypassing stores for streaming purposes, and store synchronization, word layout manipulation, and advanced prefetching operations.

E3DNow! was first implemented on the Athlon processor [97] in 1999 and continues to be implemented on current AMD architectures.

3DNow! Professional

AMD's 3DNow! Professional [98] was designed primarily to synchronize AMD's multimedia extensions with Intel's SSE, and thus ease code migration between these competing architectures. As with AMD's other multimedia extensions, 3DNow! Professional is implemented on the MMX registers and data path.

The use of the Athlon's MMX register set means that, unlike the Intel IA32 architecture, the AMD architecture does not require support for the same set of operations to be implemented for two separate register sets. All of the SWAR instructions added to the AMD architecture are available for use with its single enhanced register set.

On the other hand, the AMD architecture does not have the potential for parallelism that the Intel architecture has with its separate MMX and SSE data paths and register sets. Thus, while it may be more difficult to program the Intel architecture for optimal performance, the potential pay-off may be higher, depending on the number of pipelines available.

3DNow! Professional was to be implemented in certain Palomino-core Athlon processors starting in 2001. These included the desktop Athlon MP, but apparently not earlier mobile Athlon 4 processors (or at least, not the one in my notebook computer). 3DNow! Professional is currently implemented in Thoroughbred-core Athlon XP processors [99] released starting in the first half of 2002. 3DNow! Professional can be expected to be included in Athlon MP and XP line processors for the foreseeable future.

Extended MMX

Cyrix's Extended MMX (EMMX) [77] was intended to extend the MMX extension set in two ways. First, it extended MMX's functionality by including arithmetic instructions such as average, magnitude, and multiply high in order to make it more generally useful. Second, it added flexibility by including "implied destination" instructions.

Implied destination instructions target a register whose use is not explicitly indicated in the instruction, but rather implied by the use of its sequentially paired register. Each pair consists of the registers whose numbers differ in only the least significant bit position. Effectively, these instructions are three register instructions rather than the IA32 standard of two. This allows the instruction to avoid overwriting one of its sources.

According to [77], EMMX was implemented on the MII processor. The GXm was also intended to support EMMX according to a preliminary version of the Cyrix CPU Detection Guide [100]. Unfortunately, EMMX was phased out at about the time of Cyrix's acquisition by National Semiconductor Corporation.

In 1999, Cyrix was sold to VIA Technologies, Incorporated. The Cyrix MII is listed as a current VIA product [101]; however, it apparently has been supplanted by the VIA C3, a 1GHz processor which supports MMX and 3DNow! [102]. This processor was formerly known as the VIA Cyrix MIII [103].

2.1 Tables of Multimedia Extension Support for SWAR

The following tables contain information about the extension sets studied. This information was gathered from various sources, but was primarily taken from specifications in architectural and programming manuals.

In general, the description and tabulation of each extension set includes only those instructions that are part of that extension set and not those that are part of the underlying architecture or extension sets. For example, instructions that are included in MMX are not listed as being part of SSE, although in current architectures support for SSE implies support for MMX.

Exceptions have been made for extensions which operate on data that resides in the general register set of the underlying architecture. In this case, existing instructions that may be useful for SWAR processing have been included. Specifically, the descriptions for DEC's MVI and HP's MAX-1 and MAX-2 extensions include standard integer instructions which can be usefully applied to partitioned data stored in the integer registers on which these extensions operate.

For this analysis, the instructions have been categorized into groups which perform related types of operations. These include arithmetic instructions, shifts and rotations, bitwise-logical and bitwise-reduction instructions, various types of conditional instructions and instructions which support control flow, data movement, replication, and type conversion instructions, various types of data layout instructions, memory accesses, and cache management instructions.

Some explanation of the notational conventions used within the tables is required before the tables themselves are presented. These conventions are intended to allow the data in these tables to be described concisely. Periods have been left off from the abbreviations used in order to minimize the amount of space used.

In the row headings of the tables, the abbreviation "Part" indicates a partitioned operand, "Scalar" indicates a partitioned operand with identical field values, "Element" indicates one field of a partitioned operand, "Single" indicates a partitionable register taken as a single unpartitioned value, and "Immed" indicates an immediate operand contained in the instruction itself.

Also in the row headings, the abbreviation "Acc" denotes the use of a separate accumulator, with "Acc Init" indicating that the operation will clear the accumulator first. "Acc" by itself indicates that the result of the operation will be added to the value in the accumulator. "Acc Diff" indicates that the operation will find the differences between the operands, then add these differences to the accumulator. The notation "Acc Sub" indicates that the result of the operation will be subtracted from the accumulator.

Within the body of the tables, the notation "NxB" indicates an operand or result partitioned into N fields of B-bit integers which may be signed or unsigned. A trailing "u" indicates that the field data is treated as unsigned, and a trailing "s" indicates that it is treated as signed.

Where such an entry is listed by itself or in a comma-separated list of values, it indicates that a form of the operation where both the operands and result have the listed partitioning is supported by the extension set. Where an entry contains an arrow, the notation shows the form of the operands separated by an operator, followed by the arrow and then the form of the result.

The first table contains architectural information about representative CPUs which implement these extensions. The remaining tables describe the forms of the instructions contained in each set. In most cases, separate entries have been made for each instruction. These tables are keyed to the similarly numbered tables in Appendix C which list the corresponding instruction mnemonic for each entry.

2.1.1 Sources and Architectural Features

Table 2.1 lists the primary sources of information and the architectural parameters of a representative processor for each of the enhanced architectures.

For each extension set, the primary source of information contained in this and the following tables is indicated in the row labeled "Primary Source". Data for each extension set was taken from the listed primary source unless otherwise noted.

The rows labeled "# R/W MM Registers" indicate the number of read/write registers available for use by the corresponding multimedia extension set. Those labeled "# R/O MM Registers" indicate the number of read-only registers available for use. Some architectures reserve register 0 for use as a fast means of obtaining a constant zero value and do not allow this register to be written to.

The rows labeled "# Bits/MM Register" indicate the total number of bits that can be stored in a single register used by the corresponding extension set. This ultimately limits the amount of SWAR parallelism that can be obtained within a single multimedia pipeline.

The next row indicates which of the corresponding architecture's register sets are used by the multimedia extension set. Multimedia extensions usually operate on data in modified existing processor registers, but some use register sets that have been added expressly for use by the resident extension set. Those that are implemented using existing registers have the advantage of being able to make use of existing instructions, while those that are implemented using dedicated register sets typically have fewer restrictions on their individual use.

In some cases, multiple sets of registers are used, depending on the specific instruction applied. For example, SSE includes instructions that operate on data in the SSE-specific register set, and also instructions that operate on the MMX-specific set. Note that only DEC's MVI and HP's MAX extensions are applied to their respective general integer register sets.

The next row indicates the maximum number of memory operands that may be accessed by instructions that are not specifically intended for memory access purposes. Note that the extensions based on the Intel IA32 architecture allow memory operands for most instructions while those based on RISC architectures do not. Because of this, we will not differentiate between register and memory operands when discussing Intel IA32-based extension sets unless necessary. Also note that any particular instruction may use a different number of memory operands than the maximum.

The row marked "Maximum Source Operands" indicates the maximum number of source operands that may be used by an instruction in the corresponding extension set. This is generally inherited from the underlying architecture. Any particular instruction may have a different number of source operands than the maximum.

The next row indicates whether or not one of the source operands will be overwritten by the result of a typical instruction in the extension set. If reused, these operands will have to be copied before the overwriting instruction is applied. Architectures which allow non-source destinations help the programmer to avoid this problem as long as there are available registers.

2.1.2 Arithmetic Instructions

Tables 2.2 through 2.7 show groups of arithmetic SWAR operations including addition, subtraction, minimum, maximum, multiplication, combined operations, division, and more advanced arithmetic operations. Each table is described in turn.

Table 2.1
Comparison of Multimedia Instruction Set Extensions

Architectural Feature	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX	Motorola AltiVec
Primary Source	[60]	[61]	[82]	[64]	[65]	[104]
# R/W MM Registers	31	31	31	32	32 / 1[1]	32
# R/O MM Registers	1[2]	1[2]	1[2]	0	0[1]	0
# Bits/ MM register	64	32	64[3]	64	64 / 192[1]	128
Which registers?	Integer	Integer	Integer	Float	Float or Accumulator[1]	AltiVec Vector
Maximum Memory Operands[4]	0	0	0	0	0[1]	0
Maximum Source Operands[5]	2	2	2	3	3	3
Source Overwritten as Destination?	No	No	No	No	No	No

Architectural Feature	Sun VIS	Intel MMX	Intel SSE	Intel SSE2
Primary Source	[90]	[95]	[95]	[95]
# R/W MM Registers	32	8[6]	8	8
# R/O MM Registers	0	0[6]	0	0
# Bits/ MM register	64	64[6]	128	128
Which registers?	Float	Float[6]	SSE-specific or Float	SSE-specific or Float
Maximum Memory Operands[4]	0	1	1	1
Maximum Source Operands[5]	2	2	2	2
Source Overwritten as Destination?	No	Yes	Yes	Yes

Architectural Feature	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
	[75]	[76]	[98]	[77]
# R/W MM Registers	8	8	8	8
# R/O MM Registers	0	0	0	0
# Bits/ MM register	64	64	64	64
Which registers?	Float	Float	Float	Float
Maximum Memory Operands[4]	1	1	1	1
Maximum Source Operands[5]	2	2	2	2
Source Overwritten as Destination?	Yes	Yes	Yes	No for implied

[1]From [66].
[2]Reads as 0.
[3]From [61].
[4]Does not include load and store instructions.
[5]Does not include unique destination operand.
[6]From [105].

Addition Operations

Table 2.2 contains information on the various forms of addition available in the studied extension sets. These include modular and saturations addition, high-word results, and various reductions.

Modular addition, also known as *wrap-around addition*, is “normal” computer addition in which the stored result is the low n bits of the actual result, where n is the size of the space in which the result is to be stored. This is equivalent to taking the actual result modulo the maximum value storable in the available space. Each extension set includes some form of modular addition except for MVI, which does not, and the extensions to MMX, which use the MMX instructions for this purpose.

Most extension sets only allow the modular addition of two partitioned registers; although, as already indicated, those based on the Intel IA32 architecture also allow a memory location to be used as an operand. SSE, 3DNow!Pro, and SSE2 also contain instructions which modularly add together only the lowest element from each of two operands. By contrast, MDMX only allows modular addition to the accumulator — all other addition is saturated.

Because of its ubiquity and familiarity, modular addition should be included in any general-purpose SWAR programming model.

Saturation addition is a form of computer addition in which the result is set to the maximum storable value of the same sign when an overflow occurs. This form of addition is used primarily for multimedia applications in which the data value represents some physical parameter whose value should not wrap with incremental changes. For example, the volume level on an audio mixer should not suddenly drop to 0 when the user attempts to increase the volume above the maximum.

Again, most of the families support some form of saturation addition, but MVI, MIPS-V, VIS, E3DNow!, and SSE do not. On those architectures which do not support them, these operations can be often be emulated. One possibly method is to use a larger-precision addition, then limit the result to the values representable in the lower-precision form.

Saturation arithmetic is seldom used for numeric computation, but the saturation form of result is often more attuned to the needs of a numeric programmer than one might realize, and may be used more often in the future. Because it is reasonably available and can be relatively easily emulated on most architectures, saturation arithmetic should be included in any general-purpose SWAR programming model.

An NxB "modular addition high" (also known as "addition carry-out") zero-extends the carry bits that would result from a partitioned addition of the NxB addends and stores them in an NxB result. Only AltiVec has this operation, and thus it is not a good choice for inclusion in a portable model; however, it is useful for emulating other operations such as saturation addition.

"Saturation reduce-add with an element" (Sat. RedAdd with El.) performs a saturation addition of all of the fields of one partitioned register and the low field of a second partitioned register. That is it performs a *reduction addition* on the first partitioned register and also adds in the low field of the second. The result is stored in the low field of a third partitioned register whose other fields are zeroed.

Only AltiVec includes this operation. This is unfortunate because it can be used to optimize the implementation of reductions, which occur fairly frequently in SIMD algorithms and are often costly to emulate. Because of this, and despite the lack of support for reductions by other extension families, reductions should be included in a generalized SWAR model to facilitate traditional SIMD processing.

"Saturation partial reduce-add with even elements" (Sat. Part. RedAdd w/Even) performs a saturation addition on the N/2 sets of two neighboring fields of one partitioned register and the even element of the corresponding set of elements of a second partitioned register. The result is then stored in the even element of the corresponding set of elements of a third partitioned register whose odd elements are zeroed.

"Saturation partial reduce-add with a partitioned value" (Sat. Part. RedAdd w/Part) performs a saturation addition on the N/2 (or N/4) sets of two (or four) neighboring fields of one partitioned register and with the corresponding element in a second partitioned register. The result is then stored in the corresponding element of a third partitioned register.

The previous two instructions are only included in AltiVec, and are a bit too esoteric for general-purpose work. They are most likely to be used, if at all, as optimizations in the implementation of other operations.

"Saturation reduce-add and pack" (Sat. RedAdd and Pack) performs separate saturated reduction additions on the elements of each of the sources, then packs the sums into a partitioned result. This instruction is only included in 3DNow!, and would be most useful for optimizing the implementation of reduction operations.

"Saturation reduce-add/subtract and pack" (Sat. RedAdd/Sub and Pack) performs a saturated reduction addition on the elements of one of the sources and a reduction subtraction on the elements of a second source, then packs the differences into a partitioned result. These two instructions are only included in Enhanced 3DNow!, but may be useful for implementing reduction operations, depending on how they are defined in the programming model.

Subtraction Operations

Table 2.3 contains information of the various forms of subtraction available in the studied extension sets. These include modular and saturation subtraction, high-word results, and sums and reduced sums of absolute differences.

As with addition, modular subtraction is "normal" computer subtraction, in which the stored result is the actual result modulo the maximum value storable in the register. Each of the extension families include some form of modular subtraction except for MVI and the extensions to MMX, which again use the MMX instructions. For each family, the supported forms correspond to the supported forms of modular addition.

Table 2.2
SWAR Addition Operations

Operation Types	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX	Motorola AltiVec
Modular Addition[1]						
Part/Part	-	2x16	- 4x16	2x32f[2,3]	-	16x8, 8x16, 4x32
Immd/Part	-	-	-	-	-	-
Part/Part w/Acc (w/ or w/o Init)	-	-	-	-	2-8x8u→8x24s, 2-4x16s→4x48s	-
Scalar/Part w/Acc (w/ or w/o Init)	-	-	-	-	2-8x8u→8x24s, 2-4x16s→4x48s	-
Immd/Part w/Acc (w/ or w/o Init)	-	-	-	-	2-8x8u→8x24s, 2-4x16s→4x48s	-
Element/Element	-	-	-	-	-	-
Saturation Addition						
Part/Part	-	2x16s, 2x16u+2x16s →2x16u	4x16s, 4x16u+4x16s →4x16u	-	8x8u,4x16s	16x8s,16x8u, 8x16s,8x16u, 4x32s,4x32u,4x32f[4]
Scalar/Part	-	-	-	-	8x8u,4x16s	-
Immd/Part	-	-	-	-	8x8u,4x16s	-
Modular Add. High						
Part/Part	-	-	-	-	-	4x32u
Sat. RedAdd w/El.	-	-	-	-	-	4x32s+low 1x32s→low 1x32s
Sat. Part. RedAdd w/Even	-	-	-	-	-	4x32s
Sat. Part. RedAdd w/Part	-	-	-	-	-	16x8s+4x32s→4x32s, 16x8u+4x32u→4x32u, 8x16s+4x32s→4x32s
Sat. RedAdd and Pack	-	-	-	-	-	-
Sat. RedAdd/Sub and Pack	-	-	-	-	-	-

[1]Modular signed and unsigned addition are equivalent.
[2]Calculated to infinite precision, then rounded according to current rounding mode in FCSR.
[3]Generates exception on overflow or underflow.
[4]Rounds to nearest.

Table 2.2 cont'd.
SWAR Addition Operations

Operation Types	Sun VIS	Intel MMX	Intel SSE	Intel SSE2
Modular Addition[1]				
Part/Part	2x16,4x16, 1x32,2x32	8x8, 4x16, 2x32	4x32f[2]	16x8 8x16 4x32 2x64,2x64f
Immd/Part	-	-	-	-
Part/Part w/Acc (w/ or w/o Init)	-	-	-	-
Scalar/Part w/Acc (w/ or w/o Init)	-	-	-	-
Immd/Part w/Acc (w/ or w/o Init)	-	-	-	-
Element/Element	-	-	low 1x32f[2]	low 1x64f
Saturation Addition				
Part/Part	-	8x8s,8x8u, 4x16s,4x16u	-	16x8s,16x8u, 8x16s,8x16u
Scalar/Part	-	-	-	-
Immd/Part	-	-	-	-
Modular Add. High				
Part/Part	-	-	-	-
Sat. RedAdd w/El.	-	-	-	-
Sat. Part. RedAdd w/Even	-	-	-	-
Sat. Part. RedAdd w/Part	-	-	-	-
Sat. RedAdd and Pack	-	-	-	-
Sat. RedAdd/Sub and Pack	-	-	-	-

[1]Modular signed and unsigned addition are equivalent.
[2]Generates exception on overflow or underflow.

Table 2.2 cont'd.
SWAR Addition Operations

Operation Types	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Modular Addition[1]				
Part/Part	-	-	2x32f	-
Immd/Part	-	-	-	-
Part/Part w/Acc (w/ or w/o Init)	-	-	-	-
Scalar/Part w/Acc (w/ or w/o Init)	-	-	-	-
Immd/Part w/Acc (w/ or w/o Init)	-	-	-	-
Element/Element	-	-	low 1x32f	-
Saturation Addition				
Part/Part	2x32f	-	-	4x16s[2]
Scalar/Part	-	-	-	-
Immd/Part	-	-	-	-
Modular Add. High				
Part/Part	-	-	-	-
Sat. RedAdd w/El.	-	-	-	-
Sat. Part. RedAdd w/Even	-	-	-	-
Sat. Part. RedAdd w/Part	-	-	-	-
Sat. RedAdd and Pack	2-2x32f→ 2x32f	-	-	-
Sat. RedAdd/Sub and Pack	-	2-2x32f→ 2x32f	-	-

[1]Modular signed and unsigned addition are equivalent.
[2]Stores result to implied destination register.

Because of its ubiquity and utility, and because it is the complement of modular addition, modular subtraction should be included in any general-purpose SWAR programming model.

As with modular subtraction, each family supports the forms of saturation subtraction which correspond to the supported forms of saturation addition. For completeness, and for the same reasons that saturation addition should be included, saturation subtraction should be included in any general-purpose SWAR programming model.

An NxB "subtraction high" (also known as "subtraction carry-out") zero-extends the complement of the carry bits that would result from a subtraction of the NxB operands and stores them into an NxB result. As with the addition high, only AltiVec includes this operation. Thus, it is not an operation that should be required in a general-purpose model.

"Saturation reduce-subtract and pack" (Sat. RedSub and Pack) performs separate saturated reduction subtractions on the elements of each of the sources, then packs the subresults into a partitioned result.

"Reduce-add of absolute differences" (RedAdd of Abs. Diffs) takes the parallel absolute differences of the operands, then performs a reduction addition on these subresults. This operation is supported by several of the extension families, and is used primarily for finding pixel differences in graphics applications.

Extended MMX includes an instruction which performs a "sum of absolute differences and saturation accumulate" (Sum of Abs. Diffs; Sat Acc.) operation which is similar to the above operation but accumulates with an operand in memory rather than performing a reduction. These instructions are probably too application-specific to be included in a general-purpose SWAR programming model, but may be useful for optimization purposes.

Table 2.3
SWAR Subtraction Operations

Operation Types	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX	Motorola AltiVec
Modular Subtraction[1]						
Part/Part	-	2x16	4x16	2x32f[2,3]	-	16x8, 8x16, 4x32
Part/Part w/Acc Diff (w/ or w/o Init)	-	-	-	-	2-8x8u→8x24s, 2-4x16s→4x48s	-
Scalar/Part w/Acc Diff (w/ or w/o Init)	-	-	-	-	2-8x8u→8x24s, 2-4x16s→4x48s	-
Immd/Part w/Acc Diff (w/ or w/o Init)	-	-	-	-	2-8x8u→8x24s, 2-4x16s→4x48s	-
Element/Element	-	-	-	-	-	-
Saturation Subtraction						
Part/Part	-	2x16s, 2x16u-2x16s →2x16u	2x16s, 2x16u-2x16s →2x16u	-	8x8u,4x16s	16x8s,16x8u, 8x16s,8x16u, 4x32s,4x32u,4x32f[4]
Scalar/Part	-	-	-	-	8x8u,4x16s	-
Immd/Part	-	-	-	-	8x8u,4x16s	-
Subtraction High						
Part/Part	-	-	-	-	-	4x32u
Sat. RedSub and Pack	-	-	-	-	-	-
RedAdd of Abs. Diffs	8x8u→1x64u	-	-	-	-	-
Sum Abs Diffs; Sat Acc.	-	-	-	-	-	-

[1]Modular signed and unsigned subtraction are equivalent.
[2]Calculated to infinite precision, then rounded according to current rounding mode in FCSR.
[3]Generates exception on overflow or underflow.
[4]Rounds to nearest.

Table 2.3 cont'd.
SWAR Subtraction Operations

Operation Types	Sun VIS	Intel MMX	Intel SSE	Intel SSE2
Modular Subtraction[1]				
Part/Part	2x16,4x16, 1x32,2x32	8x8, 4x16, 2x32	$4x32f$[2]	16x8, 8x16, 4x32, 1x64,2x64,2x64f
Part/Part w/Acc Diff (w/ or w/o Init)	-	-	-	-
Scalar/Part w/Acc Diff (w/ or w/o Init)	-	-	-	-
Immd/Part w/Acc Diff (w/ or w/o Init)	-	-	-	-
Element/Element	-	-	low 1x32f[2]	low 1x64f
Saturation Subtraction				
Part/Part	-	8x8s,8x8u, 4x16s,4x16u	-	16x8s,16x8u, 8x16s,8x16u
Scalar/Part	-	-	-	-
Immd/Part	-	-	-	-
Subtraction High				
Part/Part	-	-	-	-
Sat. RedSub and Pack	-	-	-	-
RedAdd of Abs. Diffs	8x8u$\rightarrow$1x64	-	8x8u$\rightarrow$1x16u[3]	16x8u$\rightarrow$2x16u[4]
Sum Abs Diffs; Sat Acc.	-	-	-	

[1]Modular signed and unsigned subtraction are equivalent.
[2]Generates exception on overflow or underflow.
[3]Upper 3x16 is zeroed. There is no possibility of overflow.
[4]Each 64-bit quadword is reduced to a 16 bit sum. The remaining fields are zeroed.

Table 2.3 cont'd.
SWAR Subtraction Operations

Operation Types	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Modular Subtraction[1]				
Part/Part	-	-	2x32f	-
Part/Part w/Acc Diff (w/ or w/o Init)	-	-	-	-
Scalar/Part w/Acc Diff (w/ or w/o Init)	-	-	-	-
Immd/Part w/Acc Diff (w/ or w/o Init)	-	-	-	-
Element/Element	-	-	low 1x32f	-
Saturation Subtraction				
Part/Part	2x32f	-	-	4x16s[2]
Scalar/Part	-	-	-	-
Immd/Part	-	-	-	-
Subtraction High				
Part/Part	-	-	-	-
Sat. RedSub and Pack	-	2-2x32f→ 2x32f	-	-
RedAdd of Abs. Diffs	-	8x8→1x16u[3,4]	-	-
Sum Abs Diffs; Sat Acc.	-	-	-	8x8u[5]

[1]Modular signed and unsigned subtraction are equivalent.
[2]Stores result to implied destination register.
[3]Upper 3x16 is zeroed. There is no possibility of overflow.
[4]I was not able to confirm the (un)signedness of this.
[5]One operand must be memory. Result is stored in implied register.

Maximum and Minimum Operations

Table 2.4 contains information on the various forms of maximum and minimum operations and operations pertaining to the sign or magnitude of the field data which are included in the studied extension sets.

Most of the families have some form of complementary maximum and minimum instructions. These are both ubiquitous and basic enough to be included in a general-purpose model. They are normally used to obtain the larger or smaller value of the corresponding elements from two partitioned operands. However, they can also be used in the emulation of unsupported saturation operations to limit result values to the required storable range.

Extended MMX includes a partitioned binary "magnitude" instruction which, for each pair of corresponding elements, stores the value with the larger absolute magnitude without changing its sign. However, EMMX is the only family which includes such an instruction; and it is unclear if any current CPU implements the EMMX extensions. Thus, this type of operation probably should not be included in a general-purpose model at this time.

MIPS-V includes "absolute value" and "negate" instructions for operating on single-precision floating-point data. While absolute value would be a useful instruction to include in a programming model, none of the families support it for integer data. Thus, it also should probably not be included in a general-purpose model at this time. In contrast, negation is easily emulated on almost all architectures, so it probably should be included.

Enhanced 3DNow!, 3DNow!Pro, SSE, and SSE2 each include a instructions to generate a zero-extended bitmasks from the sign bits of the fields of a partitioned register. These instructions are not particularly useful except for implementing conditional operations. Because of this, they should not be included as individual operations in a general-purpose programming model, but may be useful in the implementation of others.

Table 2.4
Maximum and Minimum Operations

Operation Types	DEC MVI	HP MAX	SGI MIPS-V	SGI MDMX	Motorola AltiVec	Sun VIS	Intel MMX
Maximum							
Part/Part	8x8s,8x8u, 4x16s,4x16u	-	-	8x8u, 4x16s	16x8s,16x8u, 8x16s,8x16u, 4x32s,4x32u,4x32f[1]	-	-
Scalar/Part	-	-	-	8x8u, 4x16s	-	-	-
Immd/Part	8x8s,8x8u, 4x16s,4x16u	-	-	8x8u, 4x16s	-	-	-
Element/Element	-	-	-	-	-	-	-
Minimum							
Part/Part	8x8s,8x8u, 4x16s,4x16u	-	-	8x8u, 4x16s	16x8s,16x8u, 8x16s,8x16u, 4x32s,4x32u,4x32f[1]	-	-
Scalar/Part	-	-	-	8x8u, 4x16s	-	-	-
Immd/Part	8x8s,8x8u, 4x16s,4x16u	-	-	8x8u, 4x16s	-	-	-
Element/Element	-	-	-	-	-	-	-
Magnitude Part/Part	-	-	-	-	-	-	-
Abs. Value Part/Part	-	-	2x32f	-	-	-	-
Negate Part/Part	-	-	2x32f	-	-	-	-
Generate Sign Mask	-	-	-	-	-	-	-

Operation Types	Intel SSE	Intel SSE2	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Maximum						
Part/Part	8x8u, 4x16s, 4x32f	16x8u, 8x16s, 2x64f	2x32f	8x8u, 4x16s	2x32f	-
Scalar/Part	-	-	-	-	-	-
Immd/Part	-	-	-	-	-	-
Element/Element	low 1x32f	low 1x64f	-	-	low 1x32f	-
Minimum						
Part/Part	8x8u, 4x16s, 4x32f	16x8u, 8x16s, 2x64f	2x32f	8x8u, 4x16s	2x32f	-
Scalar/Part	-	-	-	-	-	-
Immd/Part	-	-	-	-	-	-
Element/Element	low 1x32f	low 1x64f	-	-	low 1x32f	-
Magnitude Part/Part	-	-	-	-	-	4x16s
Abs. Value Part/Part	-	-	-	-	-	-
Negate Part/Part	-	-	-	-	-	-
Generate Sign Mask	8x8s→1x32, 4x32f→1x32	16x8s→1x32, 2x64f→1x32	-	8x8s→1x32	2x32f→1x32	-

[1] $+0.0 > -0.0$, and max(NaN, anything) = QNaN.

Multiplication Operations

Table 2.5 contains information on the various forms of multiplication instructions available in the studied extensions. These include modular and saturated multiplication, multiplications producing the upper word of their results, multiplication by sign bits, and averages.

MDMX, AltiVec, VIS, and MMX each include some form of modular integer multiplication. MDMX's multiplies each generate a result in the accumulator, which is large enough to maintain the full precision of the result. On the other architectures multiplies only operate on some of the source fields or store only part of each result in a space that is smaller than that necessary to hold the entire result.

Integer multiplications supported by AltiVec operate on the even or odd fields of their source registers and create a result with fields that have twice the precision of their source fields. SSE2 has a set of similar instructions which operate on the even fields of their operands, but these are limited to unsigned data. VIS includes several types, with results of various forms, each of which multiplies an 8-bit partitioned register by a 16-bit register. MMX and SSE2 include 16-bit versions which generate the lower 16-bits of their results.

Some of these instructions can be used to perform multiplications on data which is of smaller precision than that supported. They can also be used to perform partial multiplications of larger-precision data. Thus, the multiplication of unsupported data precisions can usually be emulated, but not always easily or inexpensively.

MIPS-V, SSE, and 3DNow!Pro each include partitioned 32-bit modular floating-point multiplies, while SSE2 includes a 64-bit version. SSE and 3DNow!Pro also include an instruction which multiplies the low elements of a register which is partitioned into 32-bit floats. Again, SSE2 includes a 64-bit version.

Because multiplications often occur in numeric processing, they should be included in a general-purpose programming model. Multiplies are fairly easy to emulate if some form is available, and can be emulated by a shift-add sequence otherwise. The VIS forms are rather esoteric, having been designed to be used primarily through a set of intrinsics. Thus, they would not be good models for operations included within a general-purpose programming model. However, they can be used to support such a model with some care.

The “multiply high” instruction stores the upper part of the result of a modular multiplication. It is used to complement multiplication instructions in which the stored value is the lower part of the full result. In each case, the stored part of the result resides in the same number of bits as the source data. Thus, there is no change of partitioning when using this type of instruction. These instructions are useful for emulating saturation multiplication, but are probably not useful enough on their own to make visible as part of a high-level programming model.

MDMX includes a few forms of saturated integer multiplication, while 3DNow! includes a saturating 32-bit floating-point multiply. Saturation multiplication is generally used for multimedia algorithms, but not for numeric computation. The extension families which include multiplies usually support either modular multiplication forms or saturating forms, but not both.

Integer saturation multiplication often can be emulated with other operations. However, floating-point saturation multiplication may be impossible to emulate on some targets, and modular floating-point multiplication may be impossible or expensive to emulate if the target only supports saturation multiplication. For these reasons, one may argue either way on the point of whether or not saturation multiplication should be included in a general-purpose model.

It is only on overflow that saturation operations differ from the corresponding modular operation, so one might argue that it should be acceptable to ignore the problem. However, the purpose of saturation math is to guarantee that the result does not overflow; thus, it should always work properly.

For the sake of completeness, both modular and saturation operation should be included, for both integer and floating-point data, but without any guarantee that the target can support both forms. This is similar to how floating-point multiplication is handled by the C programming language, in which there is no guarantee of the correctness of the result on overflow.

"Multiply by sign" is supported only by MDMX. This instruction multiplies an immediate value, a single-valued partitioned operand, or a partitioned value by the sign bits of the corresponding fields of a partitioned register. If a field in this register is 0, the corresponding result will also be 0. Because it is a special-purpose instruction which is only supported by one target, it should not be included as part of a portable programming model.

Some form of average instruction is supported by most of the extension families. This operation is commonly used in image and video processing but may be less useful in a general-purpose environment. Because of this, it is also arguable as to whether or not an averaging operation should be included in a general-purpose SWAR model, although it is relatively easy to emulate and widely supported.

Combined Arithmetic Operations

Several of the extension families contain instructions which are combinations of multiplications and other operations. These instructions are intended for use in implementing specific algorithms such as FFTs. Few are implemented by more than one family, and none should be used as the basis for operations in a general-purpose programming model. For this reason, these instructions are not discussed in detail; however, an entry in table 2.6 is provided which may be useful for optimization purposes.

Division and Advanced Arithmetic Operations

Table 2.7 lists arithmetic instructions useful for performing division and more complex arithmetic operations such as square roots and exponentials.

Table 2.5
Multiplication Operations

Operation Types	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX	Motorola AltiVec
Modular Multiplication						[1]
Part/Part	-	-	-	2x32f[2,3]	-	odd 16x8s→8x16s, odd 16x8u→8x16u, even 16x8s→8x16s, even 16x8u→8x16u, odd 8x16s→4x32s, odd 8x16u→4x32u, even 8x16s→4x32s, even 8x16u→4x32u
Immd/Part	-	-	-	-	-	-
Part/Part w/Acc (w/ or w/o Init)	-	-	-	-	2-8x8u→8x24s, 2-4x16s→4x48s	-
Scalar/Part w/Acc (w/ or w/o Init)	-	-	-	-	2-8x8u→8x24s, 2-4x16s→4x48s	-
Immd/Part w/Acc (w/ or w/o Init)	-	-	-	-	2-8x8u→8x24s, 2-4x16s→4x48s	-
Part/Part w/Acc Sub (w/ or w/o Init)	-	-	-	-	2-8x8u→8x24s, 2-4x16s→4x48s	-
Scalar/Part w/Acc Sub (w/ or w/o Init)	-	-	-	-	2-8x8u→8x24s, 2-4x16s→4x48s	-
Immd/Part w/Acc Sub (w/ or w/o Init)	-	-	-	-	2-8x8u→8x24s, 2-4x16s→4x48s	-
Part/Element	-	-	-	-	-	-
Element/Element	-	-	-	-	-	-
Modular Mul. High						
Pt/Pt Store in Enh.	-	-	-	-	-	-
Pt/Pt Store in Implied	-	-	-	-	-	-
Pt/Pt Acc. w/Implied	-	-	-	-	-	-
Sat. Multiplication						
Part/Part	-	-	-	-	8x8u,4x16s	-
Scalar/Part	-	-	-	-	8x8u,4x16s	-
Immd/Part	-	-	-	-	8x8u,4x16s	-
Mult. by Sign (-,0,+)						
Part/Part	-	-	-	-	4x16s	-
Scalar/Part	-	-	-	-	4x16s	-
Immd/Part	-	-	-	-	4x16s	-
Average	-	2x16u[5]	- 4x16u[5]	-	-	16x8s,16x8u,[4] 8x16s,8x16u, 4x32s,4x32u

[1] AltiVec byte numbering is the reverse of the field numbering used in this document.
[2] Generates exception on overflow or underflow.
[3] Calculated to infinite precision, then rounded according to current rounding mode in FCSR.
[4] Each of these performs (sum+1)/2.
[5] Round to odd : NewLSB <- sum(bit1) | sum(bit0). Sum before shift.

Table 2.5 cont'd.
Multiplication Operations

Operation Types	Sun VIS	Intel MMX	Intel SSE	Intel SSE2
Modular Multiplication			[1]	
Part/Part	(4x8u)x(4x16s) →4x24s→4x16s[2], (odd 8x8s)x(4x16s) →4x24s→4x16s[3], (even 8x8u)x(4x16s) →4x24s→4x32s[4] →4x16s[3], (odd 4x8s)x(2x16s) →2x24s→2x32s[5], (even 4x8u)x(2x16s) →2x24s→2x32s[4]	4x16	4x32f	8x16, even 2x32u→1x64u, even 4x32u→2x64u, 2x64f
Immd/Part	-	-	-	-
Part/Part w/Acc (w/ or w/o Init)	-	-	-	-
Scalar/Part w/Acc (w/ or w/o Init)	-	-	-	-
Immd/Part w/Acc (w/ or w/o Init)	-	-	-	-
Part/Part w/Acc Sub (w/ or w/o Init)	-	-	-	-
Scalar/Part w/Acc Sub (w/ or w/o Init)	-	-	-	-
Immd/Part w/Acc Sub (w/ or w/o Init)	-	-	-	-
Part/Element	(4x8u)x(upper 2x16s) →4x24s→4x16s[2], (4x8u)x(lower 2x16s) →4x24s→4x16s[2]	-	-	-
Element/Element	-	-	low 1x32f	low 1x64f
Modular Mul. High				
Pt/Pt Store in Enh.	-	4x16s	4x16u	8x16u,8x16s
Pt/Pt Store in Implied	-	-	-	-
Pt/Pt Acc. w/Implied	-	-	-	-
Sat. Multiplication				
Part/Part	-	-	-	-
Scalar/Part	-	-	-	-
Immd/Part	-	-	-	-
Mult. by Sign (-,0,+)				
Part/Part	-	-	-	-
Scalar/Part	-	-	-	-
Immd/Part	-	-	-	-
Average	-	-	8x8u[6], 4x16u[6]	16x8u, 8x16u

[1]Calculated to infinite precision, then rounded according to current rounding mode in FCSR.
[2]Most significant 16 bits of 24 are stored after rounding to nearest value.
[3]Rounds to nearest by adding 1/2 of lowest included position, then truncating lower bits.
[4]Sign-extended.
[5]Left-shifted logical by 8 bits.
[6]Performs (sum+1)/2.

Table 2.5 cont'd.
Multiplication Operations

Operation Types	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Modular Multiplication				
Part/Part	-	-	2x32f	-
Immd/Part	-	-	-	-
Part/Part w/Acc (w/ or w/o Init)	-	-	-	-
Scalar/Part w/Acc (w/ or w/o Init)	-	-	-	-
Immd/Part w/Acc (w/ or w/o Init)	-	-	-	-
Part/Part w/Acc Sub (w/ or w/o Init)	-	-	-	-
Scalar/Part w/Acc Sub (w/ or w/o Init)	-	-	-	-
Immd/Part w/Acc Sub (w/ or w/o Init)	-	-	-	-
Part/Element	-	-	-	-
Element/Element	-	-	low 1x32f	-
Modular Mul. High				
Pt/Pt Store in Enh.	4x16s[1]	4x16u	-	4x16s[2]
Pt/Pt Store in Implied	-	-	-	4x16s[2]
Pt/Pt Acc. w/Implied	-	-	-	4x16s[2]
Sat. Multiplication				
Part/Part	2x32f	-	-	-
Scalar/Part	-	-	-	-
Immd/Part	-	-	-	-
Mult. by Sign (-,0,+)				
Part/Part	-	-	-	-
Scalar/Part	-	-	-	-
Immd/Part	-	-	-	-
Average	8x8u[3]	8x8u[3], 4x16u[3]	-	8x8u or 8x8[4]

[1]Rounds to nearest, then truncates low 16 bits.
[2]Adds 0x4000 (bit 14) to product, then takes bits 30-15 as result.
[3]Performs (sum+1)/2.
[4]M2 versions prior to v1.3 perform 8x8; after v1.3 perform 8x8u. Both perform sum/2.

Table 2.6
Combined Arithmetic Operations

Operation Types	DEC MVI	HP MAX	SGI MIPS-V	SGI MDMX	Motorola AltiVec
Multiply, then Add Neighboring Fields	-	-	-	-	-
Multiply/Mod. Add	-	-	2x32f[1]	-	4x32f[2], 2-8x16→8x32+[3]8x16→8x16
Negated Multiply/Mod. Add	-	-	2x32f[1]	-	-
Multiply/Sat. Add	-	-	-	-	2-8x16s→8x32s→8x17s[4] +8x16s[5]→8x16s
Multiply(w/Rnd)/Sat. Add	-	-	-	-	2-8x16s→8x32s →8x18s[6] +8x16s[7]+(8x18s)"1" →8x16s
Multiply/Mod. Subtract	-	-	2-2x32f→2x32f –2x32f→2x32f[1]	-	-
Negated Multiply/Mod. Subtract	-	-	2-2x32f→2x32f –2x32f→2x32f[1]	-	4x32f[8]
Multiply, then Modular Add Neighbor w/Part	-	-	-	-	2-16x8u→16x16u +4x32u→4x32u, 2-8x16s→8x32s +4x32s→4x32s, 2-8x16u→8x32u +4x32u→4x32u, (16x8s)x(16x8u)→16x16s +4x32s→4x32s
Multiply, then Saturate Add Neighbor w/Part	-	-	-	-	2-8x16s→8x32s +4x32s→4x32s, 2-8x16u→8x32u +4x32u→4x32u

[1]Partitioned multiply of two operands, followed by partitioned addition with a third operand. Sum (or difference) calculated to infinite precision, then rounded according to FCSR mode.
[2]Partitioned multiply of two operands, followed by partitioned addition with a third operand, then rounded to nearest.
[3]8x16 modular add. The lower half of each 32-bit field is discarded.
[4]High 17 bits of field.
[5]Sign-extended to 17 bits.
[6]High 18 bits of field.
[7]Sign-extended to 17 bits, then shifted left logically to 18 bits.
[8]Partitioned multiply of two operands, followed by partitioned subtract of third operand, negated, then rounded to nearest.

Table 2.6 cont'd.
Combined Arithmetic Operations

Operation Types	Sun VIS	Intel MMX	Intel SSE	Intel SSE2	AMD 3DNow! (All families)	Cyrix EMMX
Multiply, then Add Neighboring Fields	-	2-(4x16s) →4x32s →2x32s	-	2-(8x16s) →8x32s →4x32s	-	-
Multiply/Mod. Add	-	-	-	-	-	-
Negated Multiply/Mod. Add	-	-	-	-	-	-
Multiply/Sat. Add	-	-	-	-	-	-
Multiply(w/Rnd)/Sat. Add	-	-	-	-	-	-
Multiply/Mod. Subtract	-	-	-	-	-	-
Negated Multiply/Mod. Subtract	-	-	-	-	-	-
Multiply, then Modular Add Neighbor w/Part	-	-	-	-	-	-
Multiply, then Saturate Add Neighbor w/Part	-	-	-	-	-	-

SSE and 3DNow!Pro include 32-bit floating-point divide and square root instructions which operate on two partitioned registers or on the low elements of two partitioned registers. SSE2 provides the same functionality for 64-bit elements.

AltiVec, SSE, and 3DNow!Pro each include instructions which approximate 32-bit partitioned floating-point reciprocals and reciprocal square roots. SSE, 3DNow!, and 3DNow!Pro also support low element forms of these instructions, although the 3DNow! versions are implemented as a series of three instructions rather than just one.

AltiVec also includes a set of instructions which perform partitioned 32-bit floating-point base-2 logarithmic ($\log_2 x$) and exponential (2^x) approximations.

Because each of these instructions is supported by a few targets at most, they should not be incorporated into a portable programming model. One may choose to make an exception for division because it is the inverse of multiplication. While it can be an expensive operation for targets which do not support it, division can usually be serialized without too much of a penalty compared to its typically long clock count.

2.1.3 Shift and Rotate Instructions

Table 2.8 lists forms of shift and rotate instructions which are available in the extension sets studied. These include logical and arithmetic shifts, shift-and-add and shift-and-subtract instructions, and simple rotations.

Logical shifts are a basic operation that should be included in any general-purpose programming model which allows bit manipulation. MDMX and AltiVec include integer shifts by partitioned and replicated scalar values. Using partitioned registers simplifies the use of general expressions as shift counts by allowing each element to be shifted by a different amount. Using a replicated scalar shift count requires that the same count be used for each, although it can be a dynamic value.

AltiVec also includes full-register shifts in which the count is stored as a single value in a vector register. The Alpha architecture's full-width integer shifts can also

Table 2.7
Division and Advanced Arithmetic Operations

Operation Types	DEC MVI	HP MAX	SGI MIPS-V	SGI MDMX	Motorola AltiVec	Sun VIS	Intel MMX
Divide							
Part/Part	-	-	-	-	-	-	-
Element/Element	-	-	-	-	-	-	-
Square Root							
Part/Part	-	-	-	-	-	-	-
Element/Element	-	-	-	-	-	-	-
Reciprocal Approx.							
Part	-	-	-	-	4x32f	-	-
Element	-	-	-	-	-	-	-
Recip. Sq. Rt. Approx.							
Part	-	-	-	-	4x32f	-	-
Element	-	-	-	-	-	-	-
Log$_2$(x) Approx.							
Part	-	-	-	-	4x32f	-	-
2^x Approx.							
Part	-	-	-	-	4x32f	-	-

Operation Types	Intel SSE	Intel SSE2	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Divide						
Part/Part	4x32f	2x64f	-	-	2x32f	-
Element/Element	low 1x32f	low 1x64f	-	-	low 1x32f	-
Square Root						
Part/Part	4x32f	2x64f	-	-	2x32f	-
Element/Element	low 1x32f	low 1x64f	-	-	low 1x32f	-
Reciprocal Approx.						
Part	4x32f	-	-	-	2x32f	-
Element	low 1x32f	-	low 1x32f[1]	-	low 1x32f	-
Recip. Sq. Rt. Approx.						
Part	4x32f	-	-	-	2x32f	-
Element	low 1x32f	-	low 1x32f[1]	-	low 1x32f	-
Log$_2$(x) Approx.						
Part	-	-	-	-	-	-
2^x Approx.						
Part	-	-	-	-	-	-

[1]Performed using three instructions: the first is accurate to 14 bits, the second is an intermediate step, and the third is accurate to 24 bits.

be used by MVI in this manner, as long as one considers the register to be partitioned into a single field. Full-register logical shifts can be used to emulate partitioned shifts, and are very important for the emulation of many other unsupported operations. MMX and SSE2 go one step further by including shifts by a single-valued register which operate on a partitioned operand. This eliminates the need to emulate these particular instructions with a series of full-register shifts.

MVI, MAX-2, MDMX, MMX, and SSE2 also include shifts by immediates. These are useful for implementing common operations such as multiplication by a constant. However, they have limited usefulness in an environment where the shift count will often be an expression rather than a static constant. These shifts are still quite useful as they can be used internally by a compiler to emulate unsupported operations.

Arithmetic right shifts are typically supported in the same forms as logical right shifts or in a subset of these forms. For example, in MDMX 8-bit data is considered to be unsigned pixels, so signed (i.e. arithmetic) shifts are not included for use with this field size. These instructions are basic to many numeric algorithms and should be included in a general-purpose model both for their utility and for the sake of completeness.

MVI also includes full-register "shift-and-add" and "shift-and-subtract" instructions which are intended for use in emulating multiplication and division for these RISC systems. These instructions are not as useful in a SWAR environment because the arithmetic parts of these operations are not partitioned. HP's MAX-1 includes partitioned "shift-and-saturation-add" instructions which are limited to 16-bit operands. These instructions are more general than simple shifts, and can be used wherever simple shifts can be. However, shift-and-add and shift-and-subtract are not operations that should be included in a general-purpose model because of their lack of portability.

Only AltiVec includes a "rotate" instruction, which is partitioned and indexed by a partitioned register. Even though only one target supports rotations, they are fairly easy to implement using shifts, so they could be included in a general-purpose model.

One should note that in a model which allows multi-word lengthed vectors, a rotation would actually consist of a series of shift instructions, with some masking, rather than being comprised of rotate instructions. Thus, rotate instructions are actually only useful in certain special cases.

2.1.4 Bitwise-Logical and Bit-Reduction Instructions

Bitwise-logical operations are extremely important for SWAR processing. These operations make enable masking for conditional constructs possible, as well as vector element accesses and the masking of non-data bits. By definition, all one-bit partitioned operations are bitwise operations. Also, many operations which are unsupported for some field size can be emulated by using bitwise operations.

We refer to these operations as being *polymorphic* because they perform exactly the same function regardless of the partitioning or signedness of their operands [106]. That is, they can assume the form of any partitioning of the data.

Polymorphics can form the basic building blocks for more advanced operations. Basic digital logic gates perform bitwise-logical operations. These, in turn, form the basis of more complex digital logic including the processors whose attributes are discussed in this chapter. Similarly, complex SWAR operations can be implemented as series of polymorphics. Because of their simple utility, these operations should be included in any general-purpose programming model.

Many of these operations are actually combinations of others, and thus not all of them need be supported. However, it is important that a working set from which necessary operations can be derived is supported. For example, MMX includes the instructions AND, ANDN, OR, and XOR, but not a simple one's complement operation. This basic operation, which is used to generate PE enable masks for `if-else` conditional execution, must be derived from the available polymorphic instructions. MMX's ANDN, which complements one of its arguments then ANDs it with the other,

Table 2.8
Shift and Rotate Operations

Operation Types	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX	Motorola AltiVec	Sun VIS
Shift Left Logical[1]							
Part by Part	-	-	-	-	8x8, 4x16	16x8, 8x16, 4x32	-
Part by Scalar	-	-	-	-	8x8, 4x16	1x128 by 16x8[2]	-
Part by Single	1x64	-	-	-	-	1x128[3]	-
Part by Immd	1x64	-	4x16	-	8x8, 4x16	-	-
Shift Right Logical[4]							
Part by Part	-	-	-	-	8x8u, 4x16u	16x8u, 8x16u, 4x32u	-
Part by Scalar	-	-	-	-	8x8u, 4x16u	-	-
Part by Single	1x64u	-	-	-	-	1x128[3]	-
Part by Immd	1x64u	-	4x16u	-	8x8u, 4x16u	-	-
Shift Right Arithmetic[4]							
Part by Part	-	-	-	-	4x16s	16x8, 8x16, 4x32	-
Part by Scalar	-	-	-	-	4x16s	-	-
Part by Single	1x64s	-	-	-	-	-	-
Part by Immd	1x64s	-	4x16s	-	4x16s	-	-
Shift Left and Add							
by 1 bit	-	-	-	-	-	-	-
by 2 bits	1x64u	-	-	-	-	-	-
by 3 bits	1x64u	-	-	-	-	-	-
Shift Left and Sat. Add[5]							
by 1,2, or 3 bits	-	2x16s	-	-	-	-	-
Shift Left and Subtract[6]							
by 2 bits	1x64u	-	-	-	-	-	-
by 3 bits	1x64u	-	-	-	-	-	-
Shift Right and Sat. Add							
by 1,2, or 3 bits	-	2x16s	-	-	-	-	-
Rotate[7]							
Part by Part	-	-	-	-	-	16x8, 8x16, 4x32	-

[1]Shift left logical and shift left arithmetic are equivalent.
[2]Shift count is scalar value mod 8.
[3]Shifted by number of bytes encoded in bits 6 through 3 (121-124 in AltiVec notation) of the single.
[4]Shift right logical is indicated as being unsigned. Shift right arithmetic is indicated as being signed.
[5]Shifts are signed saturated, then signed saturating addition is performed.
[6]Shifts the minuend then subtracts the unshifted subtrahend from it.
[7]Rotating left by x bits is equivalent to rotating right by B-x bits in an NxB register.

Table 2.8 cont'd.
Shift and Rotate Operations

Operation Types	Intel MMX	Intel SSE	Intel SSE2	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Shift Left Logical[1]							
Part by Part	-	-	-	-	-	-	-
Part by Scalar	-	-	-	-	-	-	-
Part by Single	4x16, 2x32, 1x64	-	8x16, 4x32, 2x64	-	-	-	-
Part by Immd	4x16, 2x32, 1x64	-	8x16, 4x32, 2x64, 1x128[2]	-	-	-	-
Shift Right Logical[3]							
Part by Part	-	-	-	-	-	-	-
Part by Scalar	-	-	-	-	-	-	-
Part by Single	4x16u, 2x32u, 1x64u	-	8x16u, 4x32u, 2x64u	-	-	-	-
Part by Immd	4x16u, 2x32u, 1x64u	-	8x16u, 4x32u, 2x64u, 1x128u[2]	-	-	-	-
Shift Right Arithmetic[3]							
Part by Part	-	-	-	-	-	-	-
Part by Scalar	-	-	-	-	-	-	-
Part by Single	4x16s, 2x32s	-	8x16s, 4x32s	-	-	-	-
Part by Immd	4x16s, 2x32s	-	8x16s, 4x32s	-	-	-	-
Shift Left and Add							
by 1 bit	-	-	-	-	-	-	-
by 2 bits	-	-	-	-	-	-	-
by 3 bits	-	-	-	-	-	-	-
Shift Left and Sat. Add[4]							
by 1,2, or 3 bits	-	-	-	-	-	-	-
Shift Left and Subtract[5]							
by 2 bits	-	-	-	-	-	-	-
by 3 bits	-	-	-	-	-	-	-
Shift Right and Add							
by 1,2, or 3 bits	-	-	-	-	-	-	-
Rotate[6]							
Part by Part	-	-	-	-	-	-	-

[1]Shift left logical and shift left arithmetic are equivalent.
[2]Shifted by number of bytes encoded in 8-bit unsigned immediate.
[3]Shift right logical is indicated as being unsigned. Shift right arithmetic is indicated as being signed.
[4]Shifts are signed saturated, then signed saturating addition is performed.
[5]Shifts the minuend then subtracts the unshifted subtrahend from it.
[6]Rotating left by x bits is equivalent to rotating right by B-x bits in an NxB register.

can be used to do just that by ANDing the complement of the enable mask with all '1's. This generates the enable mask for the `else` body from that for the `if` body.

All of the families include a working set of these instructions or reuse those of their base family or underlying architecture. For example, AMD's 3DNow! reuses the MMX polymorphic instructions, while MVI uses those of the underlying Alpha architecture.

A general-purpose model need only include a working set of polymorphics. Whatever set is chosen should be easy to emulate on any given target using the available instructions. Because of this, a small, limited set should be chosen. For example, one could choose to incorporate in the model only those operations supported by the C programming language: AND, OR, XOR, and one's complement.

Certain instructions perform what are essentially reduction operations on the individual bits of an operand. We will refer to these as *bit-reduction* operations. These include instructions which produce a count of the '1' bits or leading or trailing '0' bits in their operands. These can be used to gather information about the aggregate state of the data elements stored in a partitioned register. Note that only DEC's MVI has these instructions and these are actually part of the underlying Alpha architecture.

Table 2.9 lists the polymorphic and bitwise-reduction operations supported by each of the extension families studied.

2.1.5 Conditionals

Supported conditional instructions fall into three basic categories: those which generate result masks or condition codes, those which modify the flow of control, and those which manipulate data.

Result masks include bitmasks and fieldmasks. A *bitmask* contains one bit per field indicating if the condition is true or false for that field. These are usually stored in a general-purpose integer register. A *fieldmask* is a partitioned value in which all the bits of each field are set if the condition is true, or cleared if the condition is false, for

Table 2.9
Polymorphic Operations

Operation Types	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX	Motorola AltiVec	Sun VIS
AND							
Part/Part	1x64	1x32[1]	1x64[1]	-	8x8,4x16	1x128	1x32,1x64
Part/Imm	1x64	-	-	-	8x8,4x16	-	-
Part/Scalar	-	-	-	-	8x8,4x16	-	-
ANDN ($\overline{A}B$ or $A\overline{B}$)[2]							
Part/Part	1x64	1x32[1]	1x64[1]	-	-	1x128	1x32,1x64
Part/Imm	1x64	-	-	-	-	-	-
NAND ($\overline{AB}$)							
Part/Part	-	-	-	-	-	-	1x32,1x64
Part/Imm	-	-	-	-	-	-	-
OR							
Part/Part	1x64	1x32[1]	1x64[1]	-	8x8,4x16	1x128	1x32,1x64
Part/Imm	1x64	-	-	-	8x8,4x16	-	-
Part/Scalar	-	-	-	-	8x8,4x16	-	-
ORN ($A+\overline{B}$ or $\overline{A}+B$)[2]							
Part/Part	1x64	-	-	-	-	-	1x32,1x64
Part/Imm	1x64	-	-	-	-	-	-
NOR							
Part/Part	-	-	-	-	8x8,4x16	1x128	1x32,1x64
Part/Imm	-	-	-	-	8x8,4x16	-	-
Part/Scalar	-	-	-	-	8x8,4x16	-	-
XOR							
Part/Part	1x64	1x32[1]	1x64[1]	-	8x8,4x16	1x128	1x32,1x64
Part/Imm	1x64	-	-	-	8x8,4x16	-	-
Part/Scalar	-	-	-	-	8x8,4x16	-	-
XORN ($A \oplus \overline{B}$)							
Part/Part	1x64	-	-	-	-	-	-
Part/Imm	1x64	-	-	-	-	-	-
NXOR ($\overline{A \oplus B}$)							
Part/Part	-	-	-	-	-	-	1x32,1x64
Part/Imm	-	-	-	-	-	-	-
Population	1x64	-	-	-	-	-	-
Leading 0 bits	1x64	-	-	-	-	-	-
Trailing 0 bits	1x64	-	-	-	-	-	-

[1]Also nullifies the next instruction on condition.
[2]Not simultaneously.

Table 2.9 cont'd.
Polymorphic Operations

Operation Types	Intel MMX	Intel SSE	Intel SSE2	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
AND							
Part/Part	1x64	4x32f	1x128,2x64f	-	-	2x32f	-
Part/Imm	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-
ANDN ($\overline{A}B$ or $A\overline{B}$)[1]							
Part/Part	1x64	4x32f	1x128,2x64f	-	-	2x32f	-
Part/Imm	-	-	-	-	-	-	-
NAND ($\overline{AB}$)							
Part/Part	-	-	-	-	-	-	-
Part/Imm	-	-	-	-	-	-	-
OR							
Part/Part	1x64	4x32f	1x128,2x64f	-	-	2x32f	-
Part/Imm	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-
ORN ($A+\overline{B}$ or $\overline{A}+B$)[1]							
Part/Part	-	-	-	-	-	-	-
Part/Imm	-	-	-	-	-	-	-
NOR							
Part/Part	-	-	-	-	-	-	-
Part/Imm	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-
XOR							
Part/Part	1x64	4x32f	1x128,2x64f	-	-	2x32f	-
Part/Imm	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-
XORN ($A \oplus \overline{B}$)							
Part/Part	-	-	-	-	-	-	-
Part/Imm	-	-	-	-	-	-	-
NXOR ($\overline{A \oplus B}$)							
Part/Part	-	-	-	-	-	-	-
Part/Imm	-	-	-	-	-	-	-
Population	-	-	-	-	-	-	-
Leading 0 bits	-	-	-	-	-	-	-
Trailing 0 bits	-	-	-	-	-	-	-

[1]Not simultaneously.

the corresponding field(s) of the operand(s). Fieldmasks are normally stored in the same register set as their operands and are intended for use as SIMD enable masks.

Because of the nature of SWAR processing, enable masking must be used to limit the effects of conditionally executed code to those register fields for which the condition holds. Though some partitioned instructions can use bitmasks and condition codes directly; generally, they must be converted to fieldmasks for use in enable masking.

Condition codes represent the status or relationship of the operand(s) of a conditional operation. There may be one set per register field, in which case each set represents the condition of the corresponding field(s) of the operand(s), or one set per register, in which case they represent the aggregate condition of the fields in the register. Control codes are usually implemented as a bitmask which is stored in a "control register".

Control flow modification includes conditional instruction nullification and branches. *Instruction nullification* skips the instruction which follows the test or blocks any effects it might have. This instruction is usually a jump which is used to skip the following section of code. Similarly, branching instructions may jump if the condition is true or continue to the next instruction if not.

Because the effects of a nullifying or branching instruction cannot be separated on a per-field basis, the usefulness of these instructions is limited to aggregate tests, such as ANYs or ALLs, or to situations when the field tests can be serialized.

Data manipulation includes conditional moves, clears, and loads. Normally, these instructions are used to conditionally generate particular values or to select data from one of two execution paths. Again, the usefulness of these instructions is generally limited to aggregate or serialized tests.

Condition Testing Operations

Table 2.10 lists SWAR instructions which test conditions and generate masks or condition codes as a result. It also includes certain instructions which manipulate data based on the values of these objects.

Each of the families except HP's MAX has a set of instructions which conditionally set a fieldmask or a bitmask, or are extensions of families which do. The basic comparison tests include "equality", "inequality", "greater than", "less than", "greater than or equal", and "less than or equal". Generally, an architecture supports a subset of these tests which allows the others to be emulated. This holds true for the studied extension families. Thus, a general-purpose programming model should not exclude any of these basic tests.

SSE, SSE2, and 3DNow!Pro include tests for checking if IEEE-compliant floating-point data can be ordered (i.e. that it does not consist of NANs). NANs (not a numbers) are bit patterns that do not represent valid floating-point values. Comparisons which operate on floating-point numbers may allow for one or both operands to be NANs. In this case, the operands may not be comparable, and are said to be *unordered*. If both operands are valid numbers, they are said to be *ordered* or *orderable*.

These extension families also include floating-point "not less nor equal" and "not less than" tests which account for unorderedness, while MIPS-V includes these and a large set of variations on the basic tests for floating-point data. These tests are either combinations of the basic tests, or tests for situations which should not occur or should be hidden from the programmer. Thus, these tests should be internal or used as optimizations; they should not be a visible part of a high-level programming model.

AltiVec includes a "compare bounds" instruction which tests if the magnitude of one operand is less or equal to the magnitude of the other. This is equivalent to comparing the absolute values of two operands, and is essentially a combination of simpler tests. Each of the AltiVec tests also have a form in which the CR6 field of the processor's condition register is modified if the condition holds for all or none of the fields. This allows aggregate tests to be performed on partitioned register data.

SSE also includes instructions which compare two floating-point fields and set the processor's condition codes accordingly. These are most likely to be used in conjunction with the underlying IA32 instructions for control flow. Because they do not set a field or bitmask, they are less useful for SWAR enable masking.

Conditional Flow Control Operations

Table 2.11 lists instructions which can modify the flow of a program based on some condition. This may be done by branching or nullifying subsequent instructions which would normally cause change in flow.

MVI and MAX each contain conditional branch instructions which can be used as tests for control structures that must be able to handle parallel data. For example, a "while" loop executes as long as the conditional expression is non-zero. One way to convert this construct for use with SWAR data is to modify the test to be true as long as the expression is true for any field. This is equivalent to performing an ANY test on the partitioned conditional expression before entering the loop body, which is executed under an *enable mask* of the fields for which the condition holds. Conditional branch instructions make it easier to implement this type of parallel construct.

MAX includes a set of instructions which perform a logical or arithmetic operation then nullify the next instruction if an aggregate condition holds. These are typically used with a subsequent unconditional jump which is nullified, and therefore not taken, if the condition holds. This allows a section of code to be executed only if the aggregate condition holds.

Full-width (i.e. 1xN) branch or null-next instructions are not generally useful for parallel conditionals because they cannot take a different action for each field. It may be possible to construct a jump table to handle each combination of field

Table 2.10
Condition Testing Operations

Operation Types	DEC MVI	HP MAX	SGI MIPS-V	SGI MDMX	Motorola AltiVec
Forms of Result	Bitmask	-	FP CC Bits	FP CC Bits	Field Mask or All/None Bits
Equality[1]					
Part/Part	1x64	-	2x32f	8x8, 4x16	16x8, 8x16, 4x32,4x32f
Part/Imm	1x64	-	-	8x8,4x16	-
Part/Scalar	-	-	-	8x8,4x16	-
El/El	-	-	-	-	-
Inequality[1]					
Part/Part	-	-	2x32f	-	-
Part/Imm	-	-	-	-	-
Part/Scalar	-	-	-	-	-
El/El	-	-	-	-	-
Greater Than					
Part/Part	-	-	2x32f	-	16x8s,16x8u, 8x16s,8x16u, 4x32s,4x32u,4x32f
El/El	-	-	-	-	-
Less Than					
Part/Part	-	-	2x32f	8x8u, 4x16s	-
Part/Imm	-	-	-	8x8u,4x16s	-
Part/Scalar	-	-	-	8x8u,4x16s	-
El/El	-	-	-	-	-
Greater or Equal					
Part/Part	8x8u	-	2x32f	-	4x32f
Part/Imm	8x8u	-	-	-	-
Part/Scalar	-	-	-	-	-
Less or Equal					
Part/Part	-	-	2x32f	8x8u, 4x16s	-
Part/Imm	-	-	-	8x8u,4x16s	-
Part/Scalar	-	-	-	8x8u,4x16s	-
El/El	-	-	-	-	-
Not Less nor Equal					
Part/Part	-	-	2x32f	-	-
Element/Element	-	-	-	-	-
Not Less Than					
Part/Part	-	-	2x32f	-	-
Element/Element	-	-	-	-	-

[1]Compare for (in)equality signed and unsigned are equivalent.

Table 2.10 cont'd.
Condition Testing Operations

Operation Types	Sun VIS	Intel MMX	Intel SSE	Intel SSE2	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Forms of Result	Bitmask[1]	Field Mask	Field Mask	Field Mask	Field Mask	-	Field Mask	-
Equality[2]								
Part/Part	4x16, 2x32	8x8, 4x16, 2x32	4x32f	16x8, 8x16, 4x32, 2x64f	2x32f	-	2x32f	-
Part/Imm	-	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-	-
El/El	-	-	1x32f	1x64f	-	-	1x32f	-
Inequality[2]								
Part/Part	4x16, 2x32	-	4x32f	2x64f	-	-	2x32f	-
Part/Imm	-	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-	-
El/El	-	-	1x32f	1x64f	-	-	1x32f	-
Greater Than	[3]							
Part/Part	4x16, 2x32	8x8s, 4x16s, 2x32s	-	16x8, 8x16, 4x32	2x32f	-	-	-
El/El	-	-	-	-	-	-	-	-
Less Than								
Part/Part	-	-	4x32f	2x64f	-	-	2x32f	-
Part/Imm	-	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-	-
El/El	-	-	1x32f	1x64f	-	-	1x32f	-
Greater or Equal								
Part/Part	-	-	-	-	2x32f	-	-	-
Part/Imm	-	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-	-
Less or Equal	[3]							
Part/Part	4x16, 2x32	-	4x32f	2x64f	-	-	2x32f	-
Part/Imm	-	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-	-
El/El	-	-	1x32f	1x64f	-	-	1x32f	-
Not Less nor Equal								
Part/Part	-	-	4x32f	2x64f	-	-	2x32f	-
Element/Element	-	-	1x32f	1x64f	-	-	1x32f	-
Not Less Than								
Part/Part	-	-	4x32f	2x64f	-	-	2x32f	-
Element/Element	-	-	1x32f	1x64f	-	-	1x32f	-

[1]Bitmask stored in an integer register.
[2]Compare for (in)equality signed and unsigned are equivalent.
[3]I was never able to confirm (un)signedness of these, but assume signed as per fixed point format.

Table 2.10 cont'd.
Condition Testing Operations

Operation Types	DEC MVI	HP MAX	SGI MIPS-V	SGI MDMX	Motorola AltiVec	Sun VIS
Forms of Result	Bitmask	-	FP CC Bits	FP CC Bits	Field Mask or All/None Bits	Bitmask[1]
Not (Greater or Equal) Pt/Pt	-	-	2x32f	-	-	-
Greater or Less Than Pt/Pt	-	-	2x32f	-	-	-
Not (Greater or Less) Pt/Pt	-	-	2x32f	-	-	-
Not Greater Than Pt/Pt	-	-	2x32f	-	-	-
Greater, Less, or Equal Pt/Pt	-	-	2x32f	-	-	-
Not (Gr., Less, or Eq.) Pt/Pt	-	-	2x32f	-	-	-
Ordered						
Part/Part	-	-	2x32f	-	-	-
Element/Element	-	-	-	-	-	-
Unordered						
Part/Part	-	-	2x32f	-	-	-
Element/Element	-	-	-	-	-	-
Unordered or Equal Pt/Pt	-	-	2x32f	-	-	-
Signaling Equal Pt/Pt	-	-	2x32f	-	-	-
Signaling Not Equal Pt/Pt	-	-	2x32f	-	-	-
Ordered or Greater Than Pt/Pt	-	-	2x32f	-	-	-
Unordered or Greater Pt/Pt	-	-	2x32f	-	-	-
Ord. or Greater or Eq. Pt/Pt	-	-	2x32f	-	-	-
Unord. or Grtr. or Eq. Pt/Pt	-	-	2x32f	-	-	-
Ordered or Less Than Pt/Pt	-	-	2x32f	-	-	-
Unordered or Less Than Pt/Pt	-	-	2x32f	-	-	-
Ordered or Less or Eq. Pt/Pt	-	-	2x32f	-	-	-
Unord. or Less or Eq. Pt/Pt	-	-	2x32f	-	-	-
Ord. or Greater or Less Pt/Pt	-	-	2x32f	-	-	-
Compare Bounds[2]Pt/Pt	-	-	-	-	4x32f	-
Set Cond. Codes						
Ordered El/El	-	-	-	-	-	-
Unord. El/El	-	-	-	-	-	-

[1]Bitmask stored in an integer register. This can be used for masked stores.
[2]Clears bit 0 of result field if vA <= vB, and clears bit 1 if vA >=-(vB). In either case, the remaining bits are cleared.

Table 2.10 cont'd.
Condition Testing Operations

Operation Types	Intel MMX	Intel SSE	Intel SSE2	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Forms of Result	Field Mask	Field Mask	Field Mask	Field Mask	-	Field Mask	-
Not (Greater or Equal) Pt/Pt	-	-	-	-	-	-	-
Greater or Less Than Pt/Pt	-	-	-	-	-	-	-
Not (Greater or Less) Pt/Pt	-	-	-	-	-	-	-
Not Greater Than Pt/Pt	-	-	-	-	-	-	-
Greater, Less, or Equal Pt/Pt	-	-	-	-	-	-	-
Not (Gr., Less, or Eq.) Pt/Pt	-	-	-	-	-	-	-
Ordered							
Part/Part	-	4x32f	2x64f	-	-	2x32f	-
Element/Element	-	1x32f	1x64f	-	-	1x32f	-
Unordered							
Part/Part	-	4x32f	2x64f	-	-	2x32f	-
Element/Element	-	1x32f	1x64f	-	-	1x32f	-
Unordered or Equal Pt/Pt	-	-	-	-	-	-	-
Signaling Equal Pt/Pt	-	-	-	-	-	-	-
Signaling Not Equal Pt/Pt	-	-	-	-	-	-	-
Ordered or Greater Than Pt/Pt	-	-	-	-	-	-	-
Unordered or Greater Pt/Pt	-	-	-	-	-	-	-
Ord. or Greater or Eq. Pt/Pt	-	-	-	-	-	-	-
Unord. or Grtr. or Eq. Pt/Pt	-	-	-	-	-	-	-
Ordered or Less Than Pt/Pt	-	-	-	-	-	-	-
Unordered or Less Than Pt/Pt	-	-	-	-	-	-	-
Ordered or Less or Eq. Pt/Pt	-	-	-	-	-	-	-
Unord. or Less or Eq. Pt/Pt	-	-	-	-	-	-	-
Ord. or Greater or Less Pt/Pt	-	-	-	-	-	-	-
Compare Bounds[1] Pt/Pt	-	-	-	-	-	-	-
Set Cond. Codes							
Ordered El/El	-	1x32f	1x64f	-	-	1x32f	-
Unord. El/El	-	1x32f	1x64f	-	-	1x32f	-

[1] Clears bit 0 of result field if vA <= vB, and clears bit 1 if vA >=-(vB). In either case, the remaining bits are cleared.

Table 2.11
Conditional Flow Control Operations

Operation Types	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX	Motorola AltiVec	Sun VIS
Branch On...							
None True	1x64	-	-	-	-	-	-
Any True	1x64	-	-	-	-	-	-
All Equal (Part/Part)	-	1x32	1x64	-	-	-	-
All Equal (Part/Immed)	-	1x32	1x64	-	-	-	-
All Inequal (Part/Part)	-	1x32	1x64	-	-	-	-
All Inequal (Part/Immed)	-	1x32	1x64	-	-	-	-
Operate and Null Next On...							
AND/Any True?	-	1x32	1x64	-	-	-	-
AND/None True?	-	1x32	1x64	-	-	-	-
ANDN/Any True?	-	1x32	1x64	-	-	-	-
ANDN/None True?	-	1x32	1x64	-	-	-	-
OR/Any True?	-	1x32	1x64	-	-	-	-
OR/None True?	-	1x32	1x64	-	-	-	-
XOR/Any True?	-	1x32	1x64	-	-	-	-
XOR/None True?	-	1x32	1x64	-	-	-	-
XOR/Any False?	-	2x16 4x8	2x32 4x16 8x8	-	-	-	-
XOR/None False?	-	2x16 4x8	2x32 4x16 8x8	-	-	-	-
Add Complement/Any False? $(A+\overline{B})$	-	2x16 4x8	2x32 4x16 8x8	-	-	-	-
Add Complement/None False? $(A+\overline{B})$	-	2x16 4x8	2x32 4x16 8x8	-	-	-	-

result, but this would be an O(2^N)-sized table for an NxB partitioning. For this reason, these instructions are not included in table 2.11. Full-width branches or null-next instructions based on conditions that are equivalent to a reduction of the field conditions (such as an unpartitioned equality test which is equivalent to a partitioned ALL-equal test) are useful, and are included in the table.

Conditional Data Manipulation Operations

Table 2.12 lists instructions which manipulate data based the results of some conditional test. These include instructions which move data or clear or load registers when some condition is met. They also include instructions which select a set of values from a set of operands depending on some condition.

Table 2.11 cont'd.
Conditional Flow Control Operations

Operation Types	Intel MMX	Intel SSE	Intel SSE2	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Branch On...							
None True	-	-	-	-	-	-	-
Any True	-	-	-	-	-	-	-
All Equal (Part/Part)	-	-	-	-	-	-	-
All Equal (Part/Immed)	-	-	-	-	-	-	-
Any Inequal (Part/Part)	-	-	-	-	-	-	-
Any Inequal (Part/Immed)	-	-	-	-	-	-	-
Operate and Null Next On...							
AND/Any True?	-	-	-	-	-	-	-
AND/None True?	-	-	-	-	-	-	-
ANDN/Any True?	-	-	-	-	-	-	-
ANDN/None True?	-	-	-	-	-	-	-
OR/Any True?	-	-	-	-	-	-	-
OR/None True?	-	-	-	-	-	-	-
XOR/Any True?	-	-	-	-	-	-	-
XOR/None True?	-	-	-	-	-	-	-
XOR/Any False?	-	-	-	-	-	-	-
XOR/None False?	-	-	-	-	-	-	-
Add Complement/Any False? $(A+\overline{B})$	-	-	-	-	-	-	-
Add Complement/None False? $(A+\overline{B})$	-	-	-	-	-	-	-

MVI includes instructions which will move a register or load an immediate value based on the equivalent of an ANY or NONE test. It also includes instructions that conditionally zero (clear) the fields of an 8x8 partitioned register based on the value of a bitmask, which is usually generated by one of the MVI testing instructions.

HP's MAX includes instructions which clear a register to generate a "false" value, then perform a comparison for equality or inequality, and conditionally nullify the following instruction based on the result. The possibly nullified instruction is usually used to load an immediate value which represents "true" into the cleared register. These instructions can be used to implement or optimize aggregate tests for SIMD-style loops and conditionals.

Extended MMX includes instructions which load the fields of a register based on the value of the corresponding fields of a partitioned register. These can be used to implement or optimize certain conditional or trinary operations.

The MIPS-V extension family includes instructions which move the fields of a register based on the value of the corresponding control code bit. These also can be used to implement or optimize certain conditional or trinary operations.

Full-width (i.e. 1xN) conditional move instructions are not generally useful for parallel conditionals because they cannot take a different action for each field. For this reason, these instructions are not included in table 2.12. Full-width conditional moves based on conditions that are equivalent to a reduction of the field conditions are included in the table.

MDMX and AltiVec include partitioned "pick" or "select" instructions which select between one of two operands for each field based on the truth of the corresponding bit in a bitmask. In MDMX this bitmask is in an integer register and in AltiVec this bitmask is in a third vector register. These instructions are useful for implementing trinary operators or for selecting between the results of two conditional instruction streams. The choice of a 128x1 select for AltiVec is very good as it allows it to be used polymorphically.

Table 2.12
Conditional Data Manipulation Operations

Operation Types	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX	Motorola AltiVec	Sun VIS
Move Reg/Imm On...							
None True	1x64	-	-	-	-	-	-
Any True	1x64	-	-	-	-	-	-
Zero Masked Bytes	8x8bm	-	-	-	-	-	-
Zero UnMasked Bytes	8x8bm	-	-	-	-	-	-
Clear Reg & Null Next/All							
Part/Part	-	1x32	1x64	-	-	-	-
Part/Imm	-	1x32	1x64	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-
Clear Reg & Null Next/Not All							
Part/Part	-	1x32	1x64	-	-	-	-
Part/Imm	-	1x32	1x64	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-
Load Reg. On...							
Zero	-	-	-	-	-	-	-
Non-Zero	-	-	-	-	-	-	-
Negative	-	-	-	-	-	-	-
Non-Negative	-	-	-	-	-	-	-
Move Reg. On...							
CC bit TRUE	-	-	-	2x32f	-	-	-
CC bit FALSE	-	-	-	2x32f	-	-	-
Pick True					[1]	[2]	
Part/Part	-	-	-	-	8x8,4x16	128x1	-
Part/Imm	-	-	-	-	8x8,4x16	-	-
Part/Scalar	-	-	-	-	8x8,4x16	-	-
Pick False					[1]	[2]	
Part/Part	-	-	-	-	8x8,4x16	128x1	-
Part/Imm	-	-	-	-	8x8,4x16	-	-
Part/Scalar	-	-	-	-	8x8,4x16	-	-

[1] Chooses destination field from source v_s or v_t based on value of condition code bit corresponding to that field.
[2] Chooses destination bit from source vector A or B based on value of corresponding bit in source vector C. This is more general, but possibly harder to generate than MDMX condition code bits.

Table 2.12 cont'd.
Conditional Data Manipulation Operations

Operation Types	Intel MMX	Intel SSE	Intel SSE2	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Move Reg/Imm On...							
None True	-	-	-	-	-	-	-
Any True	-	-	-	-	-	-	-
Zero Masked Bytes	-	-	-	-	-	-	-
Zero UnMasked Bytes	-	-	-	-	-	-	-
Clear Reg & Null Next/All							
Part/Part	-	-	-	-	-	-	-
Part/Imm	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-
Clear Reg & Null Next/Not All							
Part/Part	-	-	-	-	-	-	-
Part/Imm	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-
Load Reg. On...							
Zero	-	-	-	-	-	-	8x8
Non-Zero	-	-	-	-	-	-	8x8
Negative	-	-	-	-	-	-	8x8s
Non-Negative	-	-	-	-	-	-	8x8s
Move Reg. On...							
CC bit TRUE	-	-	-	-	-	-	-
CC bit FALSE	-	-	-	-	-	-	-
Pick True							
Part/Part	-	-	-	-	-	-	-
Part/Imm	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-
Pick False							
Part/Part	-	-	-	-	-	-	-
Part/Imm	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-

2.1.6 Data Movement, Replication, and Type Conversion Operations

Table 2.13 lists the instructions available in each of the extension families for supporting data movement, replication, and type conversion operations.

MMX includes instructions to move data between its enhanced (i.e. partitionable) registers and also between these and the underlying IA32 architecture's general-purpose integer registers. SSE includes instructions to move unaltered data between the SSE registers, but not between the MMX and SSE registers. SSE2 includes instructions to correct this problem, and also includes instructions to move data between the SSE registers and the integer register set, and to allow data to be moved in various ways between the SSE registers.

Such instructions are not necessary in DEC's MVI or HP's MAX extensions because these extensions use the general-purpose registers of the underlying architecture. For example, MAX-2 has an instruction for moving a full-width (64-bit) object between the general registers that is actually part of the PA-RISC 2.0 instruction set architecture.

Neither MDMX nor AltiVec include instructions which are used solely for moving data between their enhanced registers or between these and their general registers. Similarly, MVI does not include instructions used solely for moving data within its general register set. Moving data between registers within the same register set can usually be emulated. For example, in AltiVec, a register can be bitwise-ORed with itself and the result stored in the target register. However, writing data between different register sets usually cannot be emulated. In these cases, data must be moved via the architecture's memory subsystem.

This is the case for AltiVec. Unfortunately, memory addresses for AltiVec are held in the PowerPC's general-purpose integer registers. This causes some addressing forms to be very expensive to execute. For example, when an array or vector element is indexed using vector indexing, the index must be moved from a vector register to memory, then from memory to an integer register where it can be used in an indexed load. This triples the number of memory accesses required for each vector indexed element access.

MIPS-V includes an instruction for moving data between registers in "packed single" (2x32f) format, and another to create this format by packing two single-precision values, taken from two floating-point registers, into a single floating-point register.

VIS includes instructions for moving either 32 or 64 bits of data between its enhanced registers. It also has a set of complementary instructions which allow the moved data to be stored in complemented form. This effectively performs a one's complement operation on the data.

While MDMX does not include instructions for moving partitioned data between its enhanced registers, it does include multiple instructions for moving data between the floating-point registers and the accumulator. These are not useful as part of a portable model, as none of the other extension families has a separate accumulator. However, they would be necessary for using the accumulator to operate on floating-point data if included in such a model.

AltiVec has a set of "splat" instructions which replicate either a field of the source register or an immediate value into all of the fields of the target register. This is the only instruction in any of the families which performs an actual replication, although MDMX and SSE each include instructions which effectively replicate one operand. A general-purpose model should include the field replication to convert scalar data to partitioned data for mixed-mode operations.

MDMX also includes instructions for scaling data within the accumulator. These instructions shift each field of the accumulator right by the number of bits specified by a secondary source, round these values to an integer value by truncation or rounding upward or downward with half values, then saturate these values to fit in the fields of the destination register. The secondary source may be a partitioned register, a scalar, or an immediate value. These instructions may be useful for implementing various type conversions.

AltiVec, 3DNow!, E3DNow!, and SSE include instructions to convert data in their enhanced registers between integer and floating-point type. The SSE conversions actually move data between the SSE registers and the MMX or IA32 register sets, simultaneously making the conversion. AltiVec also includes instructions which round floating-point data to integer-valued floating-point data. SSE2 includes instructions for converting between floating-point formats within the SSE registers. These instructions may be used for visible type casting by a programmer or for internal operations by a compiler.

2.1.7 Data Extraction, Insertion, and Permutation Operations

Table 2.14 lists the instructions available within each of the extension families for supporting field extraction, insertion, and permutation operations.

In general, insertions take a bit or byte field from a source and place it in a contiguous section of the destination. Extractions typically take data from a section of the source, align it with the least significant bit of the destination, and zero- or sign- extend it to fill that destination. These instructions could be used in a vector processing model to implement vector element accesses.

Enhanced 3DNow!, SSE, and SSE2 each include instructions to allow a field to be extracted from an enhanced register to an integer register. The complementary instructions which allow a field to be inserted from an integer register or from memory without altering the remaining fields are likewise included. SSE also includes an instruction which takes the low field of a 4x32f operand from an SSE register or memory and inserts it into the low field of a second SSE register. 3DNow!Pro has an instruction that performs the same operation on a 2x32f operand, while SSE2 has one for 64-bit operands.

MVI, AltiVec, and VIS include "byte shift right and extract" instructions which shift the source data right by n bytes, then clear the upper fields to leave the data in

Table 2.13
Data Movement, Replication, and Type Conversion Operations

Operation Types	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX	Motorola AltiVec	Sun VIS
Move Reg.→Enh. Reg.	N/A	N/A	N/A	-	-	-	-
Move Enh. Reg.→Reg.	N/A	N/A	N/A	-	-	-	-
Move Enh. Reg. →Enh. Reg.	-	1x32[1]	1x64[1]	2x32f	-	-	1x32, 1x64
Move Comp. Enh. Reg. →Enh. Reg.	-	-	-	-	-	-	1x32, 1x64
Pack Singles to Part	-	-	-	2-32f→2x32f	-	-	-
Modular Move Acc→Reg							
Low Third of Acc.	-	-	-	-	3x64→8x8u, 3x64→4x16s	-	-
Middle Third of Acc.	-	-	-	-	3x64→8x8u, 3x64→4x16s	-	-
High Third of Acc.	-	-	-	-	3x64→8x8u, 3x64→4x16s	-	-
Move Regs. to Low Acc.	-	-	-	-	2-8x8u→3x64[2], 2-4x16s→3x64	-	-
Move Reg. to High Acc.	-	-	-	-	8x8u→3x64, 4x16s→3x64	-	-
Replicate Field (Element/Part)[3]	-	-	-	-	-	16x8, 8x16, 4x32	-
Replicate Sign-Extended Immediate to Part[4]	-	-	-	-	-	16x8, 8x16, 4x32	-
Shift Rt, Rnd, & Sat Acc							
toward 0	-	-	-	-	8x8u, 4x16s, 4x16u	-	-
to nearest away from 0	-	-	-	-	8x8u, 4x16s, 4x16u	-	-
to nearest toward even	-	-	-	-	8x8u, 4x16s, 4x16u	-	-
Convert int. to flt.	-	-	-	-	-	4x32u→4x32f[5] 4x32s→4x32f[5]	-
Convert flt. to int.	-	-	-	-	-	4x32f→4x32u[6] 4x32f→4x32s[6]	-
Round flt. value to int.							
to nearest	-	-	-	-	-	4x32f	-
toward zero	-	-	-	-	-	4x32f	-
toward +infinity	-	-	-	-	-	4x32f	-
toward -infinity	-	-	-	-	-	4x32f	-

[1]Also branches if condition is met.
[2]Moves source register V_t to low third, source register V_s to middle third, and a set of fields consisting of the sign bits of the fields of V_s to the upper third.
[3]Field selected is indicated by unsigned immediate.
[4]Sign-extends 5-bit immediate to size of fields, then replicates.
[5]Converts to nearest, then divides by 2^{uimm5}, where uimm5 is a 5-bit unsigned immediate.
[6]Shifts left by a 5-bit unsigned immediate, converts and rounds toward zero, then saturates.

Table 2.13 cont'd.
Data Movement, Replication, and Type Conversion Operations

Operation Types	Intel MMX	Intel SSE	Intel SSE2
Move Reg→Enh. Reg.	1x32u→1x64u[1]	-	1x32u→1x128u[1]
Move Enh. Reg→Reg.	1x64→1x32[2]	-	1x128→1x32[2]
Move Enh. Reg→Enh. Reg.	1x64	4x32f (un)aligned	low 2x64→low 2x64, low 2x64→1x64,1x64→low 2x64[3], 1x128 (un)aligned, 2x64f (un)aligned
Move Comp. Enh. Reg. →Enh. Reg.	-	-	-
Pack Singles to Part	-	-	-
Modular Move Acc→Reg			
Low Third of Acc.	-	-	-
Middle Third of Acc.	-	-	-
High Third of Acc.	-	-	-
Move Regs. to Low Acc.	-	-	-
Move Reg. to High Acc.	-	-	-
Replicate Field	-	-	-
Replicate Sign-Extended Immediate to Part[4]	-	-	-
Shift Rt, Rnd, & Sat Acc			
toward 0	-	-	-
to nearest away from 0	-	-	-
to nearest toward even	-	-	-
Convert int. to flt.	-	2x32s→low 2x32f[5], 1x32s→low 1x32f[7]	2x32s→low 2x64f[6], 1x32s→low 1x64f[7], 4x32s→4x32f, low 2x32s→2x64f
Convert flt. to int.	-	low 2x32f→2x32[8,9], low 1x32f→1x32[11,9]	2x64f→2x32[8,9], 2x64f→low 2x32[10], low 1x64f→1x32[8,9], 4x32f→4x32s
Convert flt. to flt.	-	-	2x64f→low 2x32f[10], low 2x32f→2x64f[12], low 1x64f→low 1x32f[12], low 1x32f→low 1x64f[12]
Round flt. value to int.			
to nearest	-	-	-
toward zero	-	-	-
toward +infinity	-	-	-
toward -infinity	-	-	-

[1] Zero-extended.
[2] Truncated.
[3] Upper quadword cleared.
[4] Sign-extends 5-bit immediate to size of fields, then replicates.
[5] Source is MMX register or memory. Destination is SSE register. High fields are left unchanged.
[6] Source is MMX register or memory. Destination is SSE register.
[7] Source is integer register or memory. Destination is SSE register. High fields are left unchanged.
[8] Source is SSE register. Destination is MMX register or memory.
[9] Cvt* uses rounding mode specified in MXCSR. Cvtt* truncates the fractional part.
[10] Source is SSE register or memory. Destination is SSE register with upper half cleared.
[11] Source is SSE register. Destination is integer register or memory.
[12] Source is SSE register or memory. Destination is SSE register.

Table 2.13 cont'd.
Data Movement, Replication, and Type Conversion Operations

Operation Types	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Move Reg→Enh. Reg.	-	-	-	-
Move Enh. Reg→Reg.	-	-	-	-
Move Enh. Reg→Enh. Reg.	-	-	2x32 (un)aligned	-
Move Comp. Enh. Reg. →Enh. Reg.	-	-	-	-
Pack Singles to Part	-	-	-	-
Modular Move Acc→Reg				
Low Third of Acc.	-	-	-	-
Middle Third of Acc.	-	-	-	-
High Third of Acc.	-	-	-	-
Move Regs. to Low Acc.	-	-	-	-
Move Reg. to High Acc.	-	-	-	-
Replicate Field	-	-	-	-
Replicate Sign-Extended Immediate to Part[1]	-	-	-	-
Shift Rt, Rnd, & Sat Acc				
toward 0	-	-	-	-
to nearest away from 0	-	-	-	-
to nearest toward even	-	-	-	-
Convert int. to flt.	2x32s→2x32f	even 4x16s →2x32f	2x32s→2x32f, 1x32s→low 1x32f	-
Convert flt. to int.	2x32f→2x32s	2x32f →2x32s[3]	2x32f→2x32[2], low 1x32f→1x32[2]	-
Convert flt. to flt.	-	-	-	-
Round flt. value to int.				
to nearest	-	-	-	-
toward zero	-	-	-	-
toward +infinity	-	-	-	-
toward -infinity	-	-	-	-

[1]Sign-extends 5-bit immediate to size of fields, then replicates.
[2]Cvt* uses rounding mode specified in MXCSR. Cvtt* truncates the fractional part.
[3]Sign saturated to 16 bits, then sign-extended to 32.

zero-extended form. MVI, MIPS-V, and MDMX include "byte shift left and extract" instructions which shift the data left before clearing the upper fields.

MVI's byte extraction instructions operate on a single source object and are well-suited for field extraction. Those of the other extension families operate on a pair of source objects and are best suited to handling unaligned memory accesses, although they also can be used for field extraction.

MVI also includes "byte shift and insert" instructions which shift the source data left or right by n bytes, then clear all but the byte, word, doubleword, or quadword starting at the n^{th} byte and going upwards through the register. The byte count n may be stored in a register or may be an immediate value. These instructions allow the programmer to select a set of contiguous bytes from the right end of the source and place them in any set of bytes in the destination with the other bytes cleared.

MAX includes "bit shift left and extract" instructions which take up to B bits from the right-hand field of an N×B partitioned register, starting at any bit position and extending to the left, then copy them into the target register, aligned with its right end. The number of bits copied may be taken from an immediate or stored in a shift amount register (SAR). If the count is from the SAR, the copied segment is sign- or zero- extended to fill the target field. The left-hand field (if N>1) is undefined. If the count is an immediate, it is an undefined operation if the copied segment extends beyond the end of the source field.

MAX also includes "merge, bit shift right and extract" instructions. In these, the rightmost fields of two N×B sources are concatenated and the resulting value is shifted right. The lower B bits of the $2B$-bit concatenation are then extracted. Again, the shift count can be an immediate value or from the SAR, and the left-hand field of the destination register is undefined.

Complementing these extraction instructions, MAX also contains "deposit" instructions which perform "bit shift left and insert" operations. In an N×B partitioned operation, these take up to B bits from the right end of the source and copy them into the target register. Writing begins at any of the rightmost B bit positions in the target register and extends to the left. The target register field may or may not be zeroed before the copy. Again, the number of bits copied may be from an immediate value or the shift count register. If the count is from the SAR, the copied segment is truncated to prevent it from extending beyond the target field. If the count is an immediate, the operation is undefined if the copied segment extends beyond the target field.

Any of the above instructions would be useful in implementing register field or vector element accesses in a general-purpose model. Because of their wide variety, it is probably best to hide their differences beneath a layer of abstraction.

Usually, when one inserts a field of data into a register, one needs to ensure that the surrounding data is not modified. As we have seen, MAX has bit shift and insert instructions which perform this operation on various sizes of data. However, MVI's extract and insert instructions always clear the surrounding data. To deal with this issue, MVI has a set of instructions which clear a segment of data in a register without affecting the surrounding data. The result can then be logically ORed with the result of an insertion instruction thus inserting the selected field without affecting the surrounding data.

Two types of segment-clearing instructions are available in the MVI extensions. A "clear segment low" instruction clears the byte, word, doubleword, or quadword starting at a given byte (0 to 7, stored in a register or as an immediate), and going upwards through the register. A "clear segment high" clears the remainder of the bytes in the word, doubleword, or quadword which would have been chosen by the clear-segment-low given the same arguments and assuming the target of the clear-segment-high was concatenated to the high end of the target of the clear-segment-low.

Permutations are typically generalized to perform any of the possible rearrangements, with or without repetition, of the fields of their source operand(s). There are two primary methods in which the applied permutation can be chosen. One is via an immediate value which is specified at compile time. The other is via a variable vector index which may not be known until run time.

MAX-2, E3DNow!, SSE, and SSE2 have permute instructions which use an immediate value to indicate which fields of the single source to copy. SSE2 also has instructions which permute the lower or upper fields of a single source operand based on an immediate value. SSE, SSE2, and 3DNow!Pro also have permutations which select fields from two operands based on immediate index values. In contrast, AltiVec's permute uses a vector register to choose fields from two other vector registers to be copied to the destination register.

Permutes indexed via an immediate are useful for static data layout and element replication, but are not useful dynamically. Permutes indexed via a register can be used to implement dynamic constructs. An example is the MPL `router[`*exp1*`].`*exp2* construct in which *exp2* is evaluated on the PE whose number is equal to the evaluated value of *exp1*.

In this construct, *exp1* is an arbitrary expression. The permute operation could be quite useful here, but is much less so if it cannot be indexed by anything but a compile-time constant. Because so few of the extension families support any kind of permute at all, and because only AltiVec supports a variably-indexed permute, constructs such as the MPL `router` should be avoided for now.

Operations such as byte and word swaps are special cases of permutation. Enhanced 3DNow! includes an instruction to swap the two fields of a 2x32f partitioned register. Its operation is covered by E3DNow!'s more general permute instruction. Thus it is unnecessary, but may be temporally or spatially less expensive to execute than the equivalent permute.

2.1.8 Interleaving Operations

Table 2.15 lists the various instructions which interleave fields from two partitioned sources to form a combined result. In general, these instructions combine only certain fields from their sources to form their results.

Table 2.14
Data Extraction, Insertion, and Permutation Operations

Operation Types	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX
Extract Field to Reg.	-	-	-	-	-
Insert Selected Field	-	-	-	-	-
Insert Low Field	-	-	-	-	-
Byte Shft Rt & Extract					
By Immed.	8x8u→1x[8,16,32,64][1]	-	-	-	-
By Register	8x8u→1x[8,16,32,64][1]	-	-	-	-
Byte Shft Lt & Extract					
By Immed.	8x8u→1x[16,32,64][1]	-	-	-	2-8x8u→8x8u, 2-4x16s→4x16s
By Register	8x8u→1x[16,32,64][1]	-	-	2-2x32f→2x32f	2-8x8u→8x8u, 2-4x16s→4x16s
Byte Shft Rt & Insert into Zeroed Reg	1x[16,32,64]→8x8u[1]	-	-	-	-
Byte Shft Lt & Insert into Zeroed Reg	1x[8,16,32,64]→8x8u[1]	-	-	-	-
Bit Shft Lt & Extract[2]	-	1x32s[3], 1x32u[4] - -	right 2x32s[3], right 2x32u[4] 1x64s[3], 1x64u[4]	-	-
Merge, Bit Shft Rt & Extract	-	2-1x32→1x32	2-1x32→1x32, 2-1x64→1x64	-	-
Bit Shift Left & Insert into Zeroed Reg[5]					
from Immed	-	1x32	1x32, 1x64	-	-
from Reg	-	1x32	1x32, 1x64	-	-
Bit Shift Left & Insert into Unchanged Reg[5]					
from Immed	-	1x32	1x32, 1x64	-	-
from Reg	-	1x32	1x32, 1x64	-	-
Clear Segment Low	1,2,4,or 8 bytes	-	-	-	-
Clear Segment High	2,4, or 8 bytes	-	-	-	-
Permute					
Part/Indexed by Part	-	-	-	-	-
Part/Indexed by Imm	-	-	4x16	-	-
Swap Fields	-	-	-	-	-

[1][...] indicates that there are multiple separate instructions – one for each of the values listed.
[2]Also nullifies next instruction if condition is met.
[3]Sign-extended.
[4]Zero-extended.
[5]Also nullifies next instruction if condition is met.

Table 2.14 cont'd.
Data Extraction, Insertion, and Permutation Operations

Operation Types	Motorola AltiVec	Sun VIS	Intel MMX	Intel SSE	Intel SSE2
Extract Field to Reg.	-	-	-	$4x16^1 \rightarrow 1x32^2$	$8x16^3 \rightarrow 1x32^2$
Insert Selected Field	-	-	-	low $2x16^4 \rightarrow 4x16^1$	low $2x16^4 \rightarrow 8x16^3$
Insert Low Field	-	-	-	low 4x32f→4x32f	low 2x64f→2x64f
Byte Shft Rt & Extract					
By Immed.	2-16x8→16x8	-	-	-	-
By Register		2-8x8→8x8	-	-	-
Byte Shft Lt & Extract					
By Immed.	-	-	-	-	-
By Register	-	-	-	-	-
Byte Shft Rt & Insert into Zeroed Reg	-	-	-	-	-
Byte Shft Lt & Insert into Zeroed Reg	-	-	-	-	-
Bit Shft Lt & Extract[5]	-	-	-	-	-
Merge, Bit Shft Rt & Extract	-	-	-	-	-
Bit Shift Left & Insert into Zeroed Reg[6]					
from Immed	-	-	-	-	-
from Reg	-	-	-	-	-
Bit Shift Left & Insert into Unchanged Reg[6]					
from Immed	-	-	-	-	-
from Reg	-	-	-	-	-
Clear Segment Low	-	-	-	-	-
Clear Segment High	-	-	-	-	-
Permute					
Part/Indexed by Part	2-16x8→16x8	-	-	-	-
Part/Indexed by Imm	-	-	-	$4x16^7$, $2\text{-}4x32f \rightarrow 4x32f^6$	low $4x16^6$, high $4x16^6$, $4x32^6$, $2\text{-}2x64f \rightarrow 2x64f^8$
Swap Fields	-	-	-	-	-

[1]Field selected is (unsigned immediate mod 4).
[2]Zero-extended.
[3]Field selected is (unsigned immediate mod 8).
[4]From integer register
[5]Also nullifies next instruction if condition is met.
[6]Also nullifies next instruction if condition is met.
[7]Source fields selected by a 4x2 immediate.
[8]Source fields selected by a 2x1 immediate.

Table 2.14 cont'd.
Data Extraction, Insertion, and Permutation Operations

Operation Types	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Extract Field to Reg.	-	4x16[1]→1x32[2]	-	-
Insert Selected Field	-	low 2x16[3]→4x16[1]	-	-
Insert Low Field	-	-	low 2x32f→2x32f	-
Byte Shft Rt & Extract				
By Immed.	-	-	-	-
By Register	-	-	-	-
Byte Shft Lt & Extract				
By Immed.	-	-	-	-
By Register	-	-	-	-
Byte Shft Rt & Insert into Zeroed Reg	-	-	-	-
Byte Shft Lt & Insert into Zeroed Reg	-	-	-	-
Bit Shft Lt & Extract[4]	-	-	-	-
Merge, Bit Shft Rt & Extract	-	-	-	-
Bit Shift Left & Insert into Zeroed Reg[5]				
from Immed	-	-	-	-
from Reg	-	-	-	-
Bit Shift Left & Insert into Unchanged Reg[5]				
from Immed	-	-	-	-
from Reg	-	-	-	-
Clear Segment Low	-	-	-	-
Clear Segment High	-	-	-	-
Permute				
Part/Indexed by Part	-	-	-	-
Part/Indexed by Imm	-	4x16[6]		-
			2-4x32f→4x32f[6]	
Swap Fields	-	2x32f	-	-

[1]Field selected is (unsigned immediate mod 4).
[2]Zero-extended.
[3]From integer register
[4]Also nullifies next instruction if condition is met.
[5]Also nullifies next instruction if condition is met.
[6]Source fields selected by a 4x2 immediate.

VIS includes an instruction which interleaves the fields of two (N/2)xB sources to form a single NxB result. This is the only interleave in which the result contains all of the fields of its original operands. These operands are stored as 32-bit "pixel" data in 4x8u format. The interleaved result is stored in a 64-bit floating-point register in "fixed" format.

MAX-2 includes instructions for interleaving the odd numbered fields of the two source operands into a result value and others for interleaving the even-numbered fields.

Several of the extension families have instructions which interleave the upper (higher-numbered) fields of the two source operands into a single result and corresponding instructions which interleave the lower fields.

VIS includes an interleave instruction that scales (shifts), truncates, and clips (saturates) each of the fields of a 2x32 operand to a single byte. This is stored in the low byte of the corresponding field of the result and is zero-extended to obtain a 2x32u intermediate value. A second 2x32 operand is parallel left shifted by one byte to obtain a 2x32u intermediate value in which the low byte of each field is zeroed. These intermediate values are then merged via a bitwise-OR operation to form an 8x8u result.

Both MIPS-V and MDMX include instructions to interleave the even fields of one operand with the odd fields of the second. In MDMX, the second operand may be an immediate, a single-valued partitioned register, or a partitioned register. MDMX includes alternate forms of these instructions in which the order of the data fields in each of the operands is reversed before the interleave is performed. MIPS-V also includes an instruction to interleave the odd fields of the first operand with the even fields of the second.

While interleaves may be useful internally for implementing data layout, type cast, or vector element access operations, it is not clear that they should be exposed at the programming layer. More importantly, the forms are not universally implemented or consistent, and it may be difficult to emulate any particular form chosen for such a model.

2.1.9 Catenating, Packing, and Unpacking Operations

Table 2.16 lists the instructions available for catenating or unpacking SWAR data. These terms are not used consistently, so we will provide our own definitions here.

To *catenate* two partitioned values means to copy a subset of the fields of one to the upper half of the result and a subset of the fields of the other to the lower half while maintaining the relative ordering of these fields. Note that there is no requirement that the selected fields of either source be contiguous.

To *pack* a source operand means to compact a subset of its fields from 2B bits (or more generally, from some number of bits greater than B) to B bits, shifting the fields as necessary, while maintaining their relative ordering.

To *unpack* a source operand means to expand a subset of its fields from B bits to 2B bits (or more generally, to some number of bits greater than B), shifting the fields as necessary, while maintaining their relative ordering.

MDMX includes instructions which catenate either the odd fields or the even fields of two operands to form a partitioned result of the same layout. Each of these allow one of the operands to be an immediate value or replicated scalar. AltiVec includes instructions to catenate the even fields of two vector operands, but none for odd fields. MDMX also includes instructions which catenate either the upper or lower fields of their operands. Again, one of these may be an immediate or replicated scalar value. SSE includes a similar pair of instructions which operate on partitioned register operands.

Because these forms of catenation are not universally implemented, one may wish to exclude catenations from a general-purpose programming model. However, multi-word length vectors would not normally be catenated on a per word basis, but by copying the fragments of one operand after those of the other. Thus, the lack of

Table 2.15
Interleaving Operations

Operation Types	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX	Motorola AltiVec	Sun VIS
Interleave (Merge)	-	-	-	-	-	-	2-4x8u→8x8u
Interleave odd (left)	-	-	4x16, 2x32	-	-	-	-
Interleave even (right)	-	-	4x16, 2x32	-	-	-	-
Interleave upper							
Part/Part	-	-	-	2x32f	8x8, 4x16	16x8, 8x16, 4x32	-
Part/Imm	-	-	-	-	8x8, 4x16	-	-
Part/Scalar	-	-	-	-	8x8, 4x16	-	-
Part/Zero	-	-	-	-	8x8	-	-
Interleave lower							
Part/Part	-	-	-	2x32f	8x8, 4x16	16x8, 8x16, 4x32	-
Part/Imm	-	-	-	-	8x8, 4x16	-	-
Part/Scalar	-	-	-	-	8x8, 4x16	-	-
Part/Zero	-	-	-	-	8x8	-	-
Scale, Trunc, Clip & Merge[1]	-	-	-	-	-	-	2-2x32→8x8u
Interleave even w/odd Forward or Reverse							
Part/Part	-	-	-	2x32f	4x16	-	-
Part/Imm	-	-	-	-	4x16	-	-
Part/Scalar	-	-	-	-	4x16	-	-
Interleave odd w/even Forward or Reverse							
Part/Part	-	-	-	2x32f	-	-	-
Part/Imm	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-

[1]Left shifts logically by 8 bits an 8x8u, then takes a 2x32, left shifts it logically by the GSR value, truncates the lower 23 bits of each field to form a 2x24, then unsigned saturates it to a 2x8u which is then ORed with the 8x8u register.

Table 2.15 cont'd.
Interleaving Operations

Operation Types	Intel MMX	Intel SSE	Intel SSE2	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Interleave (Merge)	-	-	-	-	-	-	-
Interleave odd (left)	-	-	-	-	-	-	-
Interleave even (right)	-	-	-	-	-	-	-
Interleave upper							
Part/Part	8x8, 4x16, 2x32	4x32f	16x8, 8x16, 4x32, 2x64,2x64f	-	-	- 2x32f	-
Part/Imm	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-
Part/Zero	-	-	-	-	-	-	-
Interleave lower							
Part/Part	8x8, 4x16, 2x32	4x32f	16x8, 8x16, 4x32, 2x64,2x64f	-	-	- 2x32f	-
Part/Imm	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-
Part/Zero	-	-	-	-	-	-	-
Scale, Trunc., Clip & Merge[1]	-	-	-	-	-	-	-
Interleave even w/odd Forward and Reverse							
Part/Part	-	-	-	-	-	-	-
Part/Imm	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-
Interleave odd w/even Forward and Reverse							
Part/Part	-	-	-	-	-	-	-
Part/Imm	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-

[1]Left shifts logically by 8 bits an 8x8u, then takes a 2x32, left shifts it logically by the GSR value, truncates the lower 23 bits of each field to form a 2x24, then unsigned saturates it to a 2x8u which is then ORed with the 8x8u register.

universality should not inhibit the designer from including vector catenations in a general-purpose model.

Only a few types of instructions meet the above definition of a pack. These come in various forms. Some are general-purpose, while others are intended for specific operations such as converting data to proprietary pixel formats. These instructions are also probably best used internally, within the implementation of a model, and probably should not be visible to a high-level programmer.

AltiVec and MMX each include instructions which pack the fields of their two operands to half-size using signed or unsigned saturation. These intermediate values are then catenated to form a single partitioned result. SSE2 extends the MMX instructions for use on the SSE integer set.

AltiVec also includes an instruction which converts data from two partitioned operands to a pixel format. The pixel data is then concatenated and stored in a partitioned destination register. This proprietary operation should not be made part of a programming model which is intended to be portable, but may be useful for implementing type casts or other operations.

MVI includes instructions which truncate the fields of a register to a single byte by discarding the upper bits, then copy the resulting fields into the low end of the result register. These instructions maintain the relative ordering of the fields and zero any unused fields.

VIS includes special-purpose instructions for scaling and packing graphics data in pixel format. These instructions logically shift each field left by the scale factor (0 to 15 bits) in the UltraSPARC's Graphics Status Register (GSR). These values are then rounded by truncating the bits lower than an implicit binary point (bits 0-6 for a 16-bit field, bits 0-15 for a 32-bit field). Finally, they are saturated to fit in the fields of the result. The GSR can be manipulated with the "rd" and "wr" instructions to change the applied scaling factor. The operation performed by this instruction is obviously too specialized for general-purpose processing.

Several instructions for unpacking or expanding data fields are also available in the various extension families. These instructions are most likely to be useful for implementing type casts in a general-purpose programming model or for internally converting unsupported data types to supported ones for emulation purposes.

Both MDMX and AltiVec include instructions which copy the lower N/2 fields of an NxB partitioned register to a destination register, maintaining their relative order, then sign-extend the data to form an N/2xB result. Complementary instructions which unpack the upper fields of their sources are also available in each of these extension families.

MVI includes unpacks which complement its "pack low byte" instructions. These copy the data from the lower fields (bytes) of the source register to the destination register starting with the lowest numbered field. Data is zero-extending as needed to fill the larger fields of the destination register.

Two instructions are included in AltiVec which complement its pixel-packing instruction. These convert packed pixels back to an unpacked form. One unpacks the lower fields of the packed pixel while the other unpacks its upper fields. These proprietary operations should not be made part of a programming model which is intended to be portable.

VIS also includes an instruction which unpacks the lower fields of one NxB operand to a Nx2B result in which the original B data bits are centered in each field and the surrounding bits are cleared. This instruction is intended to complement VIS's pixel-packing instruction, but is more generally useful because it leaves the data intact (although shifted).

2.1.10 Memory Access Instructions

Table 2.17 lists memory access instructions that may be useful for SWAR processing and are available for use by the various extension families. Each of these families has some means of accessing memory. Some include new instructions for loads and

Table 2.16
Catenating, Packing, and Unpacking Operations

Operation Types	DEC MVI	HP MAX	SGI MIPS-V	SGI MDMX	Motorola AltiVec
Catenate odd					
Part/Part	-	-	-	8x8, 4x16	-
Part/Imm	-	-	-	8x8, 4x16	-
Part/Scalar	-	-	-	8x8, 4x16	-
Catenate even					
Part/Part	-	-	-	8x8, 4x16	16x8, 8x16
Part/Imm	-	-	-	8x8, 4x16	-
Part/Scalar	-	-	-	8x8, 4x16	-
Catenate upper					
Part/Part	-	-	-	4x16	-
Part/Imm	-	-	-	4x16	-
Part/Scalar	-	-	-	4x16	-
Catenate lower					
Part/Part	-	-	-	4x16	-
Part/Imm	-	-	-	4x16	-
Part/Scalar	-	-	-	4x16	-
Unsigned Saturate, Pack, and Catenate	-	-	-	-	2-8x16s→16x8u,2-8x16u→16x8u, 2-4x32s→8x16u,2-4x32u→8x16u
Signed Saturate, Pack, and Catenate	-	-	-	-	2-8x16s→16x8s, 2-4x32s→8x16s
Pixel Pack and Catenate	-	-	-	-	2-4x32→8x16
Truncate & Pack Low Byte	2x32→8x8 4x16→8x8	-	-	-	-
Scale, Truncate, & Clip	-	-	-	-	-
Unpack Lower & Sign Extend	-	-	-	8x8u→4x16s	16x8s→8x16s, 8x16s→4x32s
Unpack Upper & Sign Extend	-	-	-	8x8u→4x16s	16x8s→8x16s, 8x16s→4x32s
Unpack Low Bytes & Zero Extend	8x8u→2x32 8x8u→4x16	-	-	-	-
Unpack Lower Pixel	-	-	-	-	8x16→16x8
Unpack Upper Pixel	-	-	-	-	8x16→16x8
Zero Expand	-	-	-	-	-

Table 2.16 cont'd.
Catenating, Packing, and Unpacking Operations

Operation Types	Sun VIS	Intel MMX	Intel SSE	Intel SSE2
Catenate odd				
Part/Part	-	-	-	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
Catenate even				
Part/Part	-	-	-	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
Catenate upper				
Part/Part	-	-	4x32f	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
Catenate lower				
Part/Part	-	-	4x32f	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
Unsigned Saturate, Pack, and Catenate	-	2-4x16s→8x8u	-	2-8x16s→16x8u
Signed Saturate, Pack, and Catenate	-	2-4x16s→8x8s, 2-2x32s→4x16s	-	2-8x16s→16x8s, 2-4x32s→8x16s
Pixel Pack and Catenate	-	-	-	-
Truncate & Pack Low Byte	-	-	-	-
Scale, Truncate, & Clip	4x16→4x8u, 2x32→2x16s[1]	-	-	-
Unpack Lower & Sign Extend	-	-	-	-
Unpack Upper & Sign Extend	-	-	-	-
Unpack Low Bytes & Zero Extend	-	-	-	-
Unpack Lower Pixel	-	-	-	-
Unpack Upper Pixel	-	-	-	-
Zero Expand	4x8u→4x16u	-	-	-

[1]Takes a 2x32, left shifts it logically by the GSR value, truncates the lower 16 bits to form a 2x31, then signed saturates it to a 2x16s.

Table 2.16 cont'd.
Catenating, Packing, and Unpacking Operations

Operation Types	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Catenate odd				
Part/Part	-	-	-	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
Catenate even				
Part/Part	-	-	-	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
Catenate upper				
Part/Part	-	-	2x32f	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
Catenate lower				
Part/Part	-	-	2x32f	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
Unsigned Saturate, Pack, and Catenate	-	-	-	-
Signed Saturate, Pack, and Catenate	-	-	-	-
Pixel Pack and Catenate	-	-	-	-
Truncate & Pack Low Byte	-	-	-	-
Scale, Truncate, & Clip	-	-	-	-
Unpack Lower & Sign Extend	-	-	-	-
Unpack Upper & Sign Extend	-	-	-	-
Unpack Low Bytes & Zero Extend	-	-	-	-
Unpack Lower Pixel	-	-	-	-
Unpack Upper Pixel	-	-	-	-
Zero Expand	-	-	-	-

stores, while others use the existing access instructions, and still others allow memory operands to their instructions.

The *alignment* of data in memory may have to be accounted for as some architectures cannot perform unaligned accesses while others prevent unaligned accesses by auto-aligning them. An *aligned* access is one in which the address is divisible by the number of bytes accessed, N. Such an access is referred to as being "aligned on an N-byte boundary". An *unaligned* access is one in which the address is not on an N-byte boundary. An *auto-aligned* access is one in which the least significant bits of the address are ignored; thus, the effective address is aligned on some non-minimal boundary even if the requested address was not.

When operating on long vectors of data, one would normally load as much of the vector as possible in order to maximize parallelism. In this case, one would perform a load of a word-sized *fragment* of the vector. The entire fragment would then be operated on, then stored with a word-sized store. For long vectors, alignment need only be an issue when accessing the first and last fragments of the vector.

By contrast, when loading a single element using a word-sized load, the entire enclosing memory fragment is loaded. The fragment may need to be shifted to justify the proper element within the register. Then, the element must be zero- or sign-extended to fill the register and clear out the surrounding data.

When storing a single field value to a particular vector element in memory, the value must be aligned with the element's position in the corresponding memory fragment, then stored without affecting the surrounding data. This is usually accomplished by loading the fragment from memory, masking out the old data in the element's position, shifting the new data to this position in another register, combining these via a bitwise OR, and finally storing the updated fragment back to memory.

When copying one element from a vector in memory directly into another, we would like to load the element, shift it into position, then store it without affecting the surrounding data. In practice, the element is typically loaded, converted to a single value by justifying it with the least significant bit (lsb), sign- or zero- extended to fill the register, then stored as in the previous paragraph.

Element load and store instructions, which move a single field of data, are only useful if they eliminate some of the above masking or alignment steps. Otherwise, they are not an improvement over full-sized load or stores except in special-case situations.

Any implementation of a portable processing model will have to be implemented on any of the target systems. Each extension family has its own set of peculiarities with regards to memory accesses. These are usually inherited from the memory system of the underlying architecture either by necessity or by convention.

MVI uses the memory access instructions of the underlying Alpha architecture. These include a set of 8-, 16-, 32- and 64- bit loads and stores which require aligned accesses, and 64-bit loads and stores which do not.

The MAX extension sets also use the memory accesses of their underlying architectures. In each case, a set of 8-, 16-, and 32- bit aligned loads and stores are included, as are instructions to store from one to four or one to eight bytes to an unaligned address. The 64-bit MAX-2 also includes 64-bit aligned loads and stores.

MIPS-V includes 64-bit auto-aligned loads and stores which are also used by the MDMX family of extensions. The underlying MIPS-IV memory-access instructions are also available to both MIPS-V and MDMX.

AltiVec includes 8-, 16-, and 32- bit aligned element loads and stores. The element loads load the data into a vector register in the same relative position that it occupies in the aligned memory quadword (128-bits) which contains it, making the surrounding bits undefined. The element stores store the data from a vector register into the aligned memory quadword (128-bits) which contains the address in the same relative position that it occupies in the vector register, without affecting the surrounding bits.

The AltiVec Technology Programming Environments Manual [68] is inconsistent in its description of vector element loads (lvebx, lvehx, lvewx). In table 4-15, they are described as loading the data into the low-order bits of the target vector register, with the remaining bits “set to boundedly undefined values”. In the individual descriptions of these instructions (which are usually more accurate), they are described as loading the data into the same relative position within the target vector register as its relative position in the quadword (128 bits) that it occupies in memory.

AltiVec also includes 128-bit auto-aligning loads and stores and two “load vector index” instructions which are used for obtaining unaligned data. These instructions load a predefined constant vector value into a register and rotate it left or right by zero to sixteen bytes, depending on the address requested. When the same address is used with a load, it is auto-aligned and returns the aligned fragment that contains the requested address. The index vector is then used as the index to a permute instruction which aligns the retrieved fragment. If the requested address was unaligned, this process must be repeated for an access of the next aligned fragment in memory. The results are then combined to form the intended unaligned access.

VIS includes aligned 8- and 16- bit loads and stores. It also includes block loads and stores which move an aligned block of 64 bytes between memory and an aligned set of eight consecutive floating-point registers without altering the cache. There is also a variation of the block store which forces a cache flush.

MMX includes an unaligned 32-bit move instruction which can also be used to load or store 32-bit data between the integer registers and the MMX registers. A 64-bit unaligned move is also included which can load or store data between memory and an MMX register or between two MMX registers. These same instructions are used by all IA32-based extension families.

SSE includes several memory access instructions. One instruction moves 128-bits of aligned data between memory and an SSE register or between two SSE registers as a set of 32-bit floats. There is also an unaligned version of this instruction. 3DNow!Pro has aligned and unaligned versions of this for the MMX register set, while SSE2 has 64-bit aligned and unaligned floating-point versions.

Another set of SSE instructions moves pairs of unaligned 32-bit floating-point data between memory and either the upper or lower halves of an SSE register without affecting the surrounding data. In order to maintain compatibility with SSE, 3DNow! Professional must provide similar functionality, although it isn't clear what form this would take. SSE2 provides similar instructions for operating on 64-bit floats.

Another SSE instruction moves 32 bits of float data between memory and the low field of an SSE register and also clears the upper fields. This same instruction can store data from the low 32-bit field of an SSE register to memory without affecting surrounding data. 3DNow! Professional contains an equivalent instruction, while SSE2 contains a set for 32-bit integer, 64-bit integer, and 64-bit floating-point data.

SSE2 also has a 2x64 integer aligned load and a corresponding unaligned load. It also contains complementary stores and a complementary store which generates a non-temporal hint.

Enhanced 3DNow! and SSE each include a 64-bit store which is intended to minimize cache pollution when storing data from an MMX register. SSE also includes a non-polluting partitioned 32-bit floating-point store from an SSE register. An MMX register version of this instruction is available in 3DNow!Pro. SSE2 rounds these out with a 64-bit floating-point SSE register version and a 32-bit instruction which stores data from an integer register. Each of these instructions generates a cache-management hint that the data is “non-temporal”.

One instruction found in Enhanced 3DNow! and SSE loads a selected 16-bit field in an MMX register from memory without affecting the surrounding fields. It can also be used as an insert instruction which takes its source data from an integer register. SSE2 extends this instruction for use with SSE registers.

The loading of immediate values is often handled in interesting ways. For example, the MAX family of extensions use the PA-RISC “load offset” instructions which are primarily intended for calculating and loading indexed addresses for memory accesses. MAX also has available an instruction which can load a 21-bit immediate, shifted by 11 bits, into a 32-bit register. This instruction is intended for address generation, but can also be used to load immediate values for computation. The MAX-2 version sign-extends the loaded data to fill a 64-bit register. VIS includes instructions which can load ’0’ or ’1’ bits into all of the bits of a 32- or 64- bit floating-point register thus supplying a means of loading these commonly-used values (0 and -1).

Various extensions also include “masked store” instructions. These store the fields of a partitioned register based on the value of a corresponding bit in a bitmask. VIS includes 8-, 16-, and 32- bit masked store instructions in which this bitmask is stored in an integer register, typically generated by a comparison instruction. Enhanced 3DNow! and SSE include an 8-bit instruction in which this bitmask consists of the most significant bits (MSbs) of each byte of an 8x8 partitioned operand. A byte from the source operand is stored if the MSb of the corresponding byte in the second operand is a ‘1’. SSE2 includes a version of this for use with the SSE registers.

MVI, Enhanced 3DNow!, and SSE each include a store synchronization (store sync) instruction which ensures that stores preceding the synchronization point in program order complete before stores which follow. This is known as *weak synchronization* because the order of every pair of stores is not necessarily maintained. That is, two stores which are scheduled before the synchronization point may be reordered. Only the order of stores occurring before the synchronization point versus those occurring after it are enforced.

SSE2 also includes a load synchronization instruction which ensures that loads which precede the synchronization point complete before loads which follow it. Furthermore, SSE2 includes a memory synchronization instruction which ensures that all loads and stores which precede the synchronization point complete before any loads or stores which follow it. MAX has a similar instruction which weakly enforces the order of all memory accesses including loads and stores and semaphore, cache flush, and cache purge instructions.

Although it isn’t really a memory access instruction, SSE2 also includes a spin-wait hint instruction that lets the processor know that the process is executing a spin-lock loop. These loops are typically used to synchronize processes that are in contention for some shared resource or to block a process until some condition is met. The Pentium 4 processor would normally detect such a loop and treat it as a "memory order violation" [95]. This hint is used to suggest to the processor that it ignore the supposed violation.

2.1.11 Cache Management Instructions

Table 2.18 lists the instructions available for supporting cache management. While these are not strictly SWAR operations, intelligent use of the memory subsystem is a necessity on current SWAR architectures to achieve speedup. Generally, the programmer should be unaware of these issues, so cache management should be handled internally by the compiler. Cache management is also rarely portable between architectures, so these operations should not be made visible by a portable programming model.

As a general rule, data prefetches are auto-aligned. That is, when a prefetch specifies a particular address, the aligned line-sized memory block is brought into the cache. Some older architectures allow unaligned prefetches which bring in the memory block that starts at the requested address.

HP's MAX-2 allows simple prefetching to be done using the standard load instructions by targeting the read-only general register 0. The block to be fetched lies at the auto-aligned value of the requested address. For write accesses, the block may be marked dirty upon being fetched. 3DNow! includes similar instructions which fetch a 32-byte block, but whose address may or may not be auto-aligned, depending on the underlying architecture.

The PA-RISC architecture's load and store instructions also take a "cache hint completer" (i.e. an opcode extension) which indicates a suggested action to take relating to the cache. One hint indicates that the data will only be used once (i.e. has that it has spatial locality, but not temporal locality). Hence, the data can be loaded into a buffer rather than into the cache, thus preventing the cache from becoming polluted by the temporary data.

Table 2.17
Memory Access Operations

Operation Types	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX	Motorola AltiVec
Load Aligned[1]	1x8u→1x64u, 1x16u→1x64u, 1x32s→1x64s, 1x64,	1x8u→1x32u, 1x16u→1x32u, 1x32u	1x8u→1x64u, 1x16u→1x64u, 1x32u→1x64u, 1x64	1x64[2]	-	1x8, 1x16, 1x32, 1x128[2]
Load Unaligned[1]	1x64	-	-	-	-	-
Load Field	-	-	-	-	-	-
Load Immediate	-	1x21→1x64[3]	1x21→1x64[3]	-	-	-
Load Zeros	-	-	-	-	-	-
Load All Ones	-	-	-	-	-	-
Load Alignment Vector	-	-	-	-	-	1x128
Store Aligned	1x64→1x8, 1x64→1x16, 1x64→1x32, 1x64,	1x32→1x8, 1x32→1x16, 1x32	1x64→1x8, 1x64→1x16, 1x64→1x32, 1x64	1x64[2]	-	1x8, 1x16, 1x32, 1x128[2]
Store Unaligned	1x64	1to4x8	1to4x8 1to8x8	-	-	-
Store Aligned w/Cache Flush	-	-	-	-	-	-
Masked Store						
by Bitmask	-	-	-	-	-	-
by MSb of Part	-	-	-	-	-	-
Store Sync	Weak	-	-	-	-	-
Load Sync	-	-	-	-	-	-
Memory Sync	-	Weak	-	-	-	-
Spin-wait Hint	-	-	-	-	-	-

[1]Unsigned type implies zero-extension. Signed type implies sign-extension.
[2]Auto-aligning.
[3]Data shifted left by 11 bits, then sign extended to left into upper 32 bits.

Table 2.17 cont'd.
Memory Access Operations

Operation Types	Sun VIS	Intel MMX	Intel SSE	Intel SSE2
Load Aligned[1]	1x8u→1x64u, 1x16u→1x64u, 64x8→8-1x64	-	4x32f	2x64,2x64f
Load Unaligned[1]	-	1x32u→1x64u 1x64	1x32f→low 4x32f[2], 2x32f→upper 4x32f[3], 2x32f→lower 4x32f[4], 4x32f	1x32→low 4x32[2], 1x64→low 2x64[2],1x64→low 2x64f[2], 1x64f→high 2x64f[3], 1x64f→low 2x64f[4], 1x128,2x64f
Load Field	-	-	1x16→4x16[5]	1x16→8x16[6]
Load Immediate	-	-	-	-
Load Zeros	1x32, 1x64	-	-	-
Load All Ones	1x32, 1x64	-	-	-
Load Alignment Vector	-	-	-	-
Store Aligned	1x64→1x8, 1x64→1x16, 8-1x64→64x8		1x64[8], 4x32f,4x32f[8]	1x32[7] 2x64,2x64[8], 2x64f,2x64f[8]
Store Unaligned	-	low 2x32→1x32 1x64	low 4x32f→1x32f, upper 4x32f→2x32f, lower 4x32f→2x32f, 4x32f	low 4x32→1x32, low 2x64→1x64,low 2x64f→1x64, high 2x64f→1x64f, low 2x64f→1x64f, 2x64,2x64f
Store Aligned w/Cache Flush	8-1x64→64x8	-	-	-
Masked Store				
by Bitmask	8x8, 4x16, 2x32	-	-	-
by MSb of Part	-	-	8x8	16x8
Store Sync	-	-	Weak	-
Load Sync	-	-	-	Weak
Memory Sync	-	-	-	Weak
Spin-wait Hint	-	-	-	Yes

[1]Unsigned type implies zero-extension. Signed type implies sign-extension.
[2]High fields cleared.
[3]Low field(s) left unchanged.
[4]High field(s) left unchanged.
[5]Field selected is (immediate mod 4).
[6]Field selected is (immediate mod 8).
[7]Data from integer register is stored with a non-temporal hint.
[8]With Non-temporal hint.

Table 2.17 cont'd.
Memory Access Operations

Operation Types	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Load Aligned	-	-	2x32f	-
Load Unaligned	-	-	1x32f$\rightarrow$low 2x32f 1x64$\rightarrow$1x32f 1x64$\rightarrow$1x32f 2x32f	-
Load Field	-	1x16$\rightarrow$4x16[1]	-	-
Load Immediate	-	-	-	-
Load Zeros	-	-	-	-
Load All Ones	-	-	-	-
Load Alignment Vector	-	-	-	-
Store Aligned	-	1x64[2]	2x32f,2x32f[2]	-
Store Unaligned	-	-	low 2x32f$\rightarrow$1x32f, 2x32f$\rightarrow$1x64, 2x32f$\rightarrow$1x64, 2x32f	-
Store Aligned w/Cache Flush	-	-	-	-
Masked Store				
by Bitmask	-	-	-	-
by MSb of Part	-	8x8	-	-
Store Sync	-	Weak	-	-
Load Sync	-	-	-	-
Memory Sync	-	-	-	-
Spin-wait Hint	-	-	-	-

[1]Field selected is (immediate mod 4).
[2]With Non-temporal hint.

MVI and AltiVec include instructions which issue a “prefetch hint” or “store hint” indicating that the data block should be prefetched because it probably will be loaded from or stored to, respectively. AltiVec also includes versions of these instructions which hint that the data should not be cached because it is expected to be “transient”. That is, that it won’t be accessed many times after the load or store is completed. MVI has a separate store hint for transient data.

The AltiVec prefetch instructions also associate a strided data stream with an identifying number. This identifier, which ranges from in value from 0 to 3, is used to indicate from which stream data should be prefetched. Whenever a stream is associated with an identifier, all associations it has with other identifiers are removed.

A separate instruction is included to disassociate an identifier from its associated stream without associating it with another. Another instruction disassociates all identifier/stream pairs. These are apparently the only instructions in any of the extension families which take non-unit, variable, strided accesses into account.

Enhanced 3DNow! and SSE each include a set of instructions which hint that a 32-byte block should be prefetched and also to which cache level the data should be sent. This allows the programmer to treat the memory system in a more hierarchical manner than a simple hint would.

MVI also includes an “evict hint” which indicates that a particular cache line would be a good choice for removal (eviction) from the cache because it will not be accessed in the near future. This instruction may initiate a write-back of the cache line if it is dirty.

SSE2 includes a “flush line” instruction which causes the specified cache line to be flushed to memory, thus cleanly freeing it for future use. MAX-1 includes instructions for flushing the data and instruction caches if they are separate entities or the combined cache if not. These instructions write the flushed line back to memory if it is dirty.

MAX-1 also includes instructions which will flush an entire cache, writing lines back if they are dirty. It also includes an instruction which “purges” a data cache

Table 2.18
Cache Management Operations

Operation Types	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX	Motorola AltiVec	Sun VIS
Prefetch Data Line	-	-	Yes	-	-	-	-
Prefetch Data Line for Write	-	-	Yes	-	-	-	-
Prefetch Hint	512 bytes[1]	-	-	-	-	Yes[2]	-
Prefetch Hint Transient	-	-	-	-	-	Yes[2]	-
Store Hint	512 bytes[1]	-	-	-	-	Yes[2]	-
Store Hint Transient	64 bytes	-	-	-	-	Yes[2]	-
Disassociate ID and Stream(s)	-	-	-	-	-	Single or All	-
Evict Hint	Yes	-	-	-	-	-	-
Flush Line	-	Data,Instr.	-	-	-	-	-
Purge Line	-	Data	-	-	-	-	-
Flush Cache	-	Data,Instr.	-	-	-	-	-

Operation Types	Intel MMX	Intel SSE	Intel SSE2	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Prefetch Data Line	-	-	-	32 bytes[3]	-	-	-
Prefetch Data Line for Write	-	-	-	32 bytes[3]	-	-	-
Prefetch Hint	-	32 bytes[4]	-	-	32 bytes[4]	-	-
Prefetch Hint Transient	-	-	-	-	-	-	-
Store Hint	-	-	-	-	-	-	-
Store Hint Transient	-	-	-	-	-	-	-
Disassociate ID and Stream(s)	-	-	-	-	-	-	-
Evict Hint	-	-	-	-	-	-	-
Flush Line	-	-	Yes	-	-	-	-
Purge Line	-	-	-	-	-	-	-
Flush Cache	-	-	-	-	-	-	-

[1]A subset or superset of the requested block may be moved.
[2]Defines the data stream by specifying up to 256 units, of size up to 32 bytes, strided by up to 32768 bytes. Also associates an identifier number between 0 and 3 with the data stream.
[3]Unaligned on some architectures.
[4]Allows hint as to which cache level to prefetch to: t0 = all cache levels; t1 = all cache levels except 0th; t2 = all cache levels except 0th and 1st; nta = non-temporal cache structure.

line. Depending on the implementation, this instruction may skip the write-back of a dirty line when executed by a level-0 (i.e. a privileged) instruction.

2.2 Summary of Multimedia Extension Sets on General-Purpose Microprocessors

In this section, the salient features of the studied extension families are reviewed. This should help us to understand the relative strengths and weaknesses of each to allow for better design in the future.

None of these extension families appears to be an attempt to develop a high-level SWAR processing model. Support is usually limited to 8-, 16-, and 32-bit field sizes, and is usually not consistently available across these sizes. Instructions tailored to specific algorithms are often available; however, instructions for interfield communication, conditional parallel execution, and partial result combination usually are not.

The range of support provided by these families varies widely, with some including a large number of SWAR operations, while others include only a few. There is usually some support for basic modular (wrap-around) and saturating arithmetic, data layout, and data repackaging between integer and partitioned storage formats. Basic bitwise, condition testing, and communication operations are sometimes supported, though some families do so via standard integer operations rather than as part of the extension family.

2.2.1 MVI

From a review of the *Alpha Architecture Handbook* [60], it is obvious that the designers of MVI had a particular set of target algorithms in mind when choosing the instructions to include, and were not attempting to develop a high-level SWAR programming model. In fact, the stated goal of Digital Equipment Corporation's MVI extensions is to "...enable support for graphics and video algorithms".

Because MVI uses the Alpha's integer registers, its standard integer instructions are available to the SWAR programmer. This means that the polymorphic, shift, data layout, and memory access instructions are directly usable and are available for emulating partitioned operations.

MVI is weak in arithmetic instructions, having only a pixel error instruction which performs an 8x8u$\rightarrow$1x64u reduce-add of the absolute differences of the corresponding elements of two sources. Other instructions must be emulated using the standard integer arithmetic instructions.

MVI includes a reasonable set of partitioned maximum and minimum instructions which are useful for emulating saturation operations which are not directly supported.

MVI does not include partitioned multiplication or division operations, so these must be emulated using shifts, adds, and subtracts. A standard set of 64-bit integer shifts are included in the Alpha architecture, including shift-and-add and shift-and-subtracts. Partitioned shifts are not included, but can be emulated using the standard shifts.

A reasonable set of polymorphics is supported by the Alpha architecture which makes it possible to emulate many unsupported instructions. Also, instructions which perform a population count, a count of leading 0 bits, and a count of trailing 0 bits eliminate the need to perform a reduction in software to test global conditions such as ANY or ALL true.

MVI is also weak in the number and type of comparison operations it includes. The only partitioned comparison included is an 8x8u greater than or equal test. This instruction generates a bitmask with ones in the bits representing the fields where the condition is true, and zeroes in the others. The “zap” and “zapnot” instructions can then be used to easily mask the set of true or false fields as needed. This is a reasonable solution to providing enable masks for conditionals, but the single testing type is too restrictive to be of much value.

The standard integer equality test can be used as a global test (ALL equal), but is only useful for emulating partitioned tests if they are serialized. The 64-bit “beq” and “bne” instructions can be used as branch on none- or any- true, respectively, as can the “cmoveq” and “cmovne” instructions. These may be useful for SWAR looping constructs because the decision on whether to enter the body is aggregated across the fields of a fragment rather than being separate for each field.

MVI contains several shift instructions for inserting and extracting fields from a partitioned register. Also included are several instructions for clearing the upper or lower sections of a register. These instructions are useful for several types of data layout and rearrangement operations and allow data movement to or from a particular field in a low number of steps.

Operations for packing 16- and 32- bit fields to 8-bit fields are available, as are operations for unpacking 8-bit fields to 16- and 32- bit fields, but not between 16- and 32- bit fields. Thus, packing 32-bit fields to 16-bit fields requires packing to 8 bits, then unpacking to 16 bits.

The Alpha architecture also includes a set of load and store instructions which allow 8-, 16-, 32-, and 64- bit operations. The loads always write into the least significant end of the register and zero- or sign- extend the data into the rest of the register. This prevents direct loading of a field without disturbing the rest of the fragment, but allows fields to be loaded individually then ORed together to form the partitioned fragment. Stores always write the least significant end of the register to memory without disturbing the remainder of the word, thus allowing nearly direct field stores. A store synchronization instruction is available to flush currently pending stores.

A set of cache prefetch hints are also included which give the programmer some control over the operation of the cache. This control may be useful when operating on long vectors.

MVI is by far the weakest of the stand-alone extensions. Most general-purpose SWAR operations would have to be emulated if only MVI is available.

2.2.2 PA-RISC MAX-1

MAX-1 arithmetic instructions include 16-bit modular and signed saturating addition and subtraction. Unsigned saturating addition and subtraction are also supported, but these combine a signed operand with an unsigned one to form an unsigned saturated result. This makes pure unsigned saturation arithmetic difficult, because it forces the programmer to offset one operand and perform corrections to the saturation step.

The lack of reduction operations makes a fundamental step in task-based SIMD arithmetic algorithms expensive because it must be emulated. This is normally done using shifts and parallel instructions. These are supported to some extent, so emulation is possible for reduction operations, but will be expensive.

Also lacking are instructions which result in the upper half of the result of an addition or subtraction. While this is not a problem in itself, these instructions are sometimes useful for emulating unsupported operations.

MAX-1 does not contain partitioned maximum or minimum operations. This has two consequences. First, these operations must be emulated if they are included in a SWAR programming model. Second, they cannot be used to emulate unsupported saturation operations. This compounds the effects of not having pure unsigned saturation arithmetic instructions.

Multiplication and division by integer or fractional constants is supported using shift-and-add instructions. These perform a left or right shift of one 16-bit operand followed by signed saturation addition with the other. Because they are only intended to support multiplication and division, these shifts are limited to one-, two-, or three-bit counts, and are equivalent to performing a multiplication or division by a value of two, four, or eight.

These shift-and-operate instructions are more general than simple shifts, but their limitations make them less useful than they could be. In general, high-level language shift operations would be constructed as a series of these instructions. This may require non-trivial compiler technology to implement.

A 16-bit unsigned average instruction which rounds its result to an odd value is also available. This is intended to optimize graphics algorithms, such as smoothing, which use this common operation. It is less useful for arithmetic algorithms.

Because MAX-1 instructions are performed on the integer register set, the PA-RISC's standard bitwise operations are available for use in SIMD masking and the emulation of more complex operations. A sufficient working set is provided to allow the emulation of any required operation.

MAX's bitwise instructions can also be used to test a condition and nullify the next instruction in the program. This instruction is usually an unconditional jump. Thus, the construct emulates a branch, and is likely to be most useful for tests on the aggregate condition of the fields in a partitioned object.

Partitioned conditional instructions in MAX-1 are limited to a handful of "unit" tests which perform an operation such as XOR or add-with-complement, then null the next instruction if an ANY or ALL test fails. There are no tests which generate a bitmask or fieldmask that could be used for SIMD enable masking. This is a significant disadvantage.

However, MAX-1 does include a fairly rich set of full-register test, branch, load, and null-next instructions. These are arguably more useful in a SWAR environment than the few unit operations which are included because they test aggregate conditional information. However, as aggregate tests, their useful operation often overlaps. Thus, from the stand-point of SWAR processing, MAX-1's set of conditional instructions is not as rich as is appears to be at first glance.

The implementation of operations which mix scalar and partitioned data often requires that the scalar object be replicated to form a partitioned object upon which the actual operation is performed. MAX-1 does not provide any means of performing this replication step, so it would have to be emulated if mixed expressions are allowed in the high-level programming model.

MAX-1 also includes shift-left-and-extract instructions which allow an arbitrary number of bits to be extracted from a partitioned register starting at an arbitrary bit position in the register. This would be used to extract a field of data from within a partitioned register and thus to implement vector element accesses in a vector processing model.

MAX-1's bit shift-right-and-extract instructions shift the concatenation of two n-bit registers by up to n bits, then extract the low n bits into the destination register. This is intended to be used for data alignment before or after an unaligned memory access, but could be used for vector shifts, in which elements are shifted between fields.

A set of bit-shift-left-and-insert instructions allow 32 or 64 bits to be extracted from a source register and inserted into an arbitrary bit position in a target register. These can either clear the other fields of the target register or leave them unchanged. This is useful for altering single field values directly, for extracting particular field values based on conditional tests, or for gather and scatter operations in which a long vector may be compressed to optimize execution, then returned to its original state.

Other than its shift-right-and-extract instructions, MAX-1 is completely bereft of combinatorial operations such as interleaves and catenations. It also lacks packs and unpacks. This makes it hard to perform type conversions or to emulate operations by converting data to a supported precision (e.g. using 16-bit additions to implement 8-bit additions).

A full set of loads and stores is included which allow any multiple of a byte to be accessed directly. Although it is intended for constructing an effective address, a load immediate instruction is also available which allows a 21-bit value to be loaded. Immediates are normally loaded using a "load offset" instruction which adds the immediate to the contents of a base register. By using register 0, which always generates 0, the immediate can be loaded. An instruction which stores multiple bytes starting at an unaligned position is also available. This is useful for optimizing the storing of long data vectors.

MAX-1 includes a memory access synchronization instruction. This weakly enforces the ordering of all memory accesses including semaphore touches and cache flushes. MAX-1 also allows many of its instructions to provide cache hints, and includes a set of cache management instructions for flushing cache lines. These allow separate data and instruction caches to be handled separately and also allow a privileged process, or the operating system, to evict lines belonging to user processes.

While MAX-1 is more complete than MVI, it is limited in scope. Because of this, a large amount of emulation would be needed to be implement a full SWAR programming model using MAX-1. Its limitation to 16-bit parallel objects restricts its usefulness for character stream and standard integer processing. However, it's biggest fault is probably the lack of instructions that would support the emulation of operations on data of unsupported precisions. MAX-1 would be a difficult, but not impossible, target for a SWAR architecture.

2.2.3 PA-RISC MAX-2

MAX-2 extends the MAX-1 extension set in two major ways. First, it extends the existing MAX-1 instructions to make use of the 64-bit PA-RISC 2.0 architecture. Second, it adds support for data alignment and rearrangement operations.

One of the limitations of MAX-1 was the lack of a set of simple, generalized shifts. This is resolved in MAX-2 with the addition of shift by immediate instructions which operate on 16-bit partitioned data. These instructions make the emulation of unsupported operations easier to implement using MAX-2 than they would be using MAX-1. They still suffer from the limitation that the index is not variable. This type of operation is difficult to emulate using constant-count shifts, so there is still room for improvement.

A generalized permute by immediate instruction allows arbitrary reordering of the fields of a partitioned register including replications. This instruction addresses one of the problems with MAX-1 by providing a means of converting scalar data into a partitioned form for use in mixed expressions.

Other problems addressed by the permute instruction are type conversion and the emulation operations on unsupported data sizes. Permute allows values to be packed and unpacked, thus making it less costly to convert between sizes and to emulate n-bit operations with 2n-bit operations.

Permute can also be used to perform a large number of operations which resemble communications. For example, a single field value may be replicated such as in a broadcast or each field value may be passed to its neighbor such as in a nearest neighbor communication operation.

Unfortunately, the permute's index vector is an immediate value. Thus, it must be known at compile-time. This limits the usefulness of the permute as a communication operation to fixed patterns. This is not a problem for type conversions and emulation, which are defined at compile-time anyway. Thus, MAX-2's permute is still very useful for implementing a generalized SWAR model.

MAX-2 also has a reasonable set of interleaving "mix" instructions which support 16- and 32- bit field sizes. These are most useful for promoting and demoting data for emulating operations on unsupported field sizes. These can be used to address the data conversion problem. The operation of these instructions is actually covered by the permute. Thus, these instructions are only useful if they provide a performance improvement over using the permute instruction.

MAX-2 also extends MAX-1's functionality by supporting cache prefetching. This is accomplished by using the "ldd" or "ldw" instructions to "load" general register 0, which is actually read-only. The "ldd" instruction indicates a load for read, while "ldw" indicates a load for write.

MAX-2 supports a reasonable range of SWAR operations; however, the supported field sizes for any given operation are often severely limited. Thus, a large amount of emulation would be required to implement a general-purpose model. While the additions beyond MAX-1 are not as useful as they could be, they do address some of the primary problems with MAX-1 and make MAX-2 a usable target for SWAR processing.

2.2.4 MIPS-V

The MIPS-V extensions include arithmetic instructions for modular addition, subtraction, and multiplication. Unary absolute value and negate instructions are also available. These instructions allow basic math operations to be performed on floating-point data.

MIPS-V does not include saturation operations, nor does it include minimum or maximum operations which could be used to emulate saturation operations.

Various forms of multiply-add and multiply-subtract instructions are also included, but these are not particularly useful for a generalized SWAR model. They are most likely to be used as optimizations for special situations.

No divide or reciprocal instructions are included in the MIPS-V extensions. Thus, floating-point division will have to be serialized if it is included in the general-purpose model. MIPS-V also lacks the square root, log, and exponential instructions included in some of the other floating-point extension families.

MIPS-V's rich set of partitioned conditional tests is by far the largest of any of the extension families. These instructions allow tests for multiple combinations of conditions including orderedness and unorderedness. These tests set condition code bits which represent the result of the test on each field. Conditional move instructions merge each field of the source into the result based on the values of these bits.

The "cvt.ps.s" instruction packs two floating-point single values into a 2x32f partitioned value. This allows two non-consecutive 32-bit values to be easily combined into a partitioned register without involving extra masking steps.

The "alnv.ps" instruction extracts either the low or middle 2x32f from the 4x32f concatenation of two 2x32f sources, and is usually used for data alignment. For the purposes of SWAR processing, it is most useful for performing neighbor communications on multi-fragment vectors or aligning unaligned memory accesses.

A set of instructions allow the upper, lower, or a mix of the even and odd fields of the two sources to be interleaved. These also may be used to facilitate certain communication operations.

Auto-aligning instructions for loading and storing the floating point registers are included which are used by both the MIPS-V ISA and MDMX. These load the aligned 64-bit block which contains the given address rather than the 64-bit block starting at the address.

As a floating-point extension supporting IEEE-compliant computing, MIPS-V does fairly well; however, the lack of support for division is disturbing. Support for saturation arithmetic is non-existent; thus, a model which includes saturation math will be difficult to implement on MIPS-V.

2.2.5 MDMX

Data stored in the accumulator is always signed, and operations which target the accumulator are always modular. Instructions which target the accumulator include addition, subtraction, and multiplication.

Data exists as a "bit array" until one of the partitioned operations is applied, at which time the data is converted into 8x8u or 4x16s form. From then on, the SHFL instruction must be used to convert between 8x8u and 4x16s forms, otherwise the data becomes undefined. Conversion from 4x16s to 8x8u requires data to be saturated with MIN or MAX and rearranged via SHFL.

MDMX instructions which target the floating-point (FP) registers are always saturated, and are performed on either 8-bit unsigned or 16-bit signed data. These include instructions for addition, subtraction, maximum, minimum, and multiplication. A 16-bit signed multiply by sign instruction is also available; however division, reciprocal, square root, log, and exponential are not.

A full set of shift instructions is also included, as is a reasonable set of polymorphics, both of which operate on the data in the FP registers.

A sufficient set of conditional tests is included for most SWAR operations. These instructions set the floating-point condition code bits based on the result of the test in each field. The "pick" instructions can then use these bit values to select which of two sources they will copy their field results from.

Another interesting feature of MDMX is that condition codes are used and can be read or written in subsets. Most of the extension families avoid using condition codes, presumably to avoid their "side-effect" status.

Several instructions are included for moving data between the accumulator and the FP registers and for packing the data in the accumulator into the more compact forms used in the FP registers.

A set of instructions which perform a byte-shift-left-and-extract operation on the concatenation of two source registers are included. These are most useful for multi-fragment communication operations such as field shifts or rotates, and for aligning unaligned data accesses.

MDMX includes a solid set of combining operations for use with the FP registers. Interleave upper, lower, and even-with-odd instructions are included, as are several forms of concatenation. These instructions are most useful for promoting and demoting data for emulation.

MDMX also includes instructions for sign-extending 8-bit values to 16-bit values. These instructions can save several when promoting 8-bit data to 16-bit data. Because of the relative completeness of the MDMX 8-bit instruction set, these instructions are not as important to SWAR processing using MDMX as they would be to extension families for which more 8-bit operations must be emulated.

Because MDMX is limited to 8-bit unsigned and 16-bit signed data, a significant amount of emulation would be necessary to implement a general-purpose model which includes 8-bit signed or 16-bit unsigned operations. This is not fatal, and MDMX's versatility and range of operations make it a reasonable target for SWAR operations.

2.2.6 AltiVec

Arithmetic operations consist of modular and saturation addition and subtraction on 8-, 16-, and 32- bit integer data, and saturation addition and subtraction on 32-bit floating-point data. A 32-bit unsigned addition high is also included which can be used to emulate 64-bit unsigned additions. The corresponding subtract high is also included.

A 32-bit signed reduce-add-with-element can be used to quickly perform multi-word reductions. Partial reduce-add instructions, which reduce subsections of a register into a partial result are also included.

Modular multiplications on 8- and 16- bit data multiply either the even or odd fields of the sources, yielding a result with doubled field widths. This allows saturation multiplication to be easily emulated. An interesting set of multiply-add and multiply-subtract instructions are available, but these are somewhat esoteric, and would probably only be used for optimizations.

Maximum and minimum instructions operating on signed and unsigned integers and floats are included, as are a full set of integer averages.

A single-precision floating-point reciprocal approximation instruction can be used to perform floating-point division. Floating-point reciprocal square root, log base two, and exponential approximations are available, but also are likely only to be used for optimization.

Partitioned shifts and rotates include 8-, 16-, and 32- bit logical and arithmetic operations, but 128-bit shifts must be performed in multiple steps by shifting by bytes, then by bits within the bytes. Full-width shifts are often used in emulation, and the lack of these is a potential problem. However, a set of polymorphics sufficient to perform enable masking and emulation is included.

Conditionals include a full set of integer equality and greater than tests and a 32-bit floating-point greater than or equal test as well. These yield a *field mask* which sets all the bits in each field of the result to either '0's or '1's depending upon the result of the test. Such a mask is immediately usable for enable masking in a SWAR environment. This form of result is probably the best single choice possible for partitioned tests.

Pick true and false instructions are also included which can be used to perform trinary operations easily. A 32-bit compare bounds instruction indicates the relationship of two floating-point operands, but is likely to be used only in special case situations.

AltiVec includes field replication for 8-, 16-, and 32- bit fields. This is most useful for converting single-valued data to vector form – an operation which occurs often in SIMD code. This can also be used to optimize the replication of other-sized fields as well. Field selection is via an immediate value, which limits the usefulness of these instructions to non-variable field indexing and internal emulation. Taking the field number from a register would allow variably-indexed fields to be selected for replication; however, this functionality is provided by a generalized permute instruction and would thus be redundant.

Replication of a 5-bit immediate, sign-extended to the field size, is also included. These can be used to load small magnitude constants in one step or larger constants in two for the supported field sizes. They can also be used to load constants into smaller fields in multiple steps.

Instructions for converting data between integer and floating-point type are also included, as is an instruction for rounding floats to an integer floating-point value. These instructions are useful for type conversion and casting.

The “vsldoi” instruction allows a 16-byte sequence to be extracted from the concatenation of two 16-byte values. It is intended for alignment purposes, but can be used for vector shifts or communication operations.

AltiVec’s “vperm” instruction performs a general permutation on two source registers to form a single result, indexed via a third register. This allows it to be used both statically for data layout and type conversions, or dynamically to support variably-indexed element accesses or data communications such as a `router` operation in MPL [107].

AltiVec includes several forms of interleaves and unpacks including saturating forms which can be used for type conversion and to emulate unsupported saturation operations. The "unpack-and-sign-extend" instructions allow signed data to be converted to larger precisions easily. This would be an expensive operation otherwise, and is one which occurs often when emulating saturation operations. The pack and unpack pixel operations are less useful and unlikely to be used by a general-purpose compiler.

AltiVec also includes instructions to load or store aligned vector elements and to load or store 128-bit blocks. These operate without changing the relative position of the element in the enclosing 128-bit block. Unfortunately, the loads are not as useful as they could be. When loading a single element, it needs to be aligned, thus requiring a shift operation. Also, the AltiVec loads leave the surrounding bits undefined, thus requiring a masking operation to clear them. When loading an element into a previously loaded fragment, we would like the surrounding elements to be unchanged, but AltiVec doesn't guarantee this, so again we must perform masking to insert the element properly. AltiVec's element stores are more useful in that they don't clobber surrounding data. This allows element stores without performing masking; however, if the data to be stored is single-valued, alignment is still required.

One problem with AltiVec is that data cannot be moved directly between the vector and general-purpose integer registers. Thus, array indices generated in the vector registers must be moved via memory to the integer registers for use in a load or store instruction.

The load-vector-for-shift instructions load a vector value which can be used as as the index for a permute operation to extract a 16-byte sequence from the concatenation of two 16-byte fragments. This can be used to implement vector shifts and rotates, but is intended for the alignment of unaligned memory accesses.

AltiVec also includes instructions to provide hints about whether data should or should not be prefetched for loads or stores. These define a data stream which can contain up to 256 units of up to 32 bytes each, strided by up to 32768 bytes. This allows hints about data stored in various memory layouts to be easily indicated.

AltiVec is a rather complete set of extensions. As a general rule, support is broad and available for each of the standard data sizes below 64 bits. Support for 64-bit data is, however, lacking.

Overall, AltiVec is a very good target for a general-purpose SWAR model, but the lack of 64-bit operations in a 128-bit environment leaves a large gap. Also, the lack of simple data moves between register sets and the mediocre memory access system make generalized addressing difficult.

2.2.7 VIS

A reasonable set of modular arithmetic instructions is included for 16- and 32- bit operations. Addition and subtraction instructions are included, as is a reduce-add of the absolute differences of 8-bit field values.

A large number of multiply instructions are included, each of which multiplies four 8-bit values with one to four 16-bit signed values. These typically produce a 24-bit intermediate value which is then converted to the format of the final result. Few of these will be generated by a general-purpose compiler except as a special case optimization.

Minimum and maximum operations are not supported by VIS, nor are divide, reciprocal, square root, logarithmic, or exponential instructions. The lack of support for saturation arithmetic or for maximum and minimum operations will make the emulation of saturation operations difficult for anyone attempting to implement them.

A large set of polymorphics is also included in the VIS extensions and can be used to facilitate the emulation of unsupported operations.

Several comparison operations are available which operate on 16- and 32- bit partitioned data. These set a bitmask in an integer register, which can then be used by a masked store instruction.

VIS includes an “faligndata” instruction which performs a “byte shift right and extract” operation. It also includes “fpack32”, “fpack16”, and “fpackfix” instructions which perform “scale, truncate, and clip” or “scale, truncate, clip, and merge” operations and were intended to be used to convert between VIS’ pixel and fixed-point formats. They are the only forms of shift instruction included in VIS. Unfortunately, these forms are not particularly useful for emulating those operations which are not supported by VIS. The lack of simple bitwise shifts severely limits emulation possibilities.

An interleave instruction allows two 4x8u partitioned registers to be merged into a single 8x8u, and can be used for field size promotion, as can an unsigned expand instruction which zero-extends the fields.

VIS includes 8- and 16- bit loads and stores, and block loads and stores which move 64 bytes of data between memory and eight of the 64-bit floating-point registers. Instructions are also included which clear or set all the bits in a register.

Masked store operations in which the fields to be stored are indexed via an integer bitmask are available to limit the effects of a store to a specific set of fields. These can be used to implement SIMD enable masking for high-level conditional code. While the masked store is a good idea, it would be better if it was indexed by a field mask instead of a bit mask. This would allow better integration with SIMD masking code.

Despite Sun’s claim to the contrary, VIS seems to be designed for specific algorithms rather than for a general-purpose model. The selection of esoteric instructions over simple or generalized instructions makes supporting a truly general-purpose model more difficult than is necessary.

2.2.8 MMX

The MMX extension family, originally from Intel Corporation [93, 71], and cloned by Advanced Micro Devices, Incorporated [73], Cyrix Corporation [74], and others such as Rise Technology Company [94], was originally "...designed to enhance performance of advanced media and communication applications" [72] while retaining "full compatibility with existing operating systems and software." [93] An overview of the MMX family is provided in [72], and detailed descriptions of the instructions are available in [95]. A short summary, including cycle counts, is available in [93].

The MMX extensions provide a fairly wide range of support for a high-level parallel programming model; however, they are limited to 8-, 16-, and 32- bit SWAR operations which are not implemented consistently across these field sizes.

MMX operates on data stored in the floating-point (FP) registers. These registers cannot be used for floating-point operations while MMX is in use, and the standard integer instructions cannot be used on the data stored in these registers. In this sense, MMX is less useful than the families which partition their standard integer registers.

The supported arithmetic instructions include a reasonably complete set of modular and saturated addition and subtraction instructions, 16-bit modular multiply and signed multiply high instructions.

A multiply-add instruction is useful for certain algorithms, but is only likely to be used as a special-case optimization by a SWAR-based compiler.

Maximum and minimum instructions, which are useful for emulating saturation operations, are not included in the MMX instruction set. This means that emulation of saturation operations is expensive using only MMX. Divide and reciprocal are excluded, as are square root, log, and exponential instructions.

Shifts on 16-, 32-, and 64- bit fields are included, and are sufficient for most SWAR needs. A solid set of polymorphics is also included. These make it possible to emulate many SWAR operations which are not supported directly by MMX instructions.

Partitioned comparison instructions return a field mask which is immediately usable for enable masking in a SWAR environment.

A full set of interleave upper, interleave lower, and catenate even instructions support data promotion and demotion for the emulation of unsupported SWAR operations.

Memory access instructions include the “movq” and “movd” instructions which are capable of moving data both between floating-point (FP) registers and between these and memory. Also, most MMX instructions allow one of the sources to be in memory, thus eliminating the need for a separate move in certain cases.

Despite its limitations, MMX is one of the more complete families of SWAR extensions. However there are enough gaps that Intel felt the need to address them as part of the SSE extensions (see section 2.2.13).

2.2.9 3DNow!

3DNow! [75] includes 32-bit saturated floating-point addition, subtraction, and multiplication, and a saturating 32-bit floating-point reduce-add-and-pack which will substantially reduce the number of instructions necessary to perform a vector reduction on floating-point data.

Floating-point maximum and minimum instructions are also included, as are instructions to approximate the reciprocal and reciprocal square root of a floating-point element. While the last of these is most likely to be used only in optimizations, the reciprocal can be used to emulate divides which are not directly supported otherwise.

Floating-point comparisons result in field masks as in MMX, which, because they use the same register set, can be used to mask integer or floating-point vectors.

Instructions are included for converting between 32-bit signed integer and floating-point data, which allow type conversion and casting to be performed easily.

A limited set of cache management instructions are also included for prefetching a 32-bit data line and marking it dirty (written to) when useful.

Because 3DNow! is an extension of MMX, MMX's shifts, polymorphics, loads, and stores can all be used with 3DNow!. To make MMX more complete, 3DNow! also includes a 16-bit signed modular multiply high and an 8-bit unsigned average instruction.

In summary, 3DNow! is a good first step toward adding floating-point SWAR capabilities to MMX and improving its coverage. There is still room for improvement which is addressed by the Athlon extensions to 3DNow! (see section 2.2.10).

2.2.10 Enhanced 3DNow! and MMX

A 32-bit floating-point reduce-subtract-and-pack performs a subtraction on the elements of two registers, then packs the results. This is the complementary operation to 3DNow!'s reduce-add-and-pack. Another floating-point instruction performs an addition on one register, a subtraction on another, and then packs these results into the destination. Depending on how a reduce-subtract is defined in the programming model, one or the other of these instructions could be used to implement the operation.

The "psadbw" instruction performs a reduce-add on the differences of two 8x8 integer values to form a 16-bit unsigned result. This can be used to optimize reduction code which can be expensive without such support.

E3DNow! also includes 8-bit unsigned and 16-bit signed integer maximum and minimum instructions. These can also be used to emulate 8-bit signed and 16-bit unsigned maximum and minimum operations and saturation operations using larger data sizes.

The "pmovmskp" instruction is used to generate a bit mask consisting of the sign bits of the 8-bit elements of a partitioned register. This would be more useful if it could be used in direct conjunction with the "maskmovq" instruction which performs a masked store of the bytes with a set sign bit. However, the bitmask forms do not match; thus, pmovmskp is not particularly useful.

Enhanced 3DNow! also includes a 16-bit unsigned multiply high, and 8- and 16-bit unsigned average instructions which fill in missing parts of the MMX integer instruction set.

Instructions to convert between 16-bit signed integers and 32-bit floating-point elements are included to complement the conversions between 32-bit data types included in 3DNow!.

A 16-bit field extraction operation can be used to quickly access vector elements which start on a 16-bit boundary, but is not as useful for others, as it would require as many instructions as a mask and align operation using full-width operations. The corresponding 16-bit insert instruction is also included. These instructions can be used to move data between the integer and MMX register sets, and the insert can also move data from memory into an MMX register.

A 16-bit permute, indexed via an immediate, can be useful for emulation and data promotion, but is not as useful as a permute indexed via another register. An instruction for swapping 32-bit floating-point fields is also included. It can also be used to swap the upper and lower halves of the register when it holds integer data, but its operation can also be performed with a permute, so it is actually redundant.

A cache-bypassing store is included, as is a store synchronization instruction which enforces the order of stores which occur before the synchronization point versus those that occur afterward.

Enhanced 3DNow! fills in many of the gaps in MMX and 3DNow! and includes instructions which will facilitate the implementation of a general-purpose model on the Athlon architecture. With these extensions the Athlon has become a mature SWAR architecture.

2.2.11 3DNow! Professional

AMD introduced the 3DNow! Professional [98] extensions to the Athlon instruction set in order to bring its various multimedia extensions to par with Intel's Streaming SIMD (SSE) extensions (see section 2.2.13). This set of 52 instructions includes those found in SSE which are not found in MMX, 3DNow!, or Enhanced 3DNow!.

2.2.12 Extended MMX

Cyrix's Extended MMX (EMMX) [77] has two purposes. First, it extends the MMX extension set by a few instructions. Second, it adds flexibility by including instructions which target a register whose use is not explicitly indicated in the instruction, but rather implied by the use of its sequentially paired register whose number differs only in the least significant bit. Effectively, these instructions are three register instructions rather than the IA32 standard of two, and allow the instruction to avoid overwriting one of its sources.

16-bit signed saturation addition and subtraction are included, both of which repeat the functionality of an existing instruction, but target an implied register. Similarly, a set of 16-bit signed multiply high instructions allow the result to be stored or accumulated with an implied register.

One addition to MMX is an 8-bit unsigned saturating sum of absolute differences instruction which accumulates with the partitioned value in the implied register. Another addition is a 16-bit signed magnitude instruction in which each element of the result is the element with the larger absolute value of the corresponding elements of the sources. Neither of these is likely to be used as anything but an optimization by a general-purpose compiler.

An 8-bit average is also included, which performs a signed operation for CPUs prior to version 1.3, and unsigned for versions after 1.3.

A set of 8-bit partitioned conditional loads is also included which load each field based on the value of the corresponding field of a test register.

To the best of my knowledge, EMMX has not been implemented on any publicly-available CPU, although according to a preliminary version of the Cyrix CPU Detection Guide [100], the GXm was intended to support EMMX.

The implied register concept is interesting, but it is unlikely that EMMX will be implemented (if it hasn't already been) because it has been overtaken by the more advanced extensions from AMD and Intel.

2.2.13 SSE

Extensions to MMX include an 8-bit unsigned reduce-sum of absolute differences, 8-bit unsigned and 16-bit signed maximum and minimum operations, and an instruction to generate a bit mask of the sign bits of an 8x8 partitioned register. A 16-bit unsigned modular multiply high is also included, as are 8- and 16- bit unsigned averages.

Instructions to insert or extract 16-bit fields into or out of an MMX register are included. These are the equivalent of the E3DNow! instructions of the same name. Similarly, a 16-bit permutation operation is included and suffers the same limitations as its E3DNow! counterpart.

The "pinsrw" instruction can be used to load a selected 16-bit field into an MMX register. The "movntq" instruction can be used to store the contents of an MMX register while minimizing cache pollution, and the "maskmovq" instruction performs an 8-bit masked store based on the sign bits of the register fields. A store synchronization instruction ensures the ordering of stores occurring before the synchronization point versus those that occur afterward.

The floating-point extensions in SSE include partitioned and low element forms of basic modular arithmetic. These include addition, subtraction, multiplication, maximum, and minimum instructions. They also include division and square root, and reciprocal and reciprocal square root approximations.

A basic set of polymorphics is also included which would be useful for emulation, but a lack of shifts tends to limit any such hopes.

Several partitioned and single element forms of conditional operations are included which result in a field mask usable for SIMD enable masking. These test basic conditions and also the orderedness (validity) of floating-point data. Instructions to set the condition codes based on the value of the low element are also included.

32-bit interleaves and concatenates operate on the SSE registers and allow for changes in data layout, type promotion, and vector shifts. A permutation instruction which is indexed via an immediate is also included. It can be used internally by a compiler, but is not as useful as a vector-indexed permute would be.

Instructions to load or store SSE registers either in their entirety or by subsection are available, as are instructions to move data between SSE registers or between SSE and MMX registers. Also included are instructions to convert data between integer and floating-point formats.

2.2.14 SSE2

SSE2 includes instructions for performing basic 64-bit floating-point partitioned and element operations including addition, subtraction, multiplication, division, maximum, minimum, and square root.

The MMX set of polymorphics are also included for use with the SSE registers, as are several forms of comparisons. A relatively large set of type conversions is also supported. New moves, shuffles, and unpacks are included to make handling 64-bit floating-point data easier.

With this set of instructions, the Pentium 4 architecture is a mature SWAR architecture. Emulation of unsupported operations is reasonably well-supported, and numerical analysts are able to use SWAR instructions for 64-bit floating-point computation.

2.3 Other SWAR architectures

This dissertation focuses on commodity microprocessors that are likely to be used as the primary processor in a desktop computing system. However, there are several special-purpose processors with SWAR architectures that I wish to acknowledge at this time.

These architectures range from communications processors to digital signal processors (DSPs), and were not intended for general-purpose computation. For this reason, they were not included in the earlier analysis although a properly designed SWAR model should be applicable.

This section contains a brief survey of these special-purpose SWAR architectures.

MicroUnity MediaProcessor

The MediaProcessor [108] by MicroUnity Systems Engineering, Incorporated is a 128-bit "broadband processor" which was designed to "communicate and process digital video, audio, data, and radio frequency signals at broadband rates...."

The MediaProcessor supports "SIGD" (Single Instruction on Groups of Data) parallelism "over data types of all sizes." This is done by dividing its 128-bit data path into 64-, 32-, 16-, 8-, 4-, 2-, and 1-bit sections. Integer operations can be performed on any of these data sizes. Single- and double-precision floating point operations are also supported. In terms of supported field sizes, this makes the MediaProcessor the most flexible of any of the architectures discussed.

Analog Devices ADSP-2116x SHARC

Analog Devices' ADSP-2116x Super Harvard ARChitecture (SHARC) [109, 110] family of processors are 32-bit system-on-a-chip digital signal processors used primarily for embedded applications.

These processors have two processing elements which can be used in SIMD mode. Each of these consists of an ALU, a shifter, and a multiplier, and operates on its own set of registers. When operated in SIMD mode, the second processor is driven by the same instruction stream as the first, otherwise it is idle.

Data can be operated on in 16-bit floating-point format, 32-bit fixed or floating-point format, or 40-bit extended floating-point format. A wide range of relatively powerful instructions are supported as is saturation arithmetic.

Analog Devices ADSP-TS101S TigerSHARC

Analog Devices ADSP-TS101S TigerSHARC DSP [111] is the first of a new line of embedded processors derived from the SHARC family. This new family is intended for use in telecommunications systems and multiprocessor signal-processing applications.

The TigerSHARC's computational blocks have two SIMD-driven 64-bit processing elements similar to those of the SHARC family. The processor can read and execute up to four instructions at a time in a VLIW manner using a 128-bit memory bus. Supported data types include 8-, 16-, and 32-bit fixed-point and floating-point formats and an extended 40-bit floating-point format.

Equator Technologies MAP-CA

Equator Technologies, Incorporated's MAP-CA Broadband Signal Processor [112] is a VLIW architecture with processing units which can operate in a SWAR manner on 8-, 16-, 32-, and 64-bit data objects.

The MAP-CA is intended to support "broadband multimedia applications" as an embedded system in "infrastructure and end-point products." These include products such as set-top systems, video surveillance systems, and copiers. Another important application is real-time software-based data compression and decompression.

3DSP UniPHY

3DSP Corporation's UniPHY (Universal Physical Layer Signal Processor) [113] is an embedded DSP with SWAR-like SIMD operation intended for broadband networking and signal processing. The UniPHY processor has a set of twelve SIMD execution units which operate on 8-, 16-, and 32-bit data objects. These are connected to a 32-word, 32-bit register file and can be executed in parallel using a set of "expansion instructions".

Philips TriMedia CPU64

Philips Research's TriMedia CPU64 processor [114] is intended for use in applications supporting connectivity between consumer electronic devices such as video recorders and personal computing systems. The CPU64 is a VLIW processor which supports SWAR-like processing in each of its function units. These operations can be performed on 8-, 16-, and 32-bit data objects in a 64-bit data space.

Texas Instruments TMS320c8x Family

Texas Instruments's TMS320 [115] family of DSPs are MIMD processors designed for video and image processing as well as telecommunications. These processors have between two and four 32-bit parallel processing elements, each of which can perform SWAR "multiple-byte arithmetic" on 8- or 16-bit data.

Texas Instruments MVP

Texas Instruments' Multimedia Video Processor (MVP) [116] is a digital signal and graphics processor based on the TMS320 and TMS340 processor families. It was intended to support applications such as image generation and processing, data compression for network transmission, and integrated multimedia-based computing environments.

The MVP consists of between one and eight processing elements which can operate in MIMD fashion. Each of these is a 32-bit processor capable of performing arithmetic SWAR operations on 8- and 16-bit data. These were intended to support digital signal, pixel, integer, and fixed-point processing.

3. DEFINITION OF A GENERAL-PURPOSE SWAR PROGRAMMING MODEL

Having studied and rejected previously-defined programming models, we turn to the task of developing a new public-domain, high-level model to allow general-purpose programmers to more fully exploit the data parallelism of their applications when targeting current COTS SWAR processors.

A well-designed model should be familiar, yet should more closely reflect the capabilities of current SWAR architectures than do current programming models. It should expand upon these capabilities when this can be done reasonably while promoting code portability between these and other architectures. It should also avoid the imposition of arbitrary limits which would preclude its future application. Such a model should remain viable beyond the lifetimes of current SWAR architectures.

The most salient aspect of these architectures is their vector SIMD nature. This has several implications for the design of a programming model including the expression of data parallelism and the execution of multiple control paths. A large number of programming languages have been developed in the past for use with SIMD and vector systems. The study of these languages presented in appendix A was undertaken to determine how these issues were addressed in these earlier languages. We will use and build upon these ideas during the development of the new model.

To be viable, this new programming model must allow the programmer to achieve his or her goals efficiently. It should allow the programmer to express data parallelism in a manner which is natural. It should also allow the programmer to use familiar programming methods which have been logically extended for SIMD-style processing. Thus, this model should be based on older, more established models, but must be consistent with the operation of current SWAR architectures.

SWAR processors are not as versatile as traditional SIMD array processors. Because of this, the programming models developed for these earlier systems promote the use of features and capabilities which are not available on current SWAR systems. Care must be taken to limit the new model to those facilities which have a SWAR counterpart and to avoid those which do not. The study of previous architectures presented in appendix A was undertaken to determine the similarities and differences between current SWAR systems and earlier vector and SIMD architectures.

One of the purposes of a general-purpose programming model is to provide a means for expressing the use of commonly available functionality. If the model does not provide the expressiveness needed by programmers, then they will be forced to use a different model. Toward this end, the model should incorporate and allow the use of features which are common to a majority of its intended targets.

While the model should allow expressiveness, it should not incorporate esoteric operations which cannot be easily constructed of more common ones. To be portable, every part of the model must be implemented for every target. Any operation included as part of the model will have to be emulated on all targets that do not support it as a hardware operation. Highly specialized operations are likely to require emulation on multiple targets and will be correspondingly difficult to port. Thus, they should be avoided.

Having said that, the model should not be limited to the capabilities of the least powerful architecture. It must be complete enough to allow a programmer to describe the algorithms to be employed, and should be self-consistent so that a programmer may have a reasonable expectation that its functionality is not arbitrarily limited. These properties should hold even if the support provided by some target architecture is lacking.

Thus, in defining this new model, a balance must be struck between promoting code portability by rejecting esoteric capabilities and providing functionality that is reasonably complete and exploitative of the advanced capabilities of various targets.

To address these issues, and as a first step in developing a new SWAR model, I analyzed various multimedia extensions. The purpose was to find the range of support for SWAR processing defined by each extension, to identify commonly supported operations, and to determine which advanced features may be useful in the implementation of the final model. This analysis was presented in the previous chapter and now provides a basis for the design of a new SWAR programming model.

3.1 Relationship to Previous Architectures

Multimedia extensions perform parallel operations on identically-typed data stored in a single processor register. Each instruction causes an identical operation to be simultaneously applied to each piece of data in the register. Thus, this new class of architecture is a limited form of SIMD in which data parallel computation is implemented within a single processor. We refer to this class of architecture as "SIMD Within A Register" (SWAR) to highlight the fact that SIMD-like processing is performed on sets of data stored in individual processor registers.

The parallel data exploited by these extensions is stored in fields which are laid-out linearly across individual CPU registers. Generally, no provision is made for arranging these fields in other geometries. Thus, the natural layout for data on these systems is one-dimensional vectors rather than multi-dimensional arrays. Because the instructions performed by these processors treat their registers as linear arrays, they are vector processors. Thus, the most natural model for an architecture which incorporates multimedia extensions is a vector parallel SIMD model.

This is in contrast to SIMD array processors such as Westinghouse's SOLOMON prototypes [117, 118], the University of Illinois' ILLIAC IV [119], the ICL DAP [120, 121], and the Goodyear MPP [122, 123]. These systems were designed to operate on multi-dimensional arrays for applications such as image processing and the simulation of processes in physical environments. They had interconnection networks that could perform regular communications operations in multiple directions. These allowed them to take the form of the data objects on which they operated. Thus, they were well-suited to an array processing model while SWAR architectures are not.

Later SIMD array processors had more advanced interconnection networks. The IBM GF11 [124, 125, 126] had a set of 576 pipelined PEs that were fully connected via a non-blocking Beneš network [127]. The Thinking Machines' CM-1 [128, 129, 130] and CM-2 [131, 130] had multiple networks including a packet-switched hypercube router network which allowed any two PEs to communicate directly. The MasPar MP-1 [132, 133, 134] and MP-2, which were developed slightly later, had similar networks and also an "X-net" which could perform a large number of regular communications patterns. These networks are beyond the capabilities of all but the most advanced of the current SWAR architectures.

SWAR architectures are a cross between purely pipelined SIMD processors such as the CRAY-1 [135] and SIMD parallel vector processors such as the CDC Cyber 205 [136, 126] or NEC SX-2 [137, 138]. The modern microprocessors upon which SWAR architectures are based are pipelined processors which overlap multiple instructions in stages. SWAR instructions are also overlapped in this manner; however, they perform in SIMD mode when executed. Thus, SWAR processors are similar to pipelined vector processors which have multiple identical functional units.

Several historic vector processors fall into this last category. For example, the TI-ASC [139] could support up to four identical vector pipelines which could be operated in a SIMD manner [140]. The NEC SX-2, Fujitsu VP200 [141] and VP2600 [142, 141], and Hitachi S810/20 [141] and S820/80 [143] are all examples of this category of architecture.

The CDC STAR-100 [144, 125] was also closely related. It was a pipelined vector processor with SWAR capabilities. Each of its two vector pipelines could process one 64-bit operation or two simultaneous 32-bit operations. Special logic inserted between the two halves of the 64-bit datapath broke the carry chains between them. This effectively separated the datapath into two independent parts which performed identical operations. This method of partitioning the processor is essentially the same method used in modern SWAR architectures.

Special-purpose single-IC SIMD processors such as the NCR GAPP [145, 146, 147] and BLITZEN from the Microelectronics Research Center of North Carolina [148] are also related to SWAR architectures. They are also single-chip processors, but are more advanced in the sense that they are array processors. Future COTS SWAR processors are likely to be single-chip array processors such as these with bit-slice or word-slice features similar to those of the MPP [122, 123] or the ILLIAC IV [119].

3.2 Relationship to Previous Programming Models

As a SIMD model, we would expect the new model to be similar to the programming models developed for previous SIMD architectures. Thus, if possible, concepts traditionally associated with SIMD processing should be incorporated into the SWAR model. In this section, some of the various programming models and languages used for parallel processing are discussed.

Most early programming languages such as FORTRAN and Algol were based on scalar programming models. Operations in these languages applied to single-valued objects and not to multi-valued objects such as vectors and arrays. As SIMD architectures were developed, parallel languages were derived from these scalar languages.

Later programming models treated vectors and arrays as single entities rather than a collection of scalar data. These models more closely captured the essence of vector and array processing. Other models were also developed which treated more complex, irregular collections of data as single entities. Each of these types of models will be discussed in turn.

Scalar Models

So-called "vectorizing" compilers analyze code written in a scalar source language to find operations and functions which can be parallelized. These are then translated into vector- or array-based parallel code for the target architecture. Thus, the programming model is a scalar one, but the target architecture is based on a vector or array model.

There have been a number of vectorizing compilers developed over the years for use with standard scalar languages such as Fortran and, more recently, C. The NX Fortran compiler [149] was a fully vectorizing compiler for Fortran 66. Other vectorizing compilers for scalar languages include Cray's CFT, Fujitsu's Fortran 77, IBM's VS Fortran, Alliant FX/8 Fortran, NEC SX Fortran, and Intel's C/C++.

With the goal of developing a model that closely matches the intended target architectures in mind, we will reject scalar programming models as being inconsistent with current commodity SWAR architectures, which are vector-based.

Modified Scalar Models

Some programming languages use scalar models which have been modified to operate on all elements of a vector or array simultaneously. Operations are denoted as indexed vector or array element operations which are similar or identical to scalar elemental operations. In some cases, special forms of indexing are used to indicate that the operation should be applied concurrently to multiple elements. In other cases, high-level language constructs are used to select indices and embody statements that operate on the elements indexed.

Generally, these mechanisms denote what should be first-class vector or array operations as a set of scalar operations. Thus, they allow parallelism while maintaining a scalar model. Some of these mechanisms allow flexible access to subsets of an object's elements and can be useful even in a vector or array model. Because of this, we will discuss a number of them briefly.

Wildcard Indexing

ILLIAC IV FORTRAN [150] used *wildcard indexing* to indicate that a particular dimension of an array was to take on all possible values. This was denoted using a syntax that matched a scalar array element access, but with an asterisk as the index for the parallelized dimension.

CFD [151, 152] used an extended form of wildcard indexing which allowed rotations to be defined by adding or subtracting an offset from the wildcard.

Wildcard indexing presents vector and array operations as a collection of scalar operations over the matching elements of the parallelized object. This should be unnecessary in a vector model, as this would represent an operation applied to the entire vector. That is, it would indicate a first-class vector operation which should be expressed more succinctly in a vector model.

Control Vector Indexing

Parallel conditionals in ILLIAC IV FORTRAN were handled using *control vector indexing*. This allows a vector to be used as an index whose values indicate whether or not an operation should be applied to the corresponding element of the indexed multi-valued object.

Control vectors must be generated by some conditional means, so their functionality can be implicitly performed by conditional language constructs such as a parallelized `if` statement. Thus, they should be unnecessary.

Index Sets

Index sets are used to specify which elements within a parallel object that an operation would be applied to. They are essentially lists of indices and/or ranges of indices which should be included. Thus, they define a subspace of the parallel object to which they are applied.

Actus [153] employed a form of index sets which were treated as first-class objects that could be operated on to form more complex sets of indices.

Index sets are useful as notational devices, but are probably unnecessary as first-class objects because their functionality, like that of control vectors, can be performed by conditional language constructs and parallel variables.

Vector Indexing

Some languages allow vector objects to be used as indices to vector or array accesses. These have a notation similar to scalar array element accesses, but used a vector name as the index. This is sometimes referred to as *vector subscripting.* Actus was also one of the first languages to allow vector subscripting.

Vector indexing is a useful concept, but requires a level of data movement unavailable on most SWAR architectures. They can be used to represent a permutation of the data in a parallel object, an operation that is only well-supported on highly-connected architectures such as the Connection Machine or the MP-1.

Extent of Parallelism

Actus introduced the concept of an *extent of parallelism.* This was the parallelism width applied to a vector or array object along a particular axis. It was intended to be independent of the size of the target architecture.

The maximum extent of parallelism and the axis along which it could be applied were specified when an object was declared. When the object was accessed in an expression, the extent of parallelism used for that access was specified using an index notation. This could be smaller than the declared maximum, to allow tailoring to the target architecture, but had to run along the same axis.

To simplify the expression of a series of statements which use the same extent of parallelism, Actus introduced the `within` construct. This specified an default extent of parallelism to be used by all statements within its body, and was similar to Pascal's `with` construct. The default was indicated within the body by a sharp symbol (#) used as an index.

The extent of parallelism, like index sets, is most useful as a notational device which allows a subspace of a parallel object to be specified for operation.

Triplet Notation

Triplets were a concise notation that defined the first and last elements of a vector or array to be accessed in parallel along a particular dimension and, optionally, the stride between them. This allowed parallel operations on regularly-spaced scalar elements to be specified without the use of looping constructs.

Triplets are most useful for non-unit stride accesses. Because current SWAR architectures are not particularly well-suited to this type of access, triplets would tend to promote inefficient use of the target architecture.

According to [154], triplets were introduced in VECTRAN [155] and BSP Fortran [156]. Various forms of triplet notation have been used in later languages, including Fortran 90 [157] and High Performance Fortran (HPF) [158]

The `DO FOR ALL` Construct

IVTRAN [159] introduced a `DO FOR ALL` construct which was used to indicate that certain array element assignments and intrinsic function calls within its body could be executed in parallel. The elements operated on could be limited to a subarray using an *index set* notation which allowed a subrange of indices along each axis to be selected.

`DO FOR ALL` should be unnecessary in first-class vector and array models because it simply denotes the parallel application of an operation or function to a subset of the object's elements chosen *a priori* or via a conditional test. This can be done using parallelized standard condition constructs.

The `where/otherwise` Construct

VECTRAN introduced a conditional `where/otherwise` construct which is similar to a parallelized `if/else`. This construct applied implicit enable masking to array element assignments in its scalar bodies. The `where` section was enabled only for those elements which passed the test, while the `otherwise` section applied the opposite enable mask within its body.

This construct appears in later languages in various forms. For example, Fortran 90's `WHERE` and `ELSEWHERE` statements have bodies which consist of first-class array assignments that are conformable to, and masked by, the construct's test expression. In this form, the `where` construct is useful for vector- and array-based models.

The `identify` Statement

VECTRAN also had an `identify` statement which allowed irregularly-shaped subarrays to be aliased (i.e. named) for later parallel operations. This separated the selection of a subset of elements from the use of this selection in parallel assignments. This is essentially equivalent to storing the result of a parallel conditional test in a variable for later use, and is thus unnecessary in a language which supports this functionality.

The `FORALL` Construct

CM Fortran [160, 131] included a `FORALL` statement [161] which was essentially equivalent to a `FOR` loop in which the iterations were known to be parallelizable. To ensure this, the body of a `FORALL` was restricted to single array element or section assignment.

The `FORALL` was equivalent to VECTRAN's `identify`, except that it combined the separate aliasing and assignment statements into a single construct. It also allowed subspace selection by value or position. `FORALL` is notationally convenient, but should be unnecessary if parallelized standard conditional constructs are available.

Parallelized Conditional Constructs

Many scalar-based languages provide parallelized versions of their standard conditional constructs. For example, Actus had parallel `if`, `while`, `for`, and `case` constructs which embodied scalar element access statements.

These constructs require that the conditional test be evaluated for each PE, element, or individual index, and that the correct set of statements be executed for each one, depending on whether or not it passed the test.

Because they are more general than many of the subspace selection mechanisms discussed above, these constructs can be used to emulate or replace them. This suggests that it may be a better strategy to use parallelized conditional constructs rather than to use less general selection mechanisms. This translates to vector and array models as well as scalar models, and is a common method for handling conditional execution in each case.

Array Models

The most commonly used non-scalar models in parallel processing are multidimensional array models. Some of these treat arrays as first-class objects, meaning that they can be operated on as a single aggregate object rather than as a set of scalar elements via looping or parallelizing constructs. Other models treat arrays as pseudo-first-class objects via modified intrinsic functions or operator overloading.

Current SWAR architectures are vector parallel processors and are thus not particularly good at array processing. In particular, they lack the memory access and communications mechanisms necessary to carry out array processing efficiently. Thus, an array model is not the best choice for supporting these architectures. However, it is instructive to look at languages based on array models to see how they have incorporated parallel operations on aggregate data.

First-class Arrays

Truly first-class operation on arrays are written using array names as operands without the need for indexing or special language constructs. Operations described in this manner are applied to the aggregate object as a single transaction and may therefore be parallelized. Normally, unary operations are applied to each element of the operand while binary operations are applied in an element-wise manner to a pair of conformable operands.

A fair number of languages are based on first-class array models. Several of these which are discussed below.

The first significant programming language to incorporate vectors and arrays as first-class objects was APL [162]. It had a mathematically-oriented notation in which algorithms where essentially descriptions of expressions to be evaluated. APL allowed vector and array operations to be described in a high-level, portable manner. It introduced a large number of intrinsic functions which could be performed on scalars, vectors, and arrays, including reductions and scans. Many of its features have been absorbed by later parallel languages.

GLYPNIR [163] was an early SIMD language for programming the ILLIAC IV. It was based on ALGOL 60 [164, 165], an early scalar language whose primary contributions were block structure, dynamically-allocated variables, and recursion. GLYPNIR introduced separate `CU` and `PE` data types. These were essentially storage class specifiers which indicated where the data should be stored, and thus exposed the separate control and parallel units of the ILLIAC IV.

GLYPNIR's `PE` variables were first-class parallel objects. They were stored and operated on in parallel across the entire PE array. These variables represented a *sword* of data residing at the same address on each of the PEs. PE variables could also be used to index a vector of swords. This allowed a *slice* of data residing at various address on the set of PEs to be accessed.

GLYPNIR also introduced parallelized conditional constructs including `IF`, `ELSE`, `FOR`, `DO`, `WHILE`, and `FOR ALL`. These used implicit PE masking to limit operations to those PEs for which the condition held.

NX Fortran [149] was a version of Fortran 66 with first-class vectors and arrays. It allowed array assignments if the shape of the data to be assigned conformed to the shape of the destination object. It also allowed promotion of scalars to multi-valued objects via replication. The NX Fortran compiler was a fully vectorizing compiler for Fortran 66, and could thus parallelize scalar code as well as array code.

C* was a parallel language for the Thinking Machine's Connection Machines. It evolved through three models of parallelism, each of which was based on multi-dimensional arrays.

The original version of C* [166] had `mono` and `poly` storage classes which were similar to GLYPNIR's `CU` and `PE` data types. A `poly` object was one which was allocated on each of the PEs in the Connection Machine's three-dimensional PE array. Operations performed on these objects were parallelized.

C* allowed the standard C assignment operators to be used as unary reductions. These operated under the "as if serial" rule, which required that their results be equivalent to executing the elemental operations in some undetermined order.

A subset of PEs could be selected for processing based on the concept of the *active set* of PEs. All parallel operations were performed on the current active set of PEs. This set could be explicitly selected using a *selection statement* or implicitly set via conditional constructs.

The format of the selection statement was `[`*`selector`*`].`*`statement`*. The selector could be a `processor` variable, an array of `processor`s, an indexed value representing a consecutive series of `processor`s, or a list of any of the above. This allowed any subset of processors to be chosen at any time to execute a statement, and thus provided a great deal of flexibility.

An active set could also be selected using standard C conditional constructs. The `if`, `else`, and `while` constructs performed their tests on the current active set, and reduced that set during the execution of their bodies by eliminating the PEs for which the condition did not hold. These constructs could be nested and operated under the "rule of local support". This required that the body was executed only if the condition held for at least one active PE.

C* also introduced the notion of a local processor and provided for inter-processor communication. The `this` keyword represented a pointer to data stored on the local processor. It could be indexed to indicate a different processor in a linear ordering of the PEs. For example, `this[`i`]`$\rightarrow x$ represented the variable x on the PE i steps from the local PE. This provided an explicit means of linear communication between PEs which allowed the local PE to access data on others.

The second version of C*, described in [167] and [168], was based on a C++ class-like construct called a *domain*. A domain defined both a data structure and a set of functions which could operate on it. An array of domain instances represented a first-class parallel. Execution of a member function caused parallel execution over the instances of the domain.

Choosing a set of active PEs was now done by executing one of the member functions of a given domain. Syntactically, this was similar to the selection operator in the original C*, except that the selector was now a domain name and the statement applied was now a member function. This function was executed by a PE if, and only if, it contained an instance of the domain.

The third version of C* [169, 131] was developed based on first-class *shapes*. Shapes are n-dimensional arrays of various sizes. They could be independently described and associated with objects as necessary. A default current shape could be set using a `with` statement. In general, objects had to be of the current shape in order to be operated on in parallel.

Conditional selection was defined in terms of an active set of data positions in the current shape. This was set by the language's conditional constructs. A VECTRAN-like `where` statement limited the active position set to those for which a conditional test held. A related `else` clause could be used to limit the active position set to those for which the condition failed. C* also provided an `everywhere` statement to allow all positions to be made active during a single statement.

Other changes included the replacement of `this` with `pcoord` which indicated the local PE's index along a given axis in the current shape, the concept of "left indexing" which allowed assignment to objects residing on other (non-local) PEs, and the addition of a Boolean data type.

Fortran 90 [157] allows first-class arrays which can be operated on in an element-wise fashion. It also allows mixed expressions on conformable objects, and treats scalars as being able to assume any shape. It incorporates many of the parallelism mechanisms discussed above such as triplet notation and `WHERE` constructs.

MPL [107, 170, 171], the MasPar Programming Language, was another SIMD variant of C which treated arrays as first-class objects. It had a `plural` type modifier which indicated that an object was multi-valued with its elements spread across the MasPar architecture's three-dimensional PE array. Operations on these objects were parallelized.

MPL supported inter-PE communication in a manner which exposed the target's architecture. This was done using a set of three new constructs: `proc`, `router`, and `xnet`. These allowed the programmer to specify an expression to be evaluated on another PE with the results communicated over one of the target's interconnection networks.

Pseudo-First-Class Arrays

Languages which do not have first-class arrays may handle them in a manner which hides this fact and allows them to appear to be first-class objects. For example, arrays can be treated as first-class objects if they are manipulated using functions rather than operators. This allows the array to be passed to, and returned from, functions as a single object and appear as a single entity in expressions which call these functions.

Some languages have *intrinsic functions* which are a required part of the language. Parallel languages are sometimes formed by using modified intrinsic functions to extend scalar languages for parallel operation. These functions perform element-wise or reductive operations on vector or array objects without requiring the definition of new language operators or the modification of existing ones. This makes it possible to treat non-first class vector and array objects as first-class objects.

Several vector and array languages have used this method of parallelization. NX Fortran provided intrinsics for generating first-class vectors and arrays. Vector LRLTRAN [172] included the reduction intrinsics `Q8SUM` and `Q8PROD` and the selection intrinsics `Q8MASK` and `Q8MERGE`. Fortran 90 added the `MAXVAL`, `MINVAL`, and `COUNT` reduction intrinsics, and CM Fortran added the `DIAGONAL` and `REPLICATE` intrinsics for array formation.

Some languages allow their intrinsic functions to be overloaded with user-defined functions. As with operator overloading, this can be used to hide parallelization performed by the compiler, and thus give the appearance of parallel operation on first-class objects. For example, C* allowed function overloading based on the shape of a function's arguments.

Another common method of providing pseudo-first-class operation is to allow *operator overloading*. When an overloaded operator is used in an expression, a user-defined function is performed on the operands. As with a modified intrinsic, this function may hide parallelizing scalar constructs or scalar operations which can be parallelized by the compiler. This gives the appearance that the language supports first-class parallel operation without it actually doing so. Fortran 90 is one language which allows a limited amount of both operator and intrinsic function overloading.

Vector Models

Single-dimensional, non-scalar vector models are less commonly used in parallel processing than are multi-dimensional array models. This is because most parallel architectures are based on two- or three-dimensional arrays of processors and thus are better served by multi-dimensional array models.

Vector architectures are less common, and are typically programmed via the vectorization of scalar code or the emulation of array code. That is, they are usually programmed using a scalar or array model. However, true vector models are more consistent with the operation of current commodity SWAR architectures than are scalar or array models. For this reason, the SWAR model described in this thesis is a vector model.

The number of pure, first-class vector languages is significantly smaller than the number of array languages. Below, a few vector languages which have some interesting features are briefly discussed.

Vector LRLTRAN [172, 154] was a language which supported first-class vectors of `REAL`, `INTEGER`, or `BIT` data. It allowed vectors to be used in mixed expressions with extension performed as needed to make vector operands of differing lengths match. This was done by appending elements of the identity value for the given operation to the shorter vector. On assignment, scalars were replicated to match the shape of the destination object.

Vector LRLTRAN also allowed vectors to be passed to, or returned from, functions. This was done using vector *descriptors*, which were used to hold the address and length of vectors. These were visible objects which could be modified during execution, and thus allowed vectors to be dynamically reshaped under user control.

Vector LRLTRAN had several methods for selecting vector elements to be operated on. First, it had a flexible indexing system in which vector expressions could be used and ranges of indices included or excluded from the set. Alternatively, it allowed `BIT` vectors to be used as control vectors. It also allowed subvectors to be aliased using *dynamic equivalencing* then used as first-class objects in a manner similar to VECTRAN's `identify` statement. A set of intrinsic functions were included to perform reductions and selection on vector objects.

C[ ] (C brackets) [173] is a vector extension of ANSI C. Vectors are first-class objects with a declarable fixed stride between elements in memory. Higher-degree objects can be declared, but are treated as vectors of vectors. As with Vector LRLTRAN, vectors can be operated on, passed to functions, and used as return values.

Pointer arithmetic has a consistent interpretation in C[ ], with element and subarray accesses taking the declared stride into account. The standard C operators were parallelized. The C* maximum and minimum operators are also available, as are new operators for population count (`?`), leading zero count (`%`), and word reversal (`@`). Unary reduction operators are also available, and are denoted by enclosing the corresponding C operator in a bracket pair. For example, reductive addition is denoted by the operator `[+]`.

C[ ] allows vectors of bit fields to be assigned values via a gather operation on an integer vector of fixed stride. However, the language is primarily intended to support data of standard precisions, and does not treat bit fields as first-class objects.

AJL (Anar Jhaveri's Language) [174] was a vector calculator language which provided basic arithmetic operations and intrinsic trigonometric functions. These could operate on both scalar (`mono`) and vector (`poly`) objects in a first-class manner. However, it was not intended to be a general-purpose programming language.

Predefined constants were available including `pi`, `e`, and the number of elements in a vector (`#`). AJL also included vector assignment from a list of elemental values, generation of linearly ranging vectors, and vector shifts, shuffles, and inverse shuffles. Operations were limited to vectors of equal lengths, and only standard precision elements were supported.

Other Models

NESL [175] is a language in which parallel data is described as recursive *sequences*. This allows complex, irregular, nested data structures to be described and operated on. Operations performed on a sequence can be performed in parallel across each of its elements or across a subset determined by a qualifying condition.

The following example from [175] shows the syntax of a typical NESL expression:

```
{negate(a): a in [3, -4, -9, 5] | a < 4}
```

This expression applies the built-in function `negate()` to each element of the sequence `[3, -4, -9, 5]` which has a value less than 4.

NESL is based on VCODE [176], a stack-based vector language which allows *segmented vectors*. *Segment descriptors* are used to define the number of elements in each segment of a vector. Most vector operations are applied to their vector operands in a segment-wise fashion and element-wise within each segment. Reductions are applied to each segment individually.

VCODE is, in turn, based on CVL [177], a low-level vector library for the C language. CVL provides a large number of vector operations on segmented or unsegmented vectors of type `int`, `double`, or `cvl bool` (which may take any useful form such as `char`s or bits. Vectors are passed to functions via *handles*, which indicate the position and layout of the vector in a dedicated vector storage area.

The sequence model is probably too irregular to be a good match for current SWAR architectures. It is also dissimilar to the majority of languages used for high-performance computing.

An unnamed fine-grained, parallel version of C developed at NASA's Goddard Space Flight Center [178] was intended to be applicable to targets of various shapes, including serial, vector, and array processors. Thus, the model took on the shape of the target architecture. To support bit-slice targets, all variables could be assigned a bit size which the compiler would use as a minimum required precision.

This language had a `parallel` storage class that represented data spread across the target's PEs. `Parallel` objects were first-class and could be used in expressions involving standard C operations which were parallelized in an element-wise fashion. The C assignment operators were modified to work with parallel objects, performing element-wise assignment or reductive assignment as necessary.

Interprocessor communication was implemented via arithmetic on pointers to `parallel` objects and treated the target's PEs as a ring. By adding an offset, n, to such a pointer, the element on the PE n steps away along the ring could be accessed.

This language was only partially implemented, and only for the serial Apple Macintosh II. It appears to have been abandoned or neglected afterward, as I have been unable to find any other references to it.

3.3 The General-Purpose SWAR Processing Model

The goal of this research was to develop a general-purpose programming model for a class of architecture currently represented by the extension sets studied in the previous chapter. Ultimately, a programming model is an abstraction which provides the programmer with a more suitable or portable target than the actual target architecture or architectures. Thus, defining a programming model is equivalent to choosing the abstraction that is provided to the programmer.

In this chapter, I develop a new general-purpose SWAR programming model in a general sense. That is, we will try to delineate what should be part of the model and what should be excluded while leaving implementation issues, such as how a particular operation is described, for the next chapter. There, I will discuss some implementations of this model. The overall purpose is to provide a consistent, portable, generalized abstraction for this class of architecture.

3.3.1 Classification

In trying to develop a new programming model, one must first decide if it will be imperative, functional, or logical. That is, will algorithms be described as a series of assignments to storage locations, as functions which can be treated as first-class objects, or as a set of logical rules from which conclusions can be determined? This question must be answered before one can progress to the details of language design.

Traditionally, the majority of languages used for parallel processing have been imperative languages which operate via side-effect. That is, they allow for the assignment of values to variables. This is directly related to the actual storage of data in the sense that a compiler assigns a value to a particular variable by storing it in the corresponding storage location. Because most programmers are familiar with this form of programming, and because it is well-established, the model which is developed in this thesis will be an imperative programming model.

3.3.2 Data Representation

How data is represented in a model determines how the programmer can use it to solve his or her current task. It is especially important to carefully choose how parallel data objects will be represented because processing this type of data is the primary goal for the new model. The type of data allowed in the model is also important. A model which is limited to a single data type, for example 8-bit integers, will probably not be useful for most programmers. Thus, the allowed types and precisions must be chosen thoughtfully.

Parallelizable Objects

As a form of SIMD architecture, SWAR architectures exploit data parallelism by applying an identical instruction to multiple streams of data simultaneously. This is sometimes modeled by SIMD languages as an operation on some form of multi-valued data object. For example, we could describe such an action as an operation on a single-dimensional vector of data. This would be a natural choice for a SWAR programming model, but is not the only possible choice.

Despite the fact that SWAR architectures are vector parallel, there are several reasons why we might want to consider an array model rather than a vector model. First, many of the large-scale problems faced by the scientific community require the modeling of physical processes in the three-dimensional real world. Second, vector processing is really just a subset of array processing in which all arrays are one-dimensional. Third, the set of operations performed on mathematical vectors are similar to the set of operations performed on arrays. Finally, an array model would not have to be expanded to incorporate arrays once array-based SWAR architectures become commonplace. From these arguments it seems clear that it would be better to develop an array-based model.

While it is true that many applications are array-based, there are also some that are vector-based. More importantly, given that we cannot know what applications will be developed using this new model, it is best to develop one which matches the intended hardware targets as closely as possible. In this case, a vector model would fit current SWAR architectures better than an array or scalar model.

Another problem is that a strong model tends to encourage the programmer to use its most powerful features. The more these features differ from the actual hardware, the more difficult they are to implement. Hence, they are less portable and often implemented incorrectly or inefficiently when they are ported.

Given the limitations of current SWAR architectures, it would probably make more sense to develop an array-based SWAR processing model if and when SWAR array processors become commonplace. Single-chip array architectures such as the NCR GAPP and three-dimensional architectures based on three-dimensional chip layouts [179] should come to dominate at some future time. In the meantime, it is probably wiser to develop a model which relates more closely to the current batch of vector-based, commodity SWAR processors. Thus, the model defined here is a vector-based model.

Vector Length

Once we have decided on vectors as the primary parallelizable data object, we must now decide what a vector is comprised of. The first issue is vector length. For any given data precision on any given architecture, there is a natural number of elements that can fit into a single register. For example, MMX registers are 64-bits long, and thus can accommodate eight 8-bit vector elements or four 16-bit elements. We refer to the set of data in a register as a *vector fragment*, and the natural length of this fragment as the *fragment* length. Some programming models codify this fragment length as the vector length. For example, it is used as the length of a `vec_*` in AltiVec.org's version of GCC for AltiVec-based processors [180].

While this may seem to be a reasonable thing to do, there are two problems with this approach. First, real-world data rarely fits this natural machine width. Second, it incorporates the specifics of the current architecture in the model, thus limiting the model's usefulness to the current architecture. We wish to avoid both of these problems, and can best do so by making the vector length variable. Thus, the general-purpose SWAR programming model allows all finite, positive, integral vector lengths. Note that vector length may be limited by external constraints such as the physics of the target machine or the limits of the operating system used.

Data Types

We must next decide what type of data the elements of a vector can consist of. The type of data which current SWAR architectures were designed to handle falls into two primary categories, both of which allow signed or unsigned data:

1. Integer data of various precisions typically representing digitized sampled analog signals or digital values generated by some multimedia program.

2. Single-precision (32-bit) and double-precision (64-bit) floating-point data typically representing the value of some physical property or the placement and/or orientation of some object in the three-dimensional real-world.

Unfortunately, people considering the use of SWAR architectures often limit their view to only common multimedia data types and thus overlook other categories of data which could be operated on using SWAR technology. Two examples are character and Boolean data. Each of these is used extensively in various applications, but rarely is either treated as a parallelizable data type.

The data types supported by the programming model may differ from those supported by the target architecture if these data types can be emulated or promoted internally by the compiler. This requires that the operations defined by the model be implemented using the data types and instructions supported by the target architecture. Where this can be done, the SWAR model need not be limited to the data types which are directly supported by the hardware. We can thus consider other possible data types and decide to what extent these types should be supported by the new model.

Integer Data All SWAR architectures support parallel integer processing at some level, but usually do so only for standard multimedia data sizes. This is based on the seemingly reasonable assumption that programmers want to use the data sizes that are natural for the data they are manipulating and that these types are known to language designers. For example, people working with grayscale pixels want 8-bit objects and those working with color pixels want 24- or 32-bit objects.

The problem with this assumption is that it eliminates generality from the language. No one knows what will be the full range of vector applications that people will invent. The data they may wish to manipulate may be best described using 3-bit or 6-bit objects. If so, the programming model shouldn't prevent the programmer from expressing operations in these terms, even if the compiler is eventually forced to implement them using other data sizes. To exclude data sizes from the model is to eliminate the possibility of exploiting them.

Perhaps an analogy is called for here. Suppose your favorite soft drink is root beer. You go to the local store to purchase a 6-pack and find that they don't carry it. Instead, you find a cola, which you decide is close enough. You check out, they use your shopper's card to track what you purchased, and then order more cola. A week later, you again go in looking for root beer, but only find cola. The process repeats a few more times. Now the store has a long history of your purchases of cola. They *know* that your favorite soft drink is cola. In fact, they know that many of their customers' favorite soft drink is cola. Because of this, they decide never to carry any other kind of soft drink, and will use your long history of buying cola as evidence of your preference for it.

The same thing happens with data types. Because everyone uses 8-, 16-, 32-, or 64-bit data sizes, why support anything else? If you look at all the C code ever written, you'll see that nobody ever uses 2-bit data types. They can't because there aren't any. This circular reasoning is used as an excuse to avoid providing more general programming models.

SWAR operations on data of non-standard precisions such as this can be performed using reasonably straight-forward, if not always efficient, methods of emulation. We shall see that this is possible on both multimedia-enhanced and unenhanced architectures. Also, data which has an unsupported precision usually can be promoted to some supported type by the implementation of the model (i.e. the compiler or library). Thus, it is often a straight-forward task to emulate operations on this type of data.

Because we can easily emulate operations on small data sizes by promoting them to larger, supported sizes, it is illogical to have the programming model enforce the use of only a few data sizes. If we do not adjust programming models to allow for more flexibility, we will pay a performance price when single-chip, bit-slice parallel architectures become widely available.

In order to support the widest range of applications, the general-purpose SWAR programming model supports integer data of any bit precision. As with vector length, external constraints may place bounds on the precision of data supported, but the model itself does not. For example, precisions greater than the number of bits in one of the target's registers may be disallowed by the implementation.

Floating-point Data While several SWAR architectures support floating-point processing, a significant number do not. These architectures would require emulation if floating-point processing is included in the model. Such emulation is usually difficult to do efficiently.

A floating-point operation is a series of integer operations which denormalize the data, then operate on the integer mantissa and exponent separately, and finally normalize the result. These steps can be done on an integer architecture, but the number of steps involved will probably offset any gains made via parallelization.

This should become less of an issue in the future as more SWAR architectures incorporate floating-point support. For now, a portable SWAR model should not require the incorporation of floating-point operations, but should not prevent them either. That is, support for parallel floating-point operations should be architecture-dependent.

If it makes sense to allow any precision of integer data, why not allow any precision for floating-point data? From a theoretical stand-point, there is no reason not to do this. Suppose we have real-valued data that is limited in range to a set of values that can be expressed using a 4-bit mantissa and a 4-bit exponent. Why should we not be able to express this?

Again, the problem becomes one of finding the balance between generalization of the model and limiting it to discourage operations which are unlikely to provide performance gains (or worse, likely to cause losses). Current architectures are generally limited to 32-bit parallel floating-point operations, with only SSE2 supporting 64-bit floating-point operations. Any other size of floating-point data will require emulation.

The emulation of odd-sized floating-point operations is possible, but is probably unreasonably inefficient. On an integer-only architecture, it is inefficient for the same reasons that emulating single- and double-precision floating-point operations are. For sizes which do not match a standard integer size, it is even worse. If standard-sized floating-point operations are supported by hardware, then the possibility exists for using temporary promotion techniques. In this case, the compiler needs to be able to manipulate the bit patterns of the floating-point data in order to create the proper form for calculation and extract the correct bits from the result. Again, this would probably be unreasonably inefficient and may even require that the data be moved to an integer register first.

Because non-standard floating-point types can be unreasonably difficult and inefficient to emulate, and because it is unlikely that they will become widely supported in the near future, there is probably no significant loss in excluding them from a current SWAR model. Thus, the current general-purpose SWAR model will only support 32- and 64-bit floating-point data on an architecture-dependent basis. Support for non-standard floating-point types will be left for the future.

Character Data Character data is often overlooked as a parallelizable data type because it is not considered numeric. However, characters are in fact typically stored using an integer code. For example, the ASCII [181] character set consists of 7-bit integer values which are used for storing and transmitting text. Thus, many operations on character data are in fact integer operations, even if the programming model used does not treat them this way.

Consider searching for a string in a text stream. This is a parallelizable task that could benefit from SWAR functionality. In fact, the size of a character on most systems (8-bits) is the same as that of a 256-color pixel — a data type which is well-supported by most multimedia extensions. However, in order for this algorithm to be parallelized, the model must treat the data as having a parallelizable type. A well-designed SWAR model should do this. Thus, the general-purpose SWAR model treats character data as a form of integer data with the same attributes with respect to parallelization.

Boolean Data Boolean (true/false) data could also benefit from SWAR processing, especially given that this information can theoretically be represented with one bit per datum. In this case, bitwise logical operations can be used to perform parallel operations across the single-bit fields of a register. This yields the highest possible parallelism on a SWAR system and should thus be supported. Similar to the handling of character data, these logical types are treated as a form of integer data by the general-purpose SWAR model.

Enumerated Data Enumerated data types should also be supported. For example, in a digital logic simulator, we may want to represent four states for each contact point between gates: high, low, high-Z, and indeterminate. This would require 2-bits per contact point. This data size does not match any multimedia data type, so multimedia architectures do not support it. Consequently, the programming models developed for these architectures fail to provide any means of expressing data of this form. This prevents the programmer from obtaining the highest possible performance when using enumerated data even if the hardware can directly support it. A good SWAR programming model should not impose this type of restriction. Thus, as with character data, enumerated data is treated as a form of integer data by the general-purpose SWAR model. An implementation may provide for explicit enumerated types such as in the C language.

Aggregate Data Elements consisting of aggregate data types such as C `struct`s or Pascal `record`s may also be useful. Data such as vectors of complex numbers could be represented in this manner with multiple elements stored in a single register or with each element striped across multiple registers. Other types or representations of data such as cylindrical or spherical coordinates could also be expressed as vectors of aggregate data.

While this may be useful, it opens up a new set of questions that which should be avoided for the time being. For example, which of the two layouts just mentioned, unstriped or striped, would be the better default method of representation? Should the user be able to specify which to use? If so, should this be via an explicit or implicit indication in the language? If not, how should the compiler make this decision?

Other questions also arise. For example, how large or complex a structure should the implementation consider to be parallelizable? Should the compiler be responsible for determining when to parallelize a vector of aggregate elements? If so, then a compiler implementing the model becomes significantly more complex than it would be without vectors of aggregates. If not, then some limitation must be built into the model to free the implementation from making this decision. While these questions are interesting, they should be avoided at this time to make the work reasonably manageable.

One may also dismiss vectors of aggregate objects for the simple reason that they do not fit well with the operation of current SWAR architectures. While some aggregate types are equivalent to small arrays of identically-typed data, in general they are comprised of objects of dissimilar types. Such types differ from the identically-typed parallel streams which SWAR instructions expect. Rather than trying to distinguish between these classes of aggregate elements, we will reject them altogether.

3.3.3 Parallel Operations

The general-purpose SWAR model treats vectors as first-class objects. Thus, a language which implements the SWAR model should support a fundamental set of vector operations in a manner which is easily expressed and meaningful. This set of operations should reflect those which are typically performed on vectors, but must also reflect the capabilities of current SWAR architectures. The operations that are supported by the SWAR model must be chosen to balance these goals. In this section, I build on the analysis of multimedia extensions from the previous chapter to delineate a set of operations that should be supported by a language which implements the general-purpose SWAR model.

Modular and Saturation Operation

One issue that can be addressed before specific operations are discussed is that of modular versus saturation operation. Recall that modular operations store only the low bits of the result which will fit into the destination, throwing any overflow bits away. The stored result is the calculated result modulated by the maximum storable value. Saturation operations handle overflow by fixing the result at the most positive or most negative representable value depending on the direction of overflow.

Multimedia operations are often performed on data which represent digitized samples of analog signals. Instructions which operate on this type of data need to do so without changing its meaning. For example, digitized music may be played through a "mixer" program which adjusts the relative strength of various data sources. An attempt to increment the strength of a signal beyond the highest value should not result in the lowest value. This would cause the signal's strength to drop unexpectedly and thus unacceptably. It would be better if the signal strength simply stayed at the maximum. Saturation operations were developed for this type of situation.

Certain multimedia extensions expect the data to be of this type and thus provide only saturating operations while others assume that the data should be handled modularly as with traditional computing. Other extensions use one or the other depending on the data size and the operation performed. Thus, there is significant variation between SWAR targets.

As a general-purpose model, SWAR should support both types of operations. Exactly how this is done is left to the implementation. For example, the SWARC language described in the next chapter associates saturation or modularity with the type of the data vectors. The type of operation applied is based on the resolution of the data types of the operands. Other languages based on the SWAR model could instead associate saturation or modularity with the operations themselves. For example, separate operators could exist for modular addition and saturation addition. This mirrors the actual operation of the hardware.

Operations which, by their nature, never overflow have equivalent modular and saturated forms, and should be included for both if for either. For example, unsigned integer division always results in an integer value which is smaller in magnitude than the dividend. Thus, it never overflows, so the modular and saturated cases never differ.

Arithmetic Operations

Basic modular and saturation arithmetic functions should be included for all data types and precisions with some caveats.

The general-purpose SWAR model includes modular and saturation addition and subtraction for all data types. Binary maximum and minimum are also included for all types. These are non-overflowing by nature, so there is no difference in their behavior under modular or saturation operation. Unary negation is also included for all signed forms and can be emulated as subtraction from 0 if necessary. Unsigned unary negation is optional.

Multiplication is included in all cases. One may wish to avoid saturation multiplication which is sometimes expensive to emulate. However, it should probably be included for the sake of completeness, and for this reason it is included in the general-purpose SWAR model.

Division, which generally results in a value that is within the bounds of the dividend and is thus non-overflowing (with the exception of signed division of the largest negative number by -1), is included in all cases. Modulus (division remainder) is included for integer vector types. Its result is always smaller and of the same sign as the dividend, and thus never overflows. Modulus is nonsensical, and thus excluded, for floating-point types.

Binary averaging is supported by most current architectures and is easily emulated for most types. However, in multimedia applications, averaging usually involves a rounding step which does not follow normal arithmetic rounding rules. For this reason, averaging is considered optional. If it is supported, it should be included for all vector data types and should be clearly and consistently defined. Also, because its result always falls between the two operands and thus never overflows, both modular and saturated versions should be supported.

More advanced operations such as square roots and exponentials should be avoided due lack of consistency or availability across architectures. These operations are not easily emulated and would thus be difficult to port between architectures.

Reductive Arithmetic Operations

Reductive versions of associative arithmetic operations are also included in the general-purpose SWAR model. The order and method of reduction are dependent on the implementation. This allows reordering of operands and logarithmic or serial implementation. Reductive versions of non-associative operations are not supported by the model. Thus, reductive addition and multiplication are allowed, but reductive subtraction, division, and modulus are not.

Combined Arithmetic Operations

The combined arithmetic operations supported by the various extensions are not consistently implemented across architectures, and should thus be avoided for portability sake. This does not preclude the use of instructions which perform these operations because any implementation of the model is free to optimize code sequences when possible. Such operations include MMX's multiply-add (MADD) instruction which performs a parallel multiply followed by a semi-reductive addition.

Certain vector operations also fall into this category. For example, it could be argued that a vector dot-product should be one of the operations defined by the model because it is a common operation in vector mathematics. However, it could also be argued that dot-product is really a composition of an elementwise multiply and a reduction addition and is thus redundant. Exactly how this common operation is provided for, if at all, should probably be left as a language definition decision.

A similar question arises for vector cross products. These operations generate a multi-dimensional array from two single-dimensional vectors. Because we would like to avoid array processing in the current model, we should avoid vector cross-products at this time.

Shift and Rotate Operations

Simple shifts include logical and arithmetic shifts left and right. These are well-supported across the various integer extension sets with the exception of VIS, which requires some non-trivial patchwork. For VIS, the `aligndata` instruction can be used to perform byte-wise shifts while its various pixel packing instructions can be used to perform bit-wise shifts. Because simple shifts are widely implemented and fundamental to bit processing, they are included in the general-purpose SWAR model.

Rotates are directly supported only by AltiVec, but they can be emulated with relative ease using shifts and polymorphics. Thus, the inclusion of rotates in a generalized model are debatable, but probably worthwhile. Both left and right rotates should be included for symmetry. The general-purpose SWAR model includes each of these.

Combined operations such as shift-and-adds are only supported by a few architectures and should be excluded from the general-purpose model as separate operations.

Bitwise Logical Operations

Bitwise logical (a.k.a. Boolean [182, 183]) operations are the basic building blocks of all complex binary computation [184]. These operations allow programmers to perform more complex operations than are directly supported by the model. Thus, a model which includes these operations is both extensible and powerful. These operations should be part of any programming model that includes the concept of a Boolean type or exposes binary digits to the programmer.

A programming model need not support every type of Boolean operation, but should include a working set. This set might not match that of any target architecture. For example, a binary NAND operation is sufficient to perform any other Boolean operation; thus, no other Boolean operation is necessary. However, it is often easier for the programmer if a larger set of Boolean operators is provided. For example, AND, OR, and NOT are often available and are familiar to most programmers. The particular working set implemented is left as a language-dependent decision.

Bit-Reduction Operations

Reductive versions of the working set of associative bit-wise logical operations should also be supported. More complex bitwise reductions, such as population counts, need not be visible to the programmer. Instructions which perform these operations are scarce and are usually difficult or expensive to emulate. Thus, they are excluded from the general-purpose SWAR model.

Conditional Operations

A reasonable set of conditional operations needs to be supported in order to allow decisions to be made. Otherwise, the usefulness of the model will be severely limited. As with bitwise logical operations, only a working set needs to be chosen when the model is implemented as a language. However, to promote self-consistency within the model, a complete set of conditional operations should be included.

One issue concerning conditional operations is whether they can appear outside the test sections of conditional constructs. Some languages disallow the use of conditional operations anywhere other than in these test sections. However, there are also languages which assign numeric values to these conditional expressions and allow them to be used within arithmetic expressions. To allow as much flexibility as possible, conditional operations should be assigned some value. Doing so requires definitions for these values. These values are dependent on the implementation of the model.

Another issue is that of "orderedness", which is probably better referred to as "orderability". Certain bit patterns are not interpretable as valid floating-point numbers. IEEE standard 754 defines these patterns as NaNs (Not-a-Numbers). The value of a NaN cannot be compared to other values, thus they are said to be "unordered". NaN patterns are not normally generated by a properly written high-level program operating under well-defined circumstances; however, they may result from improper conversion or interpretation of integer values. Thus, it should not be necessary, nor would it normally be desirable, to expose this aspect of floating-point operation to the programmer. For these reasons, these tests are excluded from the general-purpose SWAR model.

Reductive Conditional Operations

Reductive versions of the working set of conditional operations supported by an implementation of the model may also be supported. For example, a language may support a reductive greater-than operation which is true if the elements of a vector are ordered and false if they are not. These are somewhat esoteric operations, and difficult to emulate, so we may wish to avoid them. However, their inclusion would provide another level of consistency. Given this trade-off, these operations should probably be optional.

Logical Operations

Logical operations are used to combine conditional operations into more complex expressions. These enable programmers to create more complex tests than simple conditional operations allow. A working set of these should be included in any implementation of the general-purpose SWAR model. As with conditional operations, the results of logical operations need not be visible to the programmer but allow more flexibility if they are.

Reductive Logical Operations

Reductive associative logical operations produce a result which represents the aggregate condition of the parallel elements. For proper execution of conditional constructs under SIMD semantics, an implementation must internally perform operations of this sort. For example, a parallelized `while` loop should be executed while the test condition holds for any of the parallel elements. This “any” test is essential a reductive logical-OR of the result of applying the conditional operation to the parallel elements.

In terms of a programming model, the question is whether the programmer should be provided with mechanisms for performing similar operations. As with non-reductive logical operations, it is arguable whether the results of these operations should be exposed. Again, visibility allows for more flexibility. Thus, these operations should probably be explicitly available to the programmer. Therefore, a set of reductive logical operations which complement the chosen set of associative logical operations should be included.

Conditional Assignment Operations

Conditional assignment is yet another issue. “Pick” instructions select one of two possible results based on the value of an index register. Their operation is similar to that of the C trinary operator, in which the result of a conditional test causes one statement to be executed if the result is true and another to be executed if the result is false. In the case of a pick instruction, the executed statements would both be assignments to the same variable. Because this is actually a shorthand version of a particular `if-else` construct, it is redundant. Inclusion of such an operation should thus be optional.

Data Storage and Movement Operations

Imperative languages represent the storage of data using assignment statements. These are operations in which a value is stored for future processing to a storage location designated by a *variable*. This allows long, complex expressions to be split up into smaller ones, thus simplifying the expressions used. It also allows a programmer to reuse *common subexpressions*. These are expressions which appear in one or more others. Thus, the task of coding is made easier by the use of assignments.

An imperative vector model should allow vector assignment. That is, it should allow data to be assigned to a vector as a aggregate object. A simple example would be copying one vector to another. This should be expressed as a single operation, not as a series of operations on the vectors' elements. Thus, the SWAR model allows vector assignment.

On assignment, data may actually be stored to a memory location or register. Usually, the difference in destinations is hidden from the programmer and registers are used only by the compiler. Thus, assignment is an abstraction which hides the actual operation performed. As an optimization, instructions which perform moves between registers may be used internally by a compiler to implement assignments when an actual memory access is unnecessary. This can increase performance by allowing stores to be used only when the data must be written to memory.

Instructions for moving data between vector registers are often used to copy data before performing an operation which destroys one of its operands. They are also used to make a copy that can be handled differently from the original. These operations are usually internal to the compiler and not exposed to the high-level programmer. However, some languages do allow explicit assignment to "register" variables as a means of hinting that the data will be used often or does not need to be stored in memory. Although it should be unnecessary, exposure of the use of registers is considered an implementation-dependent issue.

Instructions for moving data between scalar and vector registers are used to load or store vector fragments when this cannot be done directly between the vector registers and memory. They are also used to allow operations to be applied to vector fragments which cannot be applied to them while they are in the vector registers. These instructions would normally be applied internally by the compiler as part of a multiple instruction operation. There should be no reason to expose this to the programmer.

A well-designed vector model should allow scalar to scalar assignment to allow vector elements to be operated on in a reasonable manner and to ease the construction of mixed expressions which include scalar subexpressions.

Reductive Assignment Operations

A well-designed vector model should also allow vector to scalar assignment. This is often the last step in a parallel processing algorithm in which data has been distributed to multiple processors for identical processing. This separates the task into parallel subtasks whose results must be later combined. This combination step is a reductive step in which some function of the subresults is performed to obtain the single result of the overall task.

This step should be easily expressed as an assignment of a vector to a scalar. Because there are various operations that one may wish to perform to obtain the single result, a variety of reduction operations should be available for use in this last step. Conceptually, the result of the reductive function is stored in a scalar storage location; thus, this step should be representable as a combined reductive assignment operation. Any of the reductive operations included in the model should be combinable with assignment to provide vector to scalar assignment.

Replicative Assignment Operations

Scalar to vector assignment should also be supported. This is often one of the first steps in a typical parallel processing algorithm. The initialization of vectors to a single value, such as zero, is a common operation. The scalar value is replicated and assigned to each of the vector's element. Expressing this operation as a single replicative assignment of the scalar value to the vector object is a much more elegant solution than expressing it as a series or loop of scalar assignments to each of the vector's elements. Thus, the general-purpose SWAR model allows scalar to vector assignment which operates in a replicative fashion.

Type Conversion Operations

In a typed language, one may wish to provide for the conversion of data from one type to another. There are various reasons for this. A data's type usually defines its storage format. The primary purpose of type conversion is thus to ensure that data has the correct format during processing. This means that type conversion is equivalent to converting between data formats. This is necessary to properly evaluate mixed expressions, to ensure that data is stored in the proper format, and to match function parameter and return value formats.

When type conversion is performed internally by the compiler to support mixed expressions it is called *type coercion*. For example, it is sometimes useful to use data which is stored in an integer format in an expression involving floating-point data. The conversion of data from integer to floating-point formats is necessary for this type of processing to be performed properly. Most languages have semantic rules which define when such conversion takes place.

Type coercion is also performed when an expression has been evaluated to a value of one format and needs to be stored in a location which has a different size or is assumed to hold data of another format. The value must then be converted to the correct format before being stored. This conversion is typically internal to the compiler, but is known to the programmer via the semantics of the language.

Often, conversion can be performed explicitly by the programmer using *type-casting* operations. These allow the user to perform conversion outside of mixed expressions and other situations in which the compiler would perform implicit conversion. For example, when passing an integer value to a function which expects a floating-point value, it is convenient to simply perform the conversion without storing the data to a floating-point variable or constructing a mixed expression. Type casts allow the programmer to specify such an action.

As with other operations, the level of support that a model can safely include for type conversion depends on the capabilities of the target architectures. The various extension sets include a large number of instructions which can be used to convert data between various types. Some of these were intended for this purpose, while others were not. Some instructions allow data to be converted between integer types of various sizes, while others can be used to convert between floating-point and integer data types.

Packs and unpacks can be used to convert between integer types of various precisions. As defined previously, packing instructions convert data to smaller precisions, then pack them into a smaller section of the register without changing their relative order. This is equivalent to performing a vector type conversion from one precision to another. In current multimedia extensions, this conversion is accompanied by a saturation operation. This forces each data element to the representable value nearest its original value.

Unpacking instructions perform the inverse of packing instructions, converting data to larger precisions using sign- or zero-extension as necessary. As with packs, this is equivalent to performing a vector type conversion between precisions.

Interleaving instructions also can be used to convert integer data from a smaller to a larger precision. This is done by filling a register with zeroes or with fields which are filled with the sign bits of the corresponding fields of the original register. These are then interleaved to form larger fields of zero- or sign-extended data.

Instructions which directly convert data between floating-point and integer forms are included in several extension sets. In implementations of the model which allow floating-point data, these instructions may be used internally to implement type coercion or explicitly to implement type casts.

In order to allow maximum flexibility, a general form of type casting should be included in the model and type coercion rules should be defined to allow for mixed-type and mixed-precision expressions. These rules are implementation-dependent.

To handle mixed-dimensional operations which are applied to a vector and a scalar, it is sometimes useful to convert the scalar operand to a vector which "conforms" to the shape of the vector operand by replicating the scalar's value. This allows computation to proceed using vector operands only. This conversion may be done implicitly as with type coercion or explicitly as with type casting.

Support by the various multimedia extensions for replication is mixed. Only AltiVec has explicit replication instructions. A few extensions have a number of operations which can pair a partitioned operand with a scalar one. However, most of the extensions have little support for replication or mixed operations. Despite this, as a general rule replication can be emulated using polymorphics and shifts. Thus, they are reasonably portable, though often inefficient, and should not be excluded from the model. For this reason, scalar to vector conversions via type coercion and casting are allowed by the general-purpose SWAR model. This promotes flexibility while simplifying the programming task.

Vector Element Access Operations

To provide generality and to ease the handling of boundary conditions and singularities, general-purpose vector programming models should allow vector elements to be operated on individually. Where available, extraction and insertion instructions can be used to implement vector element accesses. These instructions allow sections of a partitioned register to be isolated for further processing or recombined with other data.

While several multimedia extensions contain this type of instruction, others do not. On these architectures, it is generally possible to emulate basic forms, although several instructions may be required to do so. Thus, they should not be excluded. The SWAR model assumes that vector elements can be individually accessed, operated on, and assigned as scalar objects.

Vector Generation Operations

One problem that is not well-addressed by current multimedia extensions is that of combining single items of data into partitioned form. That is, the creation of a vector from a set of scalars. This often takes several steps because data must be positioned, masked, then inserted into the destination.

This leads to the question of how such an operation should be expressed by the programmer. Specifically, should the programmer describe this as a single operation or as the several operations that are typically used? By using elemental assignment, the programmer can express this as multiple separate operations. However, mechanisms which allow vector generation to be expressed as a single operation would also be useful and should be included in an implementation of the model.

Vector Catenation Operations

Mathematical vectors are not often concatenated, but the catenation of character vectors (i.e. strings) is a fairly common operation. A general-purpose model should include operations of this type.

Linear Interelement Communications Operations

Shifts and rotates can also be used to emulate one-dimensional communications operations, treating the register fields as in a linear array or ring. This is the most natural form of "inter-PE" communication for these architectures, and one which closely represents the use of traditional SIMD interconnects. Thus, inter-field shifts and rotates should be supported if only from a communications stand-point. Specifically, linear communication between data fields is supported by the SWAR model via vector shifts and rotates. These move data linearly and regularly between vector elements.

Non-linear Interelement Communication Operations

Advanced communications operations such as shuffles and permutations require more complex operations than most current multimedia extensions support. Because of this, these more advanced communications operations will be avoided. We will, however, discuss the capabilities of current SWAR architecture with respect to communications operations.

Interleaving instructions combine data in two registers by alternating between them, while swaps exchange data between the fields of a single register. These instructions can be used to implement various forms of interfield communication which exhibit regular access patterns. Neither interleaves nor swaps are consistently implemented across multimedia architectures. Thus, the particular communications patterns exhibited by these architectures differ. Because of this, communications operations with patterns which require this type of operation should be avoided in the current general-purpose model.

The catenating instructions included in the various multimedia extension sets combine subsets of their operands' elements without changing their relative order. Thus, these instructions also perform operations which resemble various regular communication schemes. As with interleaves and swaps, these instructions are inconsistently implemented so the operations they perform should not be included in the current model.

Packs and unpacks can also behave like regular communications operations. Packs gather data from alternate fields (PEs) of a register and pass it to a contiguous set of fields, while unpacks perform the inverse operation. These operations are not implemented by all multimedia extensions, are inconsistently implemented when they are, and can be expensive to emulate. Thus, the communications operations they represent should also be avoided.

Permutation instructions allow the fields of one or two registers to be rearranged or replicated. These operations are equivalent to communications using advanced interconnection networks such as the router networks of the Thinking Machines' CM-2 or MasPar MP-1.

Only a few extension sets include permutation operations. Due to their generality, they are difficult to emulate on architectures which do not support them. This makes them difficult to port. Thus, these operations should be avoided despite their acceptance and use in previous SIMD programming models. These operations are thus excluded from the current SWAR programming model.

As technology advances, more architectures will incorporate advanced interconnections between the fields of their registers. This will allow more complex operations such as permutations to be portable between architectures. At that time, advanced communications should be incorporated into the model. Until then, incorporation of such operations will only encourage the programmer to write code which cannot be implemented efficiently on most SWAR architectures.

Cache Management Operations

Cache management is inherently architecture-dependent. One must be aware of the size of cache lines and memory blocks to order operations intelligently. For example, when should a hint be given that a memory location will soon be needed?

Should it occur at the beginning of a block of code or the beginning of the statement in which the access occurs? This depends on the size of the source code block versus that of the cache lines. This decision also requires consideration of the availability of space in the cache.

Generally, this knowledge should be hidden from the programmer so that he or she may concentrate on the description of the algorithm at hand, not the mechanics of execution or machine control. Moreover, an optimizing compiler is likely to make modifications to the order of execution. This leaves the programmer without clear knowledge on which to base cache management decisions. In this case, cache management operations would blindly impose constraints on the reordering of instructions.

For these reasons, cache management operations should not be exposed to the programmer in a portable programming model, and are not in the general-purpose SWAR model.

3.4 Properties of a Well-Designed High-Level Language for SWAR

With the completion of this phase of research, we are now in a position to enumerate a set of properties that a well-designed high-level SWAR language should exhibit, and also to establish guidelines for implementing the general-purpose SWAR model as a full-scale high-level programming language.

The primary characteristics of such a language are:

- The primary parallelizable object is a one-dimensional vector.

- Vectors consist of one or more identically-typed data elements.

- Vector element types are architecture-independent.

- The elements of a vector are identical in type and precision.

- The elements of a vector are single-valued and non-aggregate.

- The elements of a vector have integer or floating-point type or some other type which is treated as a form of one of these types.

- Vector integer elements may have any precision subject to external constraints.

- The allowed precision and handling of floating-point vector elements is implementation-dependent.

- The layout of a vector in memory is implementation-dependent.

- Vector operations are consistent across data types and precisions.

- Vector operations are architecture-independent.

- Vector operations are closely matched to the capabilities of current SWAR architectures.

3.5 Development of the Model

The general-purpose SWAR programming model was developed jointly by Professor Hank Dietz and me to address several concerns.

Originally, Professor Dietz suggested that the we should look at multimedia extensions such as MMX because he believed that they would be interesting architectures to target.

We then designed the SLIME (SIMD Language for Intel Multimedia Extensions) programming language for use in the fall 1996 undergraduate Compiler and Language Translation Systems Course (EE468) which he was teaching and for which I was the assistant.

This language is a small MPL-like SIMD language in which the number of programming elements depends on the precision of the data to be operated on in parallel.

There are two data types in the SLIME language: `int` and `plural`. An `int` is a single standard C integer which is visible to each of the PEs. A `plural` is a multiple-valued object with a single name making only one element of the object visible to any given PE.

The precision of a `plural` object is given on the command line when the compiler is run, and is required to be one of 8, 16, 32, or 64. All plural objects are compiled with this precision and have a fixed number of elements. This number is 64 divided by the given precision. Thus, a plural object fills a 64-bit wide MMX register perfectly.

Originally, students were to implement a compiler for SLIME which would generate C-code using macros to execute the necessary MMX instructions. However, while the SLIME programming model requires all operations to be implemented for any of the given precisions, MMX does not include instructions for each of these. Thus, unsupported operations required emulation which we did not want the students to have to implement in the time allotted.

Subsequently, I made a brief survey of the multimedia sets then in existence. Over time, I have expanded and refined this survey into the tables found in chapter 2. During my initial investigation, I found the available multimedia extension sets to be both incompatible and incomplete. Also, it was clear that none of these extensions were designed to support a general-purpose parallel processing model, but were instead intended to support particular algorithms. Professor Dietz then suggested that perhaps we should attempt to develop a general-purpose model.

As we began working on this model, I realized that the supported data sizes were chosen based on the designers' beliefs about which data sizes would be most commonly used by their respective customer bases. Because this had led to incompatible extension sets, it was clear to me that this was not the path to follow when designing a general-purpose model. Rather than to assume knowledge of the data sizes needed by the application programmer, I argued that one cannot, and therefore should not, guess which data sizes will be most useful to a future applications programmer.

It was also clear that limiting the size of the parallel data set to fit into one register was not necessary, and that few real applications would use data sets of exactly the "correct" size. For example, there are 3.2 billion gene pairs in the human genome.

A good programming model should not preclude the description of algorithms which address large data sets such as these. In fact, a good model should allow the programmer to describe operations on these data sets easily. Thus, as a basic model, we opted for a SIMD model in which vectors are first-class objects with any number of elements of any precision.

It is important here to stress the difference between the model and any particular implementation of the model. Practical considerations, such as finite memory, cannot be avoided; and certain situations, such as data precisions which are greater than the size of a register, will not result in speedup. While a particular implementation of the model may avoid these situations, they should not be incorporated into the model.

As an example, suppose we had chosen to limit data precision to the maximum precision that would have provided speedup using MMX. Because MMX registers are 64 bits wide, the maximum size would have been 32 bits. While none of the extension sets contemporary with MMX included instructions for data which exceeded this precision, several current extensions do. Had the 32-bit limit been incorporated into the model, it could not have been used by a programmer to take advantage of the 64-bit capabilities of these architectures when they became available. Similarly, if we incorporate a 64-bit limit into the model, it will not allow the programmer to take advantage of any 128-bit hardware support in the future.

4. PROOF-OF-CONCEPT IMPLEMENTATIONS OF THE MODEL

Having defined a new abstract model of parallel computation which better reflects the capabilities and limitations of modern SWAR architectures than do current computational models, we now develop prototype implementations of this model and optimizations which exploit the capabilities of various target processors.

4.1 Prototype Libraries for SWAR Processing

My original plan of study called for the development of a set of small, portable libraries for writing SWAR algorithms. These were to be optimized to their target architectures and share a common portable interface to show that the model could be applied in this manner.

Two prototype libraries were created to address this goal. The first, called libMMX, provided a means to access MMX instructions in a manner similar to C function calls. The second, SWARlib, was intended to show that a portable library interface could be developed for SWAR processing.

4.1.1 libMMX

I started by creating the original incarnation of the libMMX library [185]. This provided access to the MMX set of extensions via C preprocessor macro calls. This library defined a `union` type, equal to the size of an MMX fragment, which could be treated as a repartitionable array of fields. Operations on objects of this type were performed using macros which hid the actual register usage from the programmer.

This is the correct approach for a portable library, but makes the library useless as a compiler target. Later versions of this library [186], based on a version by Professor Dietz, were written to expose register usage to make them more useful for compiler work. A set of similar libraries are used by the Scc compiler discussed in section 4.3 to support its various targets.

Following the development of this prototype library, we decided that the design of a high-level programming language and compiler should be given higher priority than was originally called for in the research plan. We felt that a compiler would be needed to perform aggressive optimizations and instruction scheduling in order to achieve a reasonable amount of speedup over large code segments.

When using libraries, the programmer is forced to perform these tasks and is less likely to achieve significant speedup over a large amounts of code. For this reason, development of a portable SWAR library was not pursued until after the compiler was relatively mature; and then, only as a proof-of-concept implementation. The resulting library framework was called SWARlib.

4.1.2 SWARlib

SWARlib does not implement a full general-purpose model, but implements enough of one to show that it could be done. Currently, SWARlib has only been targeted to MMX and AltiVec, but would be implemented similarly for any target.

SWARlib allows the programmer to create vectors of unlimited length, but violates the requirement that any field size be allowed by only allowing power-of-two field sizes. It also limits field sizes to those smaller than a fragment, but this is allowable as it does not limit the obtainable parallelism.

Each vector is implemented using a C `struct` which contains the vector's type and layout information and also a data pointer. This pointer points to an allocated area of memory which is treated as an array of fragments holding the actual data[1].

In an application, vector pointers, called `swar_vector`s are first declared, then `swar_alloc()` is called to allocate and initialize the data structure for the vector. This function takes a vector length, data precision, and signedness and saturation indicators as arguments, stores this information, and allocates memory to hold the vector data. The user is responsible for initializing this data after the return.

The `swar_vector` names can be used in calls to macros which implement the basic operations of the model. Type information is not passed explicitly in these calls. This provides a level of abstraction which makes the vectors look somewhat like first-class objects; however, basic operations must be performed via functions or macros rather than by using operators as one would with truly first-class objects. An implementation in an object-oriented language would allow first-class operation using operator overloading, but would be limited in the operators allowed.

SWARlib could have been implemented as a library in which type information is given as part of the name of each operation performed, but the current implementation more closely matches that of the SWARC language (section 4.2). This has a negative effect on type assessment. While a compiler can perform type assessment statically and arrange for correctly typed operations to be performed, a macro library in which data types are passed as arguments or as part of an argument must be assessed during execution, thus making the resulting code slower than the corresponding code generated by the compiler.

An example of a SWARlib vector operation is `swar_add()`. This macro takes three pointers, which look like simple variable names, derives type information from the underlying data structures, and then performs a (hopefully) properly typed vector addition. This is done by executing the MMX or AltiVec instruction(s) necessary to

[1] This area is dynamically allocated and thus needs to be encapsulated to force correct alignment. This is not currently done, but is a relatively minor error which should not need to be corrected to prove the viability of SWARlib.

perform the operation on each of the corresponding pairs of fragments of the source vectors and storing each subresult in the corresponding fragment of the result vector.

Currently, SWARlib assumes that the result type is the same as the destination argument, and treats the sources as being of this type. This is incorrect because the result type should be a resolution of the source arguments' types cast to the destination's type. While this leads to incorrect results, it is something that an observant user should be able to work around, and it should not be necessary to fix this to prove that the SWAR model can be implemented as a library.

Each target has a set of operations for which it lacks hardware support. These must be emulated in the library; however, the library currently contains no emulation. Emulation in SWARlib would be similar to emulation in the Scc compiler described in section 4.3. I am confident that it could be done in the framework of a library, and that it would be time-consuming to do so.

As an example of emulation, MMX does not have an instruction which performs an 8-bit unsigned maximum operation (max8u), but it does have an 8-bit unsigned greater-than comparison and a set of polymorphics. The max8u can be emulated as a series of operations similar to the following:

```
gt8u(arg0, arg1, i);
and(i, arg1, j);
not(i, i);
and(i, arg0, i);
or(i, j, i);
```

Here, `arg0` and `arg1` are the arguments to the max8u. `i` and `j` are temporary variables used to make the example clearer. In each call, the destination is the final argument with the sources preceding it.

After processing, each vector is freed by calling `swar_free()`. This deallocates the vector data and `struct`. Finally, `swar_end()` is called to perform any operations, such as MMX's `emms` instruction, necessary to put the processor back into a non-SWAR processing mode.

A completed version of SWARlib would be a full implementation of the SWAR model as a library. Using the methods described in this section, a portable library could be developed which would satisfy the requirements of the SWAR model. However, as was previously stated, it became clear that a fully operational compiler would be necessary to achieve significant performance over anything but a trival code sequence. Thus, this library was not fully implemented, nor was it made available to the public.

4.2 The SWARC Vector Language

After the development of basic SWAR libraries, the next task was to define and develop a new, high-level programming language based on the SWAR model for eventual placement in the public domain.

We chose to do this rather than to add new classes to an object-oriented language such as C++ because we be believed that it would be difficult for a C++ compiler to optimize vector code and because it would allow more flexibility for future research.

To simplify this task, we developed a module language which could take advantage of available C libraries and integrate well with ordinary C code. This allowed us to avoid writing support libraries for the language which would have been necessary to build a complete application otherwise.

The language itself is intended to allow the programmer to easily describe SWAR data and algorithms in a portable manner. The language is similar to C, but allows parallel data to be represented as vectors. In accordance with the general-purpose SWAR programming model described in section 3.3, vectors are first-class objects which can be of any length, subject to external constraints such as the amount of available memory.

To make applications portable between targets with varying word sizes, supported field sizes, and data alignments, the language only allows the programmer to specify minimal constraints on data precision and layout. This allows the compiler to choose which field sizes and layouts will actually be used based on the target's capabilities.

The remainder of this section describes the SWARC module language which we have developed and is adapted from [5].

4.2.1 Type System

Base Types

The SWARC language includes the C language's base data types to make the integration of SWARC and C code easier than it would be otherwise. This allows arguments to be passed from C code to SWARC functions without having to be cast to a vector form first. Aggregate types such as `struct`s and `union`s are not allowed in this prototype language; however, single-dimensional arrays of a base type are allowed.

The base types allowed in SWARC code are `char`, `int`, and `float`, with `char`s considered to be 8-bit `int`s. These may be modified with any of the modifiers `signed`, `unsigned`, and `const`. Also, the `int` type may be modified or replaced with the size modifiers `short`, `long`, and `long long`. The storage classes `extern`, `register`, and `static` can also be applied to a base type and have the same meanings as in C.

In addition to the normal C modifiers, two additional attributes are allowed in SWARC. These are the `modular` and `saturation` attributes which allow the programmer to specify which form of overflow handling should be used by operations performed on the object. Thus, overflow handling is specified by data type using a single operator for both modular and saturated operations.

The general form for declaring C data in SWARC is the same as in C with the exception that arrays are always one-dimensional and the dimension follows the base type name, not the name of the variable. The general form is thus:

storage-class modifiers base-type [*dimension*] *name* where everything but the base type and name are optional.

There are no explicit Boolean or enumerated types in SWARC. Each of these must be handled as a type of integer. Boolean types can be handled as 1-bit unsigned `ints`, while enumerated types can be handled using n-bit `ints` where $n = \lceil log_2 m \rceil$ for m values. Macro definitions can be used to assign names to these values.

Vector Types

SWARC extends the C type system with a first-class vector type which allows one-dimensional arrays of any length to be defined. These objects may be accessed as an aggregate entity as opposed to C arrays which can only be accessed one element at a time.

In its most general form, the type declaration for SWARC vector data specifies an object whose elements are laid-out either as an ordinary C array or packed as the compiler sees fit using a specified minimum precision. The syntax for declaring such an object is similar to the bit-field specification used in C `struct` declarations, and takes the general form:

storage-class modifiers base-type:*prec*[*width*] *name*

where the storage class, modifiers, and base type are as described above, and everything is optional except the base type and name.

The precision specifier : indicates that the object should have a SWAR (i.e. compiler-chosen) layout, and that the minimum precision required for the data may be specified with an optional integer *prec*ision. This precision specifies the minimum precision to be used for element data and may take any positive integer value subject to external constraints.

Omitting the precision specifier indicates that the object should have a C, rather than a SWAR, layout. Using the precision specifier without an integer precision is equivalent to specifying a SWAR layout with the native precision for the equivalent C layout type.

Note that while the compiler may store data with a higher precision than specified, saturation is always to the declared precision of the data. Also note that when a precision is specified, the integer base types, which include `char`, `short int`, `int`, `long int`, and `long long int`, are equivalent.

The optional `[`*width*`]` specifier indicates the C layout array dimension or the number of SWARC layout vector elements. If the `[`*width*`]` is omitted, it is taken to be one.

Examples

Some examples of SWARC declarations are in order at this point:

- "`char c`" is equivalent to the C declaration "`char c`", and specifies that `c` is a single variable of type char.
- "`float: f`" is equivalent to "`float:32 f`" on most architectures, and specifies one single-precision floating-point variable.
- "`int:7 i`" declares `i` to be an integer with at least seven bits of precision.
- "`long:[14] l`" declares `l` to be a vector with 14 visible elements, each of which is an integer field with the same number of bits of precision as a C object of type long int.
- "`char:7[14] c`" declares `c` to be a vector with 14 visible elements, each of which is an integer field with at least seven bits of precision.

Type Coercion

These type extensions require several modifications to the C type coercion rules. Scalar objects are promoted to the dimension of the other type. This allows a scalar object to be used as an operand to a vector operation without the programmer explicitly converting the scalar into a vector.

If neither object is a scalar and the dimensions are mismatched, the wider object is truncated to the width of the other. This can be used with vector shifts to extract subvectors from a vector if necessary. An implementation may optionally generate a warning about the mismatch.

Expressions which mix C and SWAR layout objects, result in the SWAR layout even if this requires the precision to be reduced. Otherwise, an expression with mixed precision yields a result with the higher precision. This is primarily to allow scalars to be converted to the precision of a vector (which is usually smaller) rather than forcing the entire vector to be converted to the precision of the scalar.

Finally, `modular` expressions are cast to `saturated` expressions when mixed. This ensures that overflow causes saturation even when generated by interaction with modular data.

Summary

This type system allows the programmer to specify vectors of any length and element precision, and thus conforms to the general-purpose SWAR model. It allows programmers to specify data types which match the precision of the data in question while leaving the compiler free to use the whatever precision and layout works best on the target architecture.

4.2.2 Control Constructs and Statements

Control flow constructs in SWARC are a superset of those in C, and operate similarly to those in MPL [107]. From the point of view of the programmer using SWARC, conditionally executed statements must be applied only to those vector elements for which the condition is true. Because SWAR instructions are applied across all the elements stored in a CPU register, a conditionally executed instruction must be applied under an *enable mask* which limits its effects to the elements which are enabled for the operation. SWARC control constructs must be modified to properly deal with this situation and to hide the underlying operations from the applications programmer. The SWARC constructs include:

- *if* statements, which operate as do C `if` statements if the conditional expression has a width (i.e. vector length) of one. Otherwise, the *if* body is executed iff the condition is true for some enabled element of the conditional vector. In this case, the body is executed under enable masking to limit the effects to those elements for which the condition is true. Likewise, the *else* body is executed, under masking, iff the condition is false for some enabled element of the conditional vector.

- *where* statements, which operate as do SWARC `if` statements, except that the *where* and *elsewhere* bodies are always executed. These bodies are masked to limit their effects to the correct set of elements.

- *everywhere* statements, which enable all elements of the vector for the statement which follows. These are used to temporarily interrupt the current enable state.

- *while* statements, which operate as do C `while` statements if the conditional expression has a width of one. Otherwise, the *while* body is executed as long as the condition is true for at least one enabled element in the vector. An element is disabled when the condition becomes false for that element, and stays that way until the loop is exited. Thus, the set of enabled elements is monotonically non-increasing with each iteration. Once all the elements become disabled, the loop exits, and the enable mask is restored to its condition before entering the loop.

- *for* statements, which are related to the SWARC `while` in the same way that the C *for* is related to the C *while*.

- *do* statements, which are related to the SWARC `while` in the same way that the C *do* is related to the C *while*.

- *continue* and *break* statements, which operate as in C except that an optional expression indicates how many nesting levels to continue or break from.

- *return* statements, which operate as in C except that no expression is allowed to be returned from a SWARC function.

- labels, block statements, and empty statements, which all operate as in C.

- function calls, which operate as in C except that arguments are passed by address, not by value. The call is executed as the body of an implied `everywhere`. This ensures compatibility with ordinary C code.

- A special block statement, which encloses ordinary C code and can be inserted wherever a statement can appear, or as a top-level declaration. These blocks are enclosed by a ${ $} pair, and will be emitted into the output code. Within these blocks, a dollar sign is used wherever a pound sign should appear in the output C code.

4.2.3 Operators

The standard C operators work as usual on C-layout data. Their definitions have been modified to work in a consistent and intelligent way with SWARC vector data:

- The unary and binary arithmetic operators operate as in C but in parallel on the elements of vector operands. These include addition and identity (`+`), subtraction and negation (`-`), multiplication (`*`), division (`/`), and modulus (`%`). Incrementation (`++`) and decrementation (`--`) are included only as prefix operators.

- The arithmetic assignment operators also work in C. These include additive (`+=`), subtractive (`-=`), multiplicative (`*=`), divisional (`/=`), and modular (`%=`) assignment operators. The associative additive and multiplicative assignments are extended as in C* [169, 187] to perform reductions when storing a vector value into a scalar or when the operator is used as a unary prefix.

- The bit shift operators (`<<` and `>>`) and assignments (`<<=` and `>>=`) operate as in C but are applied in an elementwise manner to vector operands. Bit rotates are not currently supported in the language; however, they probably will be in the future and use a notation similar to that of vector element rotates.

- The binary bitwise logical operators (`&`, `|`, and `^`) are included and operate as in C, but within each field on vector data.

- Bitwise logical assignment operators (`&=`, `|=`, and `^=`) are also included and operate as in C. These perform reductions when storing a vector into a scalar or when the operator is used as a unary prefix. The unary one's-complement operator (`~`) is also extended for parallel operation; however, there is no reductive version of this.

- Comparison operators operate as in C, but evaluate to 0 in every false field and -1 (all '1' bits) in every true field. This modification to the C definition makes the implementation of enable masking significantly simpler. These operators include less-than (`<`), less-than-or-equal (`<=`), greater-than (`>`), greater-than-or-equal (`>=`), equal (`==`), and not equal (`!=`) operators.

  Reductive comparisons are not included, primarily due to a notational conflict. Following the convention use for arithmetic and logical reductions, a reductive "greater-than" operations would be annotated as (`>=`) which conflicts with the greater-than-or-equal operator. Because they are not commonly used, these operations were not included in the language.

- Logical operators operate as do comparison operators. These include logical-AND (`&&`), logical-OR (`||`), and logical-NOT (`!`).

- The trinary conditional operator (`?:`) works as in C, but applies enable masking to block side-effects from affecting elements for which the condition does not apply.

- The C assignment operator (`=`) is defined as in C, but is extended to perform replication when assigning a scalar value to a vector and in elementwise fashion when assigning a vector value to a vector. Assignment of a vector value to a scalar is disallowed unless a reductive assignment operator is used.

- The typecast operator (`(`*type*`)`) has also been extended to allow SWARC types and is used as in C.

- The `sizeof` operator operates as in C, returning the size of its operand in bytes. This operand may be a type or object.

- The array generation operator ({}) has been extended to allow vector generation. This is typically used at initialization. Vector generation via concatenation is currently unsupported by the language. This is somewhat in keeping with the C language from which SWARC is derived.

- The array element operator (`[ ]`) has been extended to allow individual vector element accesses.

New operators have also been added to facilitate operations common to SIMD processing to be performed in the SWAR environment:

- Binary minimum (`?<`), maximum (`?>`), and average (`+/`) operators have been added to facilitate the computation of these values for scalars and vectors. Reductive unary and reductive assignment versions are also available, and take the forms: `?<=`, `?>=`, and `+/=`.

- Unary reductive and reductive assignment versions of the binary logical operators (`&&=` and `||=`) have been added to perform the SIMD ANY and ALL operations for assignments and reductions.

- The vector element shift (`[<<n]` and `[>>n]`) and rotate (`[<<%n]` and `[>>%n]`) operators have been added to ease the implementation of inter-element communication and similar algorithms.

- The `typeof` operator returns the type of its expression argument. This allows parameterized functions to be written to handle many types.

- The `widthof` operator returns the declared dimension of its expression argument.

- The `precisionof` operator returns the declared precision of its expression argument.

4.2.4 An Example Function

An example of code that can be written in SWARC is the Linpack benchmark DAXPY loop, which is actually performed as a SAXPY (Single-precision AXPY) on most SWAR hardware. A C version of the original loop looks like this:

```
for (i = 0;i < n; i++)
        dy[i] = dy[i] + da*dx[i];
```

In SWARC, the same code is written as a vector expression. Here, we show the code wrapped in a function body which can be in-lined or copied directly into the SWARC source:

```
void swar_saxpy(float:32[VECTSIZE] x, float:32[VECTSIZE] y, float s)
{
        y += (s * x);
}
```

Note that the algorithm is expressed as operations on vectors much as it would be in mathematical notation. Thus, this SWARC code is more natural than the looped C code. Also, this code describes the data and operations to be performed without exposing the structure of the target architecture. Thus, it is portable across multiple architectures. This allows users to write portable SIMD functions that can be compiled into efficient SWAR-based modules and interface code which allows these modules to be used within ordinary C programs.

4.3 The Scc Compiler

The experimental SWARC module compiler, Scc, is the first implementation of a compiler for a general-purpose SWAR language. Scc is a cross-compiler which targets several SWAR-capable architectures. These include the Intel IA32 architecture using standard C code and MMX, 3DNow!, Enhanced 3DNow!, and AltiVec architectures using C code with inlined assembly macros which make use of these extensions.

Scc is intended to be not only a proof-of-concept implementation of the SWAR model, but also to provide a framework for further SWAR research. To this end, the source code for Scc will be placed into the public domain when this dissertation is deposited.

Any portable SWAR language such as SWARC must provide the programmer with a consistent programming model. Any compiler for such a language must manage the inconsistencies of the target architectures to implement this model. The compiler must provide emulation for unsupported operations and correctly implement SIMD-style control constructs.

4.3.1 Organization

The Scc compiler consists of the front end, a back end, and a set of utilities which are used throughout the compiler. The purpose of the front end is to determine what type of processing must be performed on each source file, parse SWARC source code, and convert the SWARC source into a type-coerced, optimized intermediate representation (IR) tree representing the vector operations. The back end has the task of converting the intermediate vector tree into lists of tuples representing operations on word-sized data fragments, and generating C code to implement the operations described by these tuples based on the capabilities of the target architecture.

The primary flow of data through the compiler follows a path from the main function, through the parser, the fragmenter, and finally the scheduler.

4.3.2 The Front End

The front end consists of six major functional units. These determine how each source will be handled, parse SWARC sources to form an intermediate representation (IR) tree, and perform type checking, type coercion, and vector-based optimizations such as constant-folding of vector operations.

The parser was built using PCCTS (the Purdue Compiler Construction Tool Set, see the network newsgroup `comp.compilers.tools.pccts`). As it reads the SWARC source code, it generates top-level declarations and prototypes for the C output. As each function body is parsed an IR tree is built to represent it. This tree has a child-sibling structure and contains nodes which represent scalar and vector operations. It is optionally passed to the front end optimizer before being passed to the back end for code generation.

The front-end optimizer reconfigures the IR tree for a function by performing several optimizations. These include constant folding on scalar and vector operations, removal of code to compute conditionals with constant values and the related unused conditional bodies, and aggressive vector-level algebraic simplification. These optimizations depend not only on the type of the values, but also on their precision and size.

Figure 4.1 is a representation of the IR tree that the front end generates for our SAXPY example. The notation “2x32f” indicates an entity or operation which has two fields containing 32-bit floating point values. We see that the `ADD` performs a 2x32f addition on the 2x32f value loaded from memory location y and the product of the scalar (1x32f) object a, which is cast to a 2x32f value, and the 2x32f object x. The 2x32f result is then stored in y.

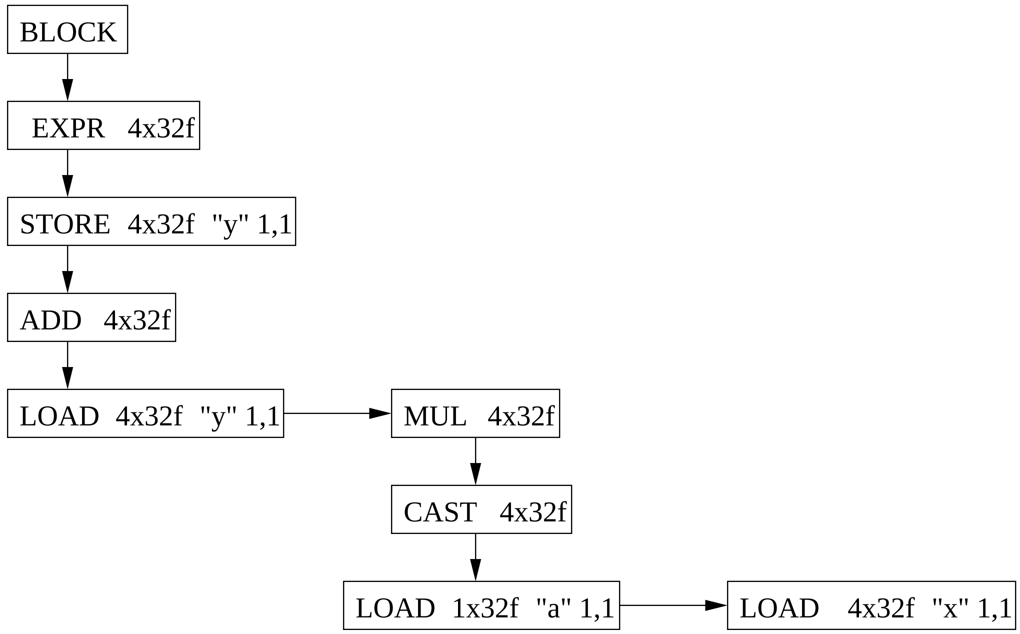

Fig. 4.1. IR tree for SWAR SAXPY

4.3.3 The Back End

The back end consists of three major functional units. These divide vector data into word-sized fragments, generate a tuple tree for each fragment, schedule the tuples, and generate output code.

Vector operations encoded in the IR tree as single tree nodes need to be converted into a series of equivalent operations on the word-length fragments of their vector arguments. This is done by the fragmenter, which converts the IR tree into lists of *tuple DAGs* (directed, acyclic graphs) which more closely represent the operations performed by hardware.

Note that fragmenting is not strip mining, although it serves a similar purpose. The primary difference is that fragmenting does not generate any loops, expensive indexing, or conditional end-of-vector tests. Instead, it generates longer sequences of fragment-based code that have the minimum possible overhead and maximum flexibility in scheduling. Future versions of Scc may use strip mining in combination

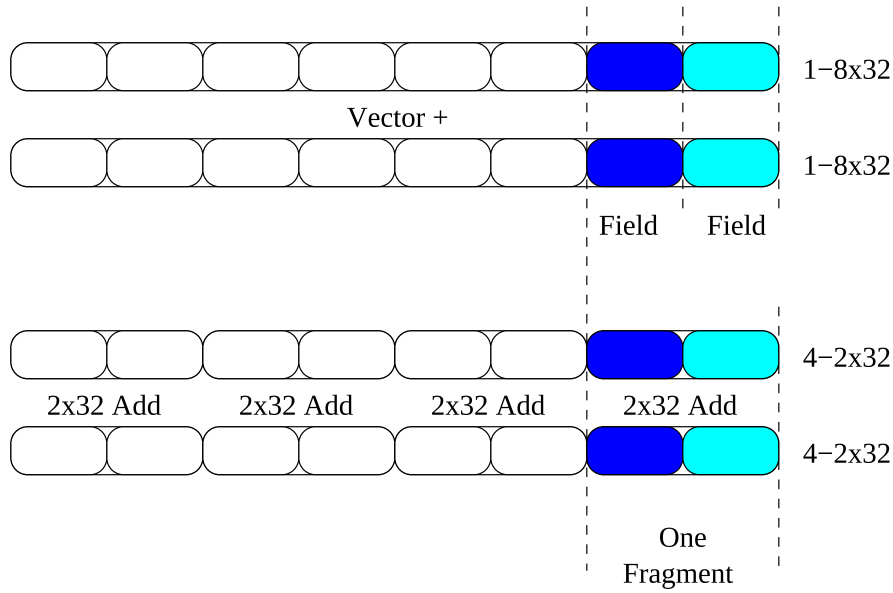

Fig. 4.2. Fragmentation of a Vector Addition

with fragmenting for very long vectors, where excessive fragmented code size might limit performance.

In figure 4.2, we see how an 8-element vector addition is fragmented into four word-sized parallel additions. In the diagram, and this discussion, the notation n-fxb indicates an entity with n parts, each of which has f fields of b bits. In the top half of the figure, a single vector addition is conceptually applied to two vectors, each of which has eight 32-bit data elements. Assuming that the target's registers have a width of 64 bits, the fragmenter can only pack two 32-bit fields into each fragment as a 2x32 SWAR entity. The lower half of the figure shows how the vector is fragmented, with each pair of elements assigned to a single fragment. The corresponding fragments of the two vectors are then added with a single hardware operation.

The operations and field sizes supported by hardware vary widely across target architectures. These differences must be accounted for during the construction of the tuple trees. Data promotion to supported field sizes and emulation of unsupported operations are performed as vector operations represented by IR tree nodes are converted to fragment operations represented by tuples. The tuple generation functions used must account for many special cases and variations while constructing the tuple DAGs.

As the tuple trees are generated, common subexpression elimination is performed by reusing previously generated, equivalent tuple trees when possible. Reduction in strength optimizations can also be performed in these functions; however, care must be taken, because these optimizations depend on the availability of an instruction with the correct data type and field width. Finally, several compiler optimizations can be applied at the fragment level during the generation of tuples to lessen the overhead of enable masking and spacer manipulation or to take advantage of the special situations created by the use of fragmentation, spacer bits, and enable masks. These optimizations will be discussed in section 4.4.

Once a tuple tree list for a basic block has been generated, the fragmenter calls the scheduler to generate output code for the list. The combined scheduler/register-allocator then performs a modified exhaustive search of the possible schedules for the tuple list based on schedule permutation [188]. A detailed model of the target pipeline is used to estimate the cost of each schedule.

The scheduler attempts to find an optimal schedule by first building an initial schedule, then trying to improve it by placing restrictions on operations such as memory accesses and relaxing these restrictions until a viable schedule can be generated.

Once a schedule for the basic block is found, output code is generated for it. This schedule is known to be optimal for the target architecture based on the pipeline cost estimation. This cost estimate takes into account emulation overhead, multiple pipeline usage, target-specific instruction costs, operand source differences, and costs related to register renaming. Unfortunately, our current cost estimation model overestimates the expected cost of memory references in certain circumstances. This causes the scheduler to choose non-optimal code sequences in certain situations.

Returning to our SAXPY example, the C code generated by Scc for the SWAR version targeting an AMD K6-2 (with four elements to keep it brief) is given below.

```
void swar_saxpy(p64_t *x, p64_t *y, float *a)
{
    register p64_t *_cpool = &(mmx_cpool[0]);
    {
        movq_m2r(*(((p64_t *) a) + 0), mm0);
        pand_m2r(*(_cpool + 2), mm0);
        movq_r2r(mm0, mm1);
        psllq_i2r(32, mm0);
        por_r2r(mm0, mm1);
        movq_r2r(mm1, mm2);
        pfmul_m2r(*(((p64_t *) x) + 1), mm1);
        pfmul_m2r(*(((p64_t *) x) + 0), mm2);
        pfadd_m2r(*(((p64_t *) y) + 1), mm1);
        pfadd_m2r(*(((p64_t *) y) + 0), mm2);
        movq_r2m(mm1, *(((p64_t *) y) + 1));
        movq_r2m(mm2, *(((p64_t *) y) + 0));
    }
_return: femms();
}

p64_t mmx_cpool[] = {
    /* 0 */      0x0000000000000000LL,
    /* 1 */      0xffffffffffffffffLL,
    /* 2 */      0x00000000ffffffffLL,
    /* 3 */      0xffffffff00000000LL
};
```

The first five statements inside the inner block load the 32-bit float value `a` into both fields of a 64-bit register. The sixth copies this value for use with another fragment. The remaining instructions perform the SAXPY on the two fragments of the vector data in `x` and `y`. Note that the above code is not optimally scheduled due the aforementioned errors in the current cost estimation code.

4.4 Implementation of Compiler Optimizations For SWAR

One goal of this research was to develop compiler optimizations which could be used to enhance code performance and alleviate the negative effects of emulation on performance.

In [106], we introduced and discussed several static compiler optimizations that apply to SWAR programming. These were based on tracking data, spacer, and mask values, and aggressively simplifying code dealing with spacers and masks. While some of these techniques can only be applied for particular types of targets and field sizes, others apply to all targets, and some can be implemented at both the vector and fragment levels.

The Scc compiler forms a framework for research on SWAR-based optimizations of vector and fragment operations and instruction scheduling for SWAR-capable targets. In this section, we discuss how these optimizations have been implemented within the framework of the Scc experimental compiler. We will briefly reintroduce these optimizations here, but refer you to [106] for a more detailed discussion. Four such optimizations are: promotion of field sizes, SWAR bitwise value tracking, simplification of spacer manipulation, and enable masking optimization.

4.4.1 Promotion Of Field Sizes

In SWARC, the application programmer may specify the minimum number of bits of precision required for a value. The compiler may choose to use more bits for storage of these values in order to make use of specialized hardware on the target architecture. For example, 16 bit values are handled very well by HP's PA-RISC MAX-2 [62], but smaller sizes are not. Thus, operations on vectors that were declared to contain 14-bit values will be converted into more efficient code sequences for MAX-2 if the vectors are internally promoted to 16-bit fields instead of being handled using 14-bit emulation.

In general, for each SWAR target, there are certain field sizes that are less efficient than some larger field size and should not be used internally. However, not all smaller field sizes are less efficient than a larger size. In some cases, the field size is small enough that the gain in parallelism outweighs the overhead of converting to an unsupported field size.

Current Scc compiler targets only directly support field sizes of 8-, 16-, 32-, and 64-bits. Data of any other field size must be promoted to one of these or any operations on it will have to be emulated by the compiler. The Scc compiler emulates 1-, 2-, and 4-bit support, as these field sizes are reasonably efficient. All other unsupported field sizes are promoted to the next larger supported or emulated field size.

Certain field sizes do not supply any added parallelism over the next larger size. This is true whenever the fragment length in fields is equal for both field sizes. In this case, it is only beneficial to emulate the smaller field size if the extra bits can be used to increase the number of spacer bits (this will be explained in section 4.4.3) between data fields. Otherwise, promoting the data to a higher precision allows the amount of emulation code necessary to be minimized without affecting the performance of the output code.

In Scc, promotion of data precision is performed in the back-end during the fragmentation phase because this is the first point in the compilation process that the parameters of the target are known. Promotion depends not only on the size of the target's registers, but also on the set of instructions available to operate on the supported field sizes.

4.4.2 Vector Algebraic Simplification and Bitwise Value Tracking

In [106], we introduced the topic of bitwise value tracking as it related to the optimization of compiler-inserted masking operations. These are primarily composed of bitwise `AND` and `OR` operations and left and right shift operations using constant-valued masks and shift distances. Consider the following example in C in which the low byte of a 16-bit word is moved into the high byte with masking:

```
x = (( (x & 0x00ff) << 8 ) & 0xff00);
```

Simple constant folding will not improve the above code because no single operation has two constant operands. However, by aggressively applying the algebraic properties of the operations involved, we can restructure the code so that constant folding can be applied. Distributing the shift over the inner expression yields:

```
x = (( (x << 8) & (0x00ff << 8) ) & 0xff00);
```

which can be folded to:

```
x = (( (x << 8) & 0xff00 ) & 0xff00);
```

From here, we see that the `AND` operations can be folded because they are associative and each has a constant operand. In this particular example, they also happen to have the same value, although this is not true generally. The code is finally converted to the equivalent, but simpler, form:

```
x = ((x << 8) & 0xff00);
```

Note that unless we are able to fold the operations at each step, we will be simply replacing one set of operations with an equal number of different operations which are probably equally expensive. A strict set of conditions must be met to make this optimization worthwhile:

- The top-level operation *op1* must have one operand which evaluates to a constant value, and another which is a tree rooted at an operation *op2*.

- *op2* must have one operand which evaluates to a constant, and a second which is a tree rooted at an operation *op3*, over which *op2* is distributive [2].

[2]Note that the distributed form of an expression is only approximately equal to the non-distributed form in finite-precision arithmetic

- *op3* must have one operand which evaluates to a constant, and be associative with *op1*. Note that *op1* and *op3* may differ. For example, in 1-bit fields, additions and exclusive-ORs are associative.

- Other restrictions may be imposed due to the exact form of the expression tree and the asymmetry of any of the operation's properties. For example, *op1* or *op3* may be required to be commutative so that operands may be reordered and associative combining of operations applied.

After ensuring that the above conditions are met, the algorithm to perform this optimization on an expression tree has four basic steps:

- Distribute *op2* over *op3*.

- Reorder the tree if necessary, depending on the commutative properties of *op1* and *op3*.

- Combine *op1* and *op3*.

- Perform constant folding on the tree.

After this last step, *op1* has been eliminated from the tree. This process can then be continued up the expression tree in the attempt to remove more operations.

In Scc, this optimization can be applied at the vector level to algebraically simplify vector operations, and at the fragment level to optimize masking and spacer operations on the tuple trees for each fragment.

4.4.3 Spacer Value Tracking and Simplification of Spacer Manipulation

Spacer bits form buffer zones between the fields of a software-partitioned register. For example, we may place three 10-bit data fields in a 32-bit register with one spacer bit, which does not contain data, placed between each pair of these fields. These bits catch carries from, and supply borrows to, the data fields of the register to keep these actions from affecting the other data fields.

The values of these bits usually need to be preset to an operation-dependent value before each operation that generates carries or borrows, otherwise they may propagate these effects to another field. After such an operation, these values may have been altered by a carry or borrow, and may need to be reinitialized to properly isolate the fields. By tracking the range of values of spacer bits between operations we can statically determine when these preset and isolation operations can be eliminated.

Also, in a series of vector operations, these spacer manipulations often result in redundant operations which can be eliminated and operations that can be optimized via bitwise value tracking. For example, consider computing `e=((a+b)-(c+d))` using a SWAR representation employing spacer bits identified by the mask `s`. The unoptimized form of the operation contains a large number of preset and isolation operations:

```
e = (((((a & ~s) + (b & ~s)) & ~s) | s) -
     ((((c & ~s) + (d & ~s)) & ~s) & ~s)) & ~s;
```

As discussed in [106], reduction of these operations can yield a significantly tighter code sequence:

```
e = (((a + b) | s) -
     ((c + d) & ~s)) & ~s;
```

For field sizes in which many operations must be emulated, the manipulation of spacer bits may be a significant fraction of all the instructions executed. Thus, optimization techniques which reduce the frequency of spacer manipulations are desirable.

A generalized, but rudimentary form of spacer manipulation is implemented in the front-end of the current Scc compiler. Currently, it should be strong enough to detect and optimize code, such as that above, in which the necessary conditions for optimization are easily checked. However, it does not currently do so because the compiler is not building identical trees for the mask loads.

Full spacer value tracking, in which the value of the spacer bits is determined and maintained for each node of the IR tree is not currently performed. Such processing would be especially useful in the back-end, where masking tuples are often generated to handle partially-filled fragments. These operations do not appear in front-end processing.

4.5 Comparison with Concurrent Work

In chapter 1, we discussed work performed at MIT's Laboratory for Computer Science concerning Superword Level Parallelism (SLP) and also the VSUIF project at the University of Toronto. In this section, we will compare the approach taken for our Scc compiler with the approaches taken by these other research groups.

Any compiler targeting multimedia-enhanced processors will have to perform each of the following steps regardless of the source language or target architecture:

- Find parallelizable code in the source.
- Convert the parallelizable code into parallelized fragment-based operations.
- Output fragment code as sequential instructions.

The primary difference between our approach and that of the University of Toronto and MIT groups is in how the first two of these steps are performed. The first of these steps is concerned with identifying parallelizable code in the source. How this is done depends on the source language and its structure.

Both the Toronto and the MIT groups make modifications to the SUIF compiler to convert sequential C source code into parallelized output. Thus, neither of these groups allows for non-standard data types. Only our approach incorporates this possibility in the programming model. This is done through a significant modification of the source language.

In our approach, the source is written as first-class vector code (which conceptually could be extended to array code). Detection of parallelism here is simple – vector operations are inherently parallelizable. Thus the first step is trivial, and the second consists of simply fragmenting the vector code and scheduling it.

The Toronto approach is essentially traditional vectorization. The compiler vectorizes loops of sequential scalar code, written in a language such as C. This is then strip-mined at the fragment level to form fragment code, with the target architecture treated as a parallel vector machine.

The SLP approach is more complex. In this approach, looped, sequential scalar code is unrolled, then the entire basic block is searched for code which can be combined into fragments. Hence, this approach builds fragments in the direction opposite to that of our approach or that of the Toronto group.

In order to find combinable statements, the SLP detection algorithm starts by finding references to adjacent memory locations which can be accessed with a single load or store instruction. For example, accesses of adjacent array elements generally can be combined unless they cross an alignment boundary on a target which cannot handle unaligned accesses. Thus, the SLP compiler tries to combine operations on data which is already allocated in packed form. This keeps the cost of packing low, but the cost of having the parallelizing compiler locate data stored in pre-packed form is high.

In our approach, data is expected to be explicitly stored as a pre-packed vector. Vector operations are thus not only known to be parallelizable, but are also known to be in pre-packed form. This makes it trivial for the compiler to recognize this type of parallelism and eliminates the costs associated with packing and unpacking native vector data. The downside is that our approach forces the programmer to store data in packed form or, if necessary, convert it by hand.

After the source has been converted to fragment-based operations, the third step is similar for each of the compilers. Optimizations such as common sub-expression elimination and constant folding are performed and some form of scheduling technique is used to schedule the sequential fragment operations.

One of the primary problems with the design of multimedia sets has been the lack of sufficient mechanisms to minimize the costs of manipulating data layouts. It is costly to pack data into the proper form for SWAR-like parallelism to be applied.

Often the speedup obtained through exploiting this parallelism is offset by the expense of packing and unpacking the data. If sufficient mechanisms are not provided for dealing with this issue, then the problem must be avoided or minimized.

Our approach to dealing with this issue is to force the data to be laid-out in packed form (as SWAR vectors) at all times. This eliminates the need to convert data layouts from unpacked to packed form and vice versa. One problem with our approach is that we allow vector code to be linked with non-vector code. This means that in certain cases, the programmer must ensure the proper data layout by generating the data in packed form or by performing packing before passing the data to vector-based functions. This is not really part of the SWAR model, but is related to the way in which it is currently supported within the SWARC language/Scc compiler framework.

The SLP group handles this issue while choosing how single-valued operations are combined to form operations on fragments. Part of the SLP combination algorithm determines if a result can be reused in packed form in a subsequent instruction. If so, it is left packed; otherwise, it is unpacked for storage.

One negative aspect of our compiler implementation concerns the size of the problem attacked. In our approach, a large amount of sequential fragment code may be generated when a vector operation on long vectors is fragmented. This is represented by a large graph in memory during compilation. Scheduling the fragment operations represented by this graph is both time and space intensive and may take several minutes to compile a fairly small benchmark.

The MIT group faces a similar problem, which is exacerbated by the fact that their approach is to unroll any loops, then coalesce single-valued operations into fragment-based operations. Thus, their compiler generates even larger internal representations embodying the individual statements before packing them into fragment-based operations. Packing and scheduling these operations is also time-intensive, and since the problem set is larger (single-valued operations versus fragment operations) it is more time-consuming than our approach.

These problems are avoided by the Toronto group's strip-mining approach because loops in the source code are converted directly into loops in the output code without unrolling or fragmentation. This keeps the problem size small and minimizes the time required to generate output code. However, it lessens the possibility of interloop optimization, especially in regards to masking and emulation code.

5. EVALUATION OF GENERAL-PURPOSE SWAR MODEL AND IMPLEMENTATIONS

To ensure that the new SWAR model is a viable replacement for current parallel programming models and that it allows programmers to exploit the SWAR technology of various COTS processors, it is necessary to analyze the performance of an incarnation of the model. The SWARC language developed in the previous chapter is one such incarnation. This language was implemented using the Scc compiler which is also described in the previous chapter.

By studying the coding of various benchmarks and applications, we can determine if the model is portable and complete. By studying their performance, we can determine if performance gains are possible and develop an intuition about the type of performance gains that can be expected.

The goal of the final phase of this research was to develop and employ metrics to examine the measurable effects of SWAR-based technology. In this chapter, I will discuss a set of benchmark programs that have been used to evaluate the SWAR model, the SWARC language, and the Scc compiler. These include a brute-force test of arithmetic expression handling, an algorithm for increasing the resolution of LCD panels, an algorithm which mimics DNA subsequence searches using non-standard precision data, and a version of the Linpack benchmark modified with modular SWARC code.

5.1 An Integer Expression Validation Program

The "valid" program is used to ensure that the Scc compiler generates proper code for the majority of binary operations allowed in the SWARC language. This program was the primary means of testing the compilation of mathematical, logical, and conditional expressions, and gives a good indication of how well the compiler has been ported to a given target.

Basic arithmetic operations tested are: addition, subtraction, multiplication, division, modulus, minimum, and maximum. Logical AND and OR are also included, as are the bitwise logical AND, OR, and XOR operations. The bitwise logical combination AND-NOT is also tested, primarily because this validation program was originally developed for an MMX target. The equality, inequality, less-than, less-than-or-equal, greater-than, and greater-than-or-equal comparisons are also included. Bit shift instructions are not. Each of the operations tested is done so for both modular and saturated data.

For each operation, a test is conducted which compares the results of Scc-compiled SWARC code and C code compiled by the native C compiler for every possible element value replicated throughout the register. That is, given a particular data precision, for every representable m and n, the operation is applied to one vector that consists of elements which each have the value m while the other has elements which each have the value n.

Currently, this validation program tests operations on vector fragments consisting of signed or unsigned integer elements with power-of-two data precisions up to 32 bits. This test flags any discrepancies in calculation as compared against the C version of the same operation. The causes of these discrepancies can then be studied and action taken to correct errors.

By default, element precisions of up to 8 bits are tested when the program is run because this type of exhaustive testing can usually be done quickly for these smaller precisions. Exhaustive testing for larger precisions takes significant time (on the order of days or centuries) to test on current hardware. To allow useful testing to be done in reasonable time, 16-bit tests are limited to one type of data (unsigned modular, unsigned saturated, signed modular, or signed modular) per run. In addition, both operand values can be strided in non-unit intervals for tests on 32-bit data elements.

The sources for this program are not included in this dissertation, but are part of the Scc compiler distribution. Some sections are included in appendix E.

This program has been successfully ported to several target architectures including AMD K6-II and Athlon systems using 3DNow! , Intel Pentium and Pentium 4 systems using MMX, and a Motorola 7400 system using AltiVec. It has even been ported to an unenhanced Pentium laptop computer by generating standard C code for the target. This shows that vector arithmetic expressions can be properly described, compiled, and ported to various enhanced and unenhanced target architectures.

5.2 An Integer Benchmark — Subpixel Rendering

One benchmark test was conducted by Professor Dietz and others in a classroom setting in 1999. The purpose of this informal test was to determine what, if any, performance gains could be obtained for Scc-generated SWARC code versus optimized serial C code and hand-generated SWAR code.

Color Liquid Crystal Display units are commonly found on laptop computers and are becoming more prevalent for desktop and television systems. Each pixel of one of these displays actually consists of a set of three monochromatic "subpixels" of different colors: red, green, and blue. These are usually arranged as vertical stripes that have 1/3 the width of the full pixel. By using these subpixels to triple the horizontal resolution used, the quality of the displayed image can be significantly improved [5].

Unfortunately, treating subpixels like full pixels results in color fringing. To remedy this, a 5-point software filter was used which applies 1/9, 2/9, 3/9, 2/9, 1/9 weightings to the linear set of subpixels surrounding each subpixel on the display. While this matches well with the SWAR vector model, the filter is relatively expensive due to odd weightings and because the memory reference pattern for subpixels has a non-unit stride of three bytes.

An optimized serial C version of this filter has been in use with the PAPERS video wall library for some time. The problem of coding this filter was also assigned to 16 individual students as part of a SWARC project in the “Programming Parallel Machines” course (Spring 1999 in Purdue’s School of Electrical and Computer Engineering). Students could write their own MMX code by hand or they could write SWARC code then use the Scc compiler to generate C code which could by hand-tuned or an executable which was ready to run.

At least a few of the students achieved more than 5x speedup over the optimized C code using Scc-generated MMX code. While some students wrote their own MMX code by hand, the fastest version used unedited Scc-generated code.

This benchmark showed that the SWAR model could be used to describe a useful parallel algorithm, that this could be coded using the SWARC language, and that the Scc compiler could be used to generate parallel MMX integer code for standard precision data and achieve significant performance gains for this algorithm over optimized serial code.

5.3 An Integer Emulation Benchmark — Gene Matching

A third benchmark program operates on integer data of a non-standard precision. This forces the compiler to emulate unsupported operations on all current multimedia-enhanced architectures. The benchmark can thus be used as a test of the Scc compiler’s ability to generate correct emulation code. This program, dna.Sc, mimics a series of searches for a particular sequence of nucleotides in longer chains of DNA.

The SWAR model and the SWARC language allow a natural description of these entities and the algorithms which manipulate them. Each DNA chain is represented by a fixed-length (350-element) vector of 2-bit pseudo-random data which represents the four possible nucleotides at each position in the chain. Similarly, the target sequence is represented by a shorter, fixed-length (3-element) vector which also consists of 2-bit data.

These data objects match the physical entities which they describe more precisely than do the objects one would be forced to use under other programming models. This allows the compiler to generate better output code. For example, describing these entities as vectors of 2-bit objects allows the maximum amount of parallelism to be exploited during execution. It also allows the programmer to avoid structural overhead, such as looping constructs, required by non-vector models.

The core of this program was written in the SWARC language and is shown in appendix F. It was ported via the experimental Scc module compiler to various multimedia-enhanced target architectures and even to targets which do not directly support any form of SWAR parallelism. While this program contains some non-portable sections, they are entirely restricted to the C interface code.

The program was compiled and run on several platforms including a 166MHz Pentium-based laptop computer with no multimedia support, a 300MHZ K6-2 desktop system with 3DNow!, a 1.5GHz Pentium 4 system with and without using MMX, and a 500MHz PowerBook G4 with and without using AltiVec.

This benchmark proves that the SWARC language can been successfully used to describe algorithms which are best suited to data of non-standard precisions. That it can be ported between a diverse set of targets proves the portability of the SWAR model and the SWARC language. Also, this benchmark shows that speedup can be obtained on various target architectures for data types which they do not directly support.

The rest of this section is a discussion of the results obtained from porting this program to various target architectures and an analysis of the problems encountered during the development of this program. For each target, the benchmark was run for Scc-generated code using 2-bit integers and employing various fragment sizes, compiler optimization levels, and optimization types. It was also run for GCC-compiled C code using 32-bit integer data and separately for C code using 8-bit character data.

5.3.1 Analysis of Results on AltiVec Target

The AltiVec code generated by Scc achieved speedup, though significantly less than one would hope given AltiVec's 128-bit registers and the 2-bit data. The optimal speedup would have been approximately 128/2 or 64x over serial 32-bit integer or 8-bit character code. The average speedup obtained over measured trials ranged from about 3.8x to about 4.6x — only 1/16 of the optimal speedup. The results of these trials are given in table G.1 in appendix G.

Correct operation of the Scc-generated AltiVec code was assumed to be verified by comparing the results with the GCC-generated C versions and finding no difference in the calculated totals. The operation of the C programs was verified by hand using smaller DNA vectors.

Note that the best speedup, 4.636x, was achieved by Scc-generated C code operating on 32-bit fragments of 2-bit data vectors in the PowerPC's general register set. While this code was incorrect (the calculated total is slightly off, probably due to incorrect handling of end fragments), it is remarkable because it does *not* use the AltiVec instruction set.

The best speedup using the AltiVec instructions was 4.567x, which is nearly as good. Given that the AltiVec registers are four times as large as the PowerPC's general registers, we would expect the 128-bit fragment AltiVec SWAR code to be about four times as fast as the 32-bit fragment SWAR integer code. It is instructive to examine why this level of performance was not achieved.

The primary problem is that loads and stores are inefficient. This is partly due to the interaction of the Scc compiler with the underlying C compiler. Scc generates variables using this compiler, which is assumed to be the GNU C compiler, GCC. GCC allows an `aligned` attribute to be associated with variables and types; however, it only applies to statically allocated objects. Thus, the alignment of automatic (i.e. local) variables and function parameters is not guaranteed, and Scc is forced to assume that they are unaligned.

AltiVec memory accesses are auto-aligning. That is, a given address is converted to the nearest aligned address before memory is accessed. Thus, aligned loads into the vector registers can be accomplished with a single instruction but unaligned loads require extra processing.

To handle an unaligned load at address *addr*, two auto-aligning loads must be executed: one which loads the 128-bit (16-byte) object starting at the aligned address below *addr*, and one which loads the 128-bit object starting at the aligned address above *addr*. These are followed by an instruction which loads an alignment index fragment which is then used in a permute instruction to rearrange the bytes as needed.

Thus, a typical load to an AltiVec register takes four times as many instructions as a load to a general register. In fact, depending on the precision of the object being loaded, the load may require up to two more instructions to place the object into the bit field that Scc considers to be field 0.

In its current incarnation, Scc simply assumes that all loads and stores, except those which access the statically allocated constant and spill pools, are unaligned. Thus, because of the way in which variables are declared and passed between functions, Scc must execute several extra steps to retrieve data in a known-to-be-aligned form.

Stores are subject to the same restrictions, but here the problem is solved by using a permute (which requires an index vector) followed by four 32-bit stores. Hence, Scc takes six instructions to perform a 128-bit store from an AltiVec register. Smaller precision stores can be implemented using fewer instructions because we can replicate the value throughout the register in one instruction, then let the following element store select the correct field. Comparing these operations to a general register store, it takes up to six times as many instructions to store an object which resides in an AltiVec register.

The situation is made worse when a 128-bit object is accessed because the two halves of the object must be swapped to place the low end of the data at the low end of the vector register. Thus, loads take up to five operations and stores up to seven.

Loads and stores of vector elements are even more complex when the element is indexed by a value in an AltiVec register. This is because AltiVec expects all the parts of an address to be in the general registers, but does not provide a means of directly moving parts which reside in a vector register to them. Thus, these parts must be stored to memory, then loaded into the general registers before they can be used as part of an address in a memory operation.

While one may argue that all addressing data should be generated in the general-purpose integer registers, we will dismiss this argument because any integer value should be usable as a vector index, regardless of which register set it is generated in. In my opinion, the failure to support addressing modes which use vector registers, or to provide instructions which allow vector data to be moved directly to the general-purpose registers, is a significant flaw in the AltiVec instruction set. However, the Scc compiler should do a better job of alleviating this problem by aggressively combining vector element memory operations.

Another problem encountered exists because the Scc compiler was originally written to target only Intel-based architectures. These allow up to two registers to be named in each instruction. Scc has not yet been fully converted to support the three- and four-address code that AltiVec allows. Thus, current Scc-generated C output is two-address code. Extra instructions are used to save register values which would be overwritten in two-address code but need not be in three-address code. This makes the Scc-generated AltiVec code both longer and slower than is necessary.

Despite these problems, these tests show that SWARC code operating on non-standard integer data types can be ported to a PowerPC G4 target using its standard integer instructions or AltiVec-enhanced instruction set. It also shows that this code can achieve significant speedup in either case.

5.3.2 Analysis of Results on MMX Target

Scc-generated MMX code did not achieved speedup in any of the tests performed on a Pentium 4 target. The speedup obtained over measured trials was between approximately 0.4x and 0.8x. These results are summarized in table G.2 in appendix G for 2-bit Scc-generated MMX code, 2-bit Scc-generated C-only code using the target's 32-bit general-purpose integer registers, GCC-generated C code using 32-bit integer data, and GCC-generated C code using 8-bit character data.

The best-case Scc code was generated without using the MMX registers, with Scc running at optimization level 0, and with Scc only performing back-end peephole optimizations. Thus, we might assume that the overhead of using the MMX-enhanced hardware was greater than the gains made. However, an inspection of the generated C code reveals that the MMX-based C code is hindered by the relatively small number of enhanced registers available. Scc's spill code is admittedly horrendous, so there is a high penalty for spills. This is the primary reason for the relatively poor performance of the MMX code.

The worst-case Scc code performed better than the worst-case GCC code. Hence, the range of performance of Scc-generated code falls within that of the GCC-generated code. Thus, the choice of data precisions and compiler switches has more effect than the choice between the Scc and GCC compilers.

Correct operation of the Scc-generated MMX code was assumed to be verified by comparing the results with the GCC-generated C versions and finding no difference in the calculated totals. Note that there is no difference in the results of the Scc-generated non-MMX code and the GCC-generated code.

These tests show that SWARC code operating on non-standard integer data types can be ported to a Pentium 4 target using its integer instruction set or MMX extensions, and that the range of performance of this code is similar to that of GCC-generated code from a C source.

5.3.3 Analysis of Results on 3DNow! Target

The Scc-generated 3DNow! code also achieved speedup when run on the K6-2 target. Again, this was significantly less than the theoretical maximum of 64/2 or 32x over serial 32-bit integer or 8-bit character code, but was more than either the AltiVec-based code on the PowerPC target or the MMX code on the Pentium 4 target.

The obtained speedup for the Scc-generated code ranged from approximately 3.9x to 5.1x. The results are summarized in table G.3 in appendix G for 2-bit Scc-generated 3DNow! code, 2-bit Scc-generated C-only code using the target's 32-bit general-purpose registers, GCC-generated C code using 32-bit integers, and GCC-generated C code using 8-bit characters.

As with the MMX target, correct operation of the Scc-generated 3DNow! code was assumed to be verified by comparing the results with the GCC-generated C versions and finding no difference in the calculated totals.

The 3DNow! code suffers from the same problems as the MMX code in relation to register spills. Interestingly though, the 3DNow! trials all obtained speedup over the best GCC-generated C code. This is a significant difference in two relatively similar architectures. The reason for this needs to be studied, but may include the use of 3DNow!'s `femms` instruction which is intended to be a faster version of the MMX `emms` instruction, or an architectural issue such as the number of available pipelines for the given code sequence or the design of the memory hierarchy. It may also be due to differences in the GCC-generated C code for the different targets.

These tests show that SWARC code operating on non-standard integer data types can be ported to a K6-2 target using its standard integer instruction set or its 3DNow! extensions. It also shows that this code can achieve significant speedup in either case.

5.3.4 Analysis of Results on IA32 Target

When run on the unenhanced Pentium target, Scc-generated IA32 code achieved speedup in only one case, but not by a significant amount over the best GCC-generated C code. In the majority of cases, the Scc-generated code was actually slower. This is to be expected because the architecture does not provide any form of SWAR instructions other than the basic polymorphics (bitwise logical operations). However, this isn't the point of porting this code to an unenhanced 32-bit architecture. The important point proven here is that the SWARC code can be ported to an unenhanced architecture without modification.

The speedup for Scc-generated code ranged from approximately 0.42x to 1.03x. It is worth noting that the GCC-generated code achieved speedups ranging from 0.28x to 1.00x. Thus, the choice of compiler switches appears to affect the performance more than the choice between Scc and GCC. The results are summarized in table G.4 in appendix G for 2-bit Scc-generated C-only code using 32-bit integer fragments in the general registers, GCC-generated C code using 32-bit integers, and GCC-generated C code using 8-bit characters.

Correct operation of the Scc-generated C code was again verified by comparing the results with the GCC-generated C versions and finding no difference in the calculated totals.

These tests show that SWARC code operating on non-standard integer data types can be ported to an unenhanced IA32 target using its general registers and integer instruction set. It also shows that this code can achieve performance similar to that of standard C code.

5.4 A Floating-Point Benchmark — Linpack

As a benchmark for floating-point performance, a version of the Linpack benchmark used for ranking a wide range of machines for the Top 500 Supercomputers list [189] was modified using Scc-generated code. This was run on several systems and compared with the standard C version for single-precision data.

In one test using a 400MHz AMD K6-2 platform [5], the standard C version achieved 54 Mflops. The modified version included a few lines of the core DAXPY, DDOT, and DSCAL loops which were rewritten using hand-inlined Scc-generate 3DNow! code. Using this code, the performance increased to approximately 90 Mflops. While a significant improvement, performance was hindered by the Scc scheduler's conservative estimations of load cost which was previously discussed. A hand-tuned version of the Scc-generated 3DNow! code schedule achieved more than 220 Mflops.

In more recent testing, a C version of Linpack was modified to conditionally call SWARC code compiled by Scc for the DAXPY, DDOT, and DSCAL loops. This was constructed as two sources: one in SWARC, the other in C which were compiled and combined by the Scc compiler (no hand coding). The SWARC source is presented in appendix H. This was compiled for various fixed subvector lengths and maximum optimization times. Results of the trial runs for this set of tests are also presented in appendix H.

These tests were conducted on a 1GHz AMD Athlon-based system, with and without using 3DNow!, and on a 500MHz PowerPC G4-based system, with and without using AltiVec. Significant improvement of between 51.9% and 105% was achieved for the 3DNow! target, taking performance from the 250–270 Mflops range to the 407–616 Mflops range. Performance on the AltiVec target was, however, disappointing. It never reached the level of the corresponding C code compiled by the native C compiler. In fact, there was between a 7.6% and 8.9% decrease in performance from around 176 Mflops to between 160 and 167 Mflops for the best Scc-generated code. The worst Scc-generated code was near 50 Mflops — around a 70% decrease. This degradation is most likely due to the poor handling of memory accesses both by the AltiVec target and by the Scc compiler as discussed in the section on the dna benchmark (section 5.3).

This benchmark showed that the SWAR model could be applied to standard high-performance computing algorithms, that the SWARC language could be used to describe portable code modules for operating on single-precision floating-point data, and that these modules could be translated by the Scc compiler into 3DNow!- or AltiVec-based parallel floating-point code. It also shows that the Scc-generated code can achieve significant speedup over GCC-generated code for the 3DNow! target. It also highlights the weaknesses of the AltiVec target and the current Scc compiler with regards to the handling of memory accesses.

6. CONCLUSION

In this thesis, a new, abstract model of parallel computation was developed which better reflects the capabilities and limitations of modern SWAR (SIMD Within A Register) architectures than did previously-defined computational models.

A summary of the support provided by various multimedia extension sets for general-purpose SWAR processing was compiled (section 2) and presented as a set of tables describing the type of SWAR operations supported by each of these families (section 2.1 and appendix C) with an accompanying analysis of their capabilities (section 2.2).

These capabilities were shown to vary significantly, with some extensions offering little support for SWAR processing, having only a few SIMD instructions, while others offered significantly better support with larger, more complete repertoires. Commonly-supported operations were identified, and the suitability of the various types of operations which these extensions perform was considered in terms of inclusion in a general-purpose SWAR programming model.

This work formed a basis for the design of the new, general-purpose SWAR programming model developed in this research (section 3.3) and hereby placed in the public domain. This programming model allows general-purpose applications programmers to exploit vector SIMD parallelism when targeting SWAR-capable commodity off-the-shelf (COTS) processors in a portable, target-independent manner.

This model more closely reflects the capabilities and limitations of current SWAR processors than did previously-defined models by allowing for commonly-supported operations such as saturation addition while discouraging esoteric operations such as full permutations and less efficient operations such as complex communications and multi-dimensional array operations.

A list of properties that a well-designed, full-scale, high-level language for SWAR should exhibit (section 3.4) was enumerated. This formed the basis for the development of prototype implementations of the model. These guidelines can also be used by others who wish to develop languages based on the general-purpose SWAR processing model.

Prototype implementations of the SWAR model were developed and presented including various libraries (section 4.1) and the SWARC modular programming language (section 4.2) which provides a portable, target-independent language for expressing data parallel applications in terms of vector processing. These implementations show that the SWAR programming model is viable and can be implemented in various forms.

The Scc compiler for the SWARC language (section 4.3) was enhanced through the development of various techniques for emulating unsupported operations, for exploiting the advanced features of various targets, and for optimizing SWAR-based target code. These advancements allow code to be generated for a variety of multimedia-enhanced architectures and even unenhanced processors. The current version of the Scc compiler is hereby placed in the public domain.

Various metrics were also developed and applied to evaluate the portability, completeness, and performance of the SWARC language and Scc compiler. These took the form of SWARC programs and include:

1. A validation program to thoroughly test the correctness of Scc-generated code for the majority of binary operations allowed in SWARC. This is limited to power-of-two data precisions through 32-bits, but includes both signed and unsigned and both modular and saturated data types.

2. A program to test the portability and performance of code which operates on non-standard precision integer data.

3. A version of the standard Linpack benchmark to test the single-precision floating-point performance of Scc-generated code on various platforms which support floating-point SWAR operations.

4. Various other programs developed by me or by others.

These metrics show that the SWAR model is viable as exhibited by its implementation as the SWARC vector processing language. Specifically, they show that:

1. The SWARC language allows general-purpose integer and floating-point vector applications to be described in a consistent, natural, and portable manner.

2. SWARC applications may use standard precision floating-point data or integer data of standard or non-standard precisions, including those which are not supported directly by the target architecture.

3. SWARC applications can include scientific and high-performance algorithms as well as multimedia algorithms.

4. SWARC code can be, and has been, ported to various multimedia-enhanced and unenhanced architectures.

These metrics also show that the Scc optimizing compiler for the SWARC language is viable and capable of generating highly efficient code, although it has been found to be lacking in certain respects. Specifically, these metrics show that:

1. The Scc compiler can generate output which exploits the multimedia enhancements of various targets to achieve performance gains.

2. The Scc compiler can generate standard C code output which can be ported to various unenhanced processors.

3. Significant speedup can be achieved for integer and floating-point applications.

4. Significant speedup can be achieved, or significant degradation avoided, for applications which require the emulation of operations on non-standard precision integer data.

5. The Scc compiler's interaction with the underlying C compiler has implications in regards to the layout of data in memory which can have a significant negative impact on performance.

6. The fixed-vector size required by the Scc compiler is a liability, albeit one that can be easily addressed using known techniques.

7. The fragmentation of large vectors, as opposed to strip-mining them, can have a significant effect on the size of code and can negatively impact the compiler's ability to generate efficient code.

In summary, the general-purpose SWAR processing model developed in this thesis is a new, abstract model of parallel computation which better reflects the capabilities and limitations of modern SWAR architectures than did previously-defined computational models and allows programmers to exploit the capabilities of current SWAR architectures in a portable and consistent manner.

6.1 Future Research

This work provides a starting point for future research and the development of practical programming languages for SWAR processing. Future research may include:

1. Extension of the model to array-based SWAR architectures when they become commonplace. Current commodity SWAR processors are primarily based on one-dimensional vector parallel architectures. Future COTS processors will likely be based on multi-dimensional array parallel architectures. This will require consideration of certain aspects of SIMD processing which have been safely ignored in the current work.

2. Refinement of the SWARC language as SWAR architectures evolve. The set of operations which a typical multimedia enhanced architecture supports can be expected to grow as this paradigm matures and new architectures are developed. Certain operations will become more common while others will be orphaned.

3. Continued development of new emulation techniques for unsupported SWAR operations. Portability depends greatly on the ability to emulate operations which are unsupported by the target architecture. Further research will be necessary to increase the range of viable targets and the repertoire of viable operations.

4. Development of new languages based on the general-purpose SWAR model. These may include application-specific languages or languages which denote parallel data or operations in a manner which differs from current SWAR languages.

LIST OF REFERENCES

[1] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[2] Bowen Alpern, Larry Carter, and Kang Su Gatlin. Microparallelism and high-performance protein matching. In *Proceedings of Supercomputing '95*, pages 536–551, San Diego, California, November 1995.

[3] Hank Dietz. Technical summary: SWAR technology. Technical report, School of Electrical and Computer Engineering, Purdue University, February 1997. http://dynamo.ecn.purdue.edu/~hankd/SWAR/over.html.

[4] Randall Fisher. General-purpose SIMD Within A Register: Parallel processing on consumer microprocessors. Technical report, School of Electrical and Computer Engineering, Purdue University, Ph.D. Thesis Proposal, November 1997.

[5] Randall J. Fisher and Henry G. Dietz. The Scc Compiler: SWARing at MMX and 3DNow! In Larry Carter and Jeanne Ferrante, editors, *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, La Jolla, California, August 1999. Springer-Verlag.

[6] Mark Richard Spieth. *Single Processor Multiple Data Parallel Processing.* PhD thesis, Royal Melbourne Institute of Technology, Melbourne, Australia, Date unknown.

[7] Intel Corporation. MMX technology tools providers: Intel corporation. Technical report, Intel Corporation, http://developer.intel.com/design/perftool/vtune/, March 1997.

[8] Intel Corporation. MMX technology tools providers: NuMega technologies. Technical report, Intel Corporation, http://developer.intel.com/drg/tools/numega.ht, March 1997.

[9] Intel Corporation. MMX technology tools providers: Intel corporation. Technical report, Intel Corporation, http://developer.intel.com/design/perftool/perflibst/spl/, April 1997.

[10] Intel Corporation. Intel Image Processing Library. Technical report, Intel Corporation, http://developer.intel.com/design/perftool/perflibst/ipl/, May 1997.

[11] Intel Corporation. MMX technology tools providers: Intel corporation. Technical report, Intel Corporation, April 1997. http://developer.intel.com/design/perftool/perflibst/rpl/.

[12] Intel Corporation. MMX(TM) Technology Tools Providers: IBM Corporation. Technical report, Intel Corporation, http://developer.intel.com/drg/tools/ibm.htm, March 1997.

[13] Apple Computer, Inc. Frequently asked questions: Apple velocity engine. Technical report, Apple Computer, Inc., 2001. http://www.apple.com/scitech/physicalscience/VE102501.pdf.

[14] Sun Microsystems, Inc. mediaLib, Sun Microelectronics. Technical report, Sun Microsystems Corporation, http://www.sun.com/sparc/vis/mediaLib.html, March 1997.

[15] Sun Microsystems, Inc. mediaLib Object Code License. Technical report, Sun Microsystems Corporation, http://www.sun.com/sparc/vis/download/mlib/, March 1997.

[16] Iain Nicholson. libSIMD – the Single Instruction Multiple Data Library. Technical report, Hosted by SourceFORGE.net, February 2000. http://libsimd.sourceforge.net.

[17] Intel Corporation. Intel Fortran Compiler for Windows. Technical report, Intel Corporation, http://www.intel.com/software/products/compilers/f60/fwindows.htm, April 2002.

[18] Intel Corporation. Intel C++ Compiler for Windows. Technical report, Intel Corporation, http://www.intel.com/software/products/compilers/c60/cwindows.htm, April 2002.

[19] Intel Corporation. MMX technology tools providers: Microsoft corporation. Technical report, Intel Corporation, http://developer.intel.com/drg/tools/ms_vc.htm, April 1997.

[20] Metrowerks, Inc. Optimizing code for AMD-K6. Technical report, Metrowerks, Inc., October 1998. http://www.metrowerks.com/pdf/Optimizing_Code_for_AMD_K6.pdf.

[21] Metrowerks, Inc. Support for AMD's 3DNow! technology. Technical report, Metrowerks, Inc., 2002. http://www.metrowerks.com/desktop/windows/amd/.

[22] Metrowerks, Inc. CodeWarrior for Mac OS, Profession Edition 8.0 features. Technical report, Metrowerks, Inc., 2002. http://www.metrowerks.com/products/macos/?features.

[23] Q Software Solutions, GmbH. Q Software Solutions. Technical report, Q Software Solutions, 1999. http://www.q-software-solutions.com/lccwin32/index.shtml.

[24] Christopher W. Fraser and David R. Hansen. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley Publishing Company, Redwood City, California, 1995.

[25] codeplay, Ltd. Codeplay: Vectorc overview. Technical report, codeplay, Ltd., 2002. http://www.codeplay.com/vectorc/index.html.

[26] Green Hills Software, Inc. Embedded software development tools – powerpc family. Technical report, Green Hills Software, Inc., 2002. http://www.ghs.com/products/PowerPC_development.html.

[27] Sun Microsystems, Inc. 3/29/96 - sun launches the VIS software developer's kit. Technical report, Sun Microsystems Corporation, http://www.sun.com/smi/Press/sunflash/9603/sunflash.960329.5709.html, March 1996.

[28] Sun Microsystems, Inc. VSDK License. Technical report, Sun Microsystems Corporation, http://www.sun.com/sparc/vis/download/vsdk/, April 1997.

[29] Ruby B. Lee and Michael D. Smith. Media processing: A new design target. *IEEE Micro*, 16(4):6–9, August 1996. Guest Editors' Introduction to Special Issue on Media Processing.

[30] Free Pascal Web Team. Free Pascal - home page. Technical report, Free Pascal Organization, November 1997. http://www.freepascal.org.

[31] Michael Van Canneyt and Florian Klampfl. Free Pascal supplied units: Reference guide. Technical report, Free Pascal Organization, December 1998. http://rs1.szif.hu/˜marton/fpc/units.

[32] Oxford Micro Devices, Inc. New method for programming Intel multimedia extensions (MMX). Technical report, Oxford Micro Devices, Inc., 1996. http://oxfordmicrodevices.com/pr040396.html.

[33] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Inc., Boston, Massachusetts, June 1996. ISBN 1-882114-66-3.

[34] Joe Wolf. Advanced optimization with the Intel C/C++ compiler. Technical report, Intel Corporation, 1999. Formerly at http://developer.intel.com/vtune/newsletr/opts.htm.

[35] codeplay, Ltd. Codeplay: Vectorc vectorization. Technical report, codeplay, Ltd., 2002. http://www.codeplay.com/vectorc/feat_vec.html.

[36] The Portland Group, Inc. The portland group – PGF77 Workstation. Technical report, The Portland Group, Inc., 2000. http://www.pgroup.com/prodworkpgf77.htm.

[37] Doug Miles. PGI Workstation, parallel F90/C/C++ compilers and tools for Intel processor-based workstations, servers and clusters. Technical report, The Portland Group, Inc., 2000. www.ahpcc.unm.edu/projects/vista-azul/ACTC_April_00/pgi.pdf.

[38] Veridian Systems. Vast/parallel – Fortran and C automatic parallelizing preprocessors. Technical report, Veridian Systems, January 2002. http://www.psrv.com/vast_parallel.html.

[39] Robert N. Braswell and Malcolm S. Keech. An evaluation of Vector Fortran 200 generated by Cyber 205 and ETA-10 pre-compilation tools. In *Proceedings of Supercomputing '88 [Vol.1]*, pages 106–113, Orlando, Florida, November 1988.

[40] Veridian Systems. VAST-F/AltiVec. Technical report, Veridian Systems, September 2001. http://www.psrv.com/altivecf.html.

[41] Veridian Systems. VAST/AltiVec. Technical report, Veridian Systems, June 2001. http://www.psrv.com/altivec.html.

[42] Absoft Corporation. Pro Fortran for Mac OS 9. Technical report, Absoft Corporation, 2002. http://www.absoft.com/newproductpage.html.

[43] Absoft Corporation. Pro Fortran for PPC/Linux. Technical report, Absoft Corporation, 2002. http://www.absoft.com/newppcproductpage.html.

[44] Derek DeVries and Corinna G. Lee. A vectorizing SUIF compiler. In *Proceedings of the First SUIF Compiler Workshop*, pages 59–67, January 1996. http://www.eecg.toronto.edu/~devrier/done.ps.

[45] R. Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 29, pages 31–37. ACM, 1994. http://suif.stanford.edu/suif/suif.html.

[46] Krste Asanovic. *The Torrent Architecture Manual*. University of California, Berkeley, 1994.

[47] John Wawrzynek, Krste Asanovic, Brian Kingsbury, James Beck, David Johnson, and Nelson Morgan. Spert-II: A vector microprocessor system. *IEEE Computer*, 29(3):79–86, March 1996.

[48] Todd Mowry and Antonia Zhai. Scalar optimization & code generation development at the University of Toronto. In *Proceedings of the First SUIF Compiler Workshop*, January 1996. http://www-suif.stanford.edu/suifconf/suifconf1/papers/paper22.ps.

[49] Derek DeVries. A vectorizing SUIF compiler: Implementation and performance. Master's thesis, University of Toronto, June 1997. http://www.eecg.toronto.edu/~devrier/paper.ps.

[50] Massachusetts Institute of Technology. Superword level parallelism. Technical report, Massachusetts Institute of Technology, 2000. http://www.cag.lcs.mit.edu/slp/.

[51] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. Technical report, Massachusetts Institute of Technology Laboratory for Computer Science, MIT/LCS Technical Memo LCS-TM-601, November 1999.

[52] Samuel Larsen. Exploiting superword level parallelism with multimedia instruction sets. Master's thesis, Massachusetts Institute of Technology, May 2000.

[53] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, Vancouver, British Columbia, June 2000.

[54] Rainer Leupers. Code selection for media processors with SIMD instructions. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE 2000)*, Paris, France, March 2000. www.sigda.acm.org/Archives/-ProceedingArchives/Date/Date2000/papers/2000/date00/pdffiles/01a_1.pdf.

[55] Rainer Leupers and Steven Bashford. Graph-based code selection techniques for embedded processors. *ACM Transactions on Design Automation of Electronic Systems*, 5(4):1901–1909, October 2000. http://ls12-www.cs.uni-dortmund.de/publications/papers/2000-todaes.ps.gz.

[56] Paul Cockshott. Vector Pascal. Technical report, Department of Computer Science, University of Glasgow, Semptember 2001. http://www.faraday.gla.ac.uk/papers/vp-msp3.pdf.

[57] Paul Cockshott. Direct compilation of high level languages for multi-media instruction-sets. Technical report, Department of Computer Science, University of Glasgow, November 2000. http://www.dcs.gla.ac.uk/˜wpc/reports/ilcg/mmxcomp.pdf.

[58] Mike Kelley and Matt Postiff. Survey and implementation of limited SIMD instruction set architecture extensions. Technical report, University of Michigan, Ann Arbor, Michigan, December 1996. http://www.eecs.umich.edu/˜postiffm/papers/598report.ps.

[59] Pradeep K. Dubey. Architectural alternatives for mediaprocessing. Technical report, International Business Machines, July 1997. http://www.research.ibm.com/people/p/pradeep.

[60] Compaq Computer Corporation. *Alpha Architecture Handbook, Version 4*. Digital Equipment Corporation, October 1998. http://ftp.digital.com/pub/Digital/info/semiconductor/literature/% -alphaahb.pdf.

[61] Ruby B. Lee. Accelerating multimedia with enhanced microprocessors. *IEEE Micro*, 15(2):22–32, April 1995.

[62] Ruby Lee and Jerry Huck. 64-bit and multimedia extensions for the PA-RISC 2.0 architecture. In *Proceedings of Compcon '96, Technologies for the Information Superhighway, Digest of Papers*, pages 152–160, Los Alamitos, California, 1996. IEEE Computer Society Press.

[63] Ruby B. Lee. Subword parallelism with MAX-2. *IEEE Micro*, 16(4):51–59, August 1996.

[64] Silicon Graphics, Inc. MIPS V instruction set. Formerly available via WWW, 1996.

[65] Silicon Graphics, Inc. MIPS digital media extension. Formerly available via WWW, 1996.

[66] MIPS Technologies, Inc. *MIPS Extension for Digital Media with 3D*. MIPS Technologies, Inc., March 1997. Formerly at http://www.mips.com/Documentation/isa5_tech_brf.pdf.

[67] Sam Fuller. Motorola's AltiVec technology. Technical report, Motorola, Inc., Austin, California, 1998. http://www.mot.com/SPS/PowerPC/teksupport/-teklibrary/papers/altivec_wp.pdf.

[68] Motorola, Inc. *AltiVec Technology Programming Environments Manual, Rev. 1.0.* Motorola, Inc., Denver, Colorado, February 2001. http://e-www.motorola.com/brdata/PDFDB/MICROPROCESSORS/32_BIT/-POWERPC/ALTIVEC/ALTIVECPEM.pdf.

[69] Sun Microsystems, Inc. Ultrasparc the visual instruction set (VIS): On chip support for new-media processing. Technical report, Sun Microsystems Corporation, http://www.sun.com/sparc/whitepapers/wp95-022/, 1996.

[70] L. Kohn, G. Maturana, M. Tremblay, A. Prabhu, and G. Zyner. The Visual Instruction Set (VIS) in UltraSPARC. In *Proceedings of Compcon '95, Technologies for the Information Superhighway, Digest of Papers*, pages 462–469, San Francisco, March 1995. IEEE Computer Society Press.

[71] Intel Corporation. MMX technology overview. Technical report, Intel Corporation, February 1997. Formerly at http://developer.intel.com/drg/mmx/.

[72] Intel Corporation. Intel architecture MMX technology: Programmer's reference manual. Technical report, Intel Corporation, http://developer.intel.com/drg/mmx/Manuals/prm/prm_covr.htm, March 1996.

[73] Advanced Micro Devices, Inc. *AMD-K6 Processor Multimedia Extensions, Rev. D.* Advanced Micro Devices, Inc., Sunnyvale, California, January 2000. http://www.amd.com/K6/k6docs/pdf/20726.pdf.

[74] Cyrix Corporation. Multimedia instruction set extensions for a sixth-generation x86 processor. Technical report, Cyrix Corporation, August 1996. ftp://ftp.cyrix.com/developr/hc-mmx4.pdf.

[75] Advanced Micro Devices, Inc. *3DNow! Technology Manual, Rev. G.* Advanced Micro Devices, Inc., Sunnyvale, California, March 2000. http://www.amd.com/K6/k6docs/pdf/21928.pdf.

[76] Advanced Micro Devices, Inc. *AMD Extensions to the 3DNow! and MMX Instruction Sets Manual, Rev. D.* Advanced Micro Devices, Inc., Sunnyvale, California, March 2000. http://www.amd.com/products/cpg/athlon/techdocs/pdf/22466.pdf.

[77] Cyrix Corporation. *Application Note 108: Cyrix Extended MMX Instruction Set.* Cyrix Corporation.

[78] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual: Vol. 1 Basic Architecture.* Intel Corporation, 2001. http://developer.intel.com/design/pentium4/manuals/245470.htm.

[79] Paul Rubinfeld, Bob Rose, and Michael McCallig. Motion Video Instruction extensions for Alpha. Technical report, Digital Equipment Corporation, http://www.digital.com/alphaoem/papers/pmvi.pdf, October 1996.

[80] P. Knebel, B. Arnold, M. Bass, W. Kever, J. D. Lamb, R. B. Lee, P. L. Perez, S. Undy, and W. Walker. HP's PA7100LC: A low-cost superscalar PA-RISC processor. In *Proceedings of Compcon Spring '93, Digest of Papers*, pages 441–447. IEEE Computer Society Press, 1993.

[81] Ruby Lee and Jerry Huck. HPTechnical computing - 64-bit and multimedia extensions in the PA-RISC 2.0 architecture. Technical report, Hewlett-Packard Company, June 1996. http://www.hp.com/ahp/framed/-technology/micropro/architecture/docs/pa2go3.html.

[82] Gerry Kane. *PA-RISC 2.0 Architecture*. Prentice-Hall, Inc., Upper Saddle River, New Jersey 07458, 1996.

[83] D. Hunt. Advanced performance features of the 64-bit PA-8000. In *Proceedings of Compcon '95, Technologies for the Information Superhighway, Digest of Papers*, pages 123–128. IEEE Computer Society Press, March 1995.

[84] Ashok Kumar. The HP PA-8000 RISC CPU. *IEEE Micro*, 17(2):27–32, April 1997.

[85] Inc. MIPS Technologies. Silicon graphics previews new high-performance MIPS microprocessor roadmap. Technical report, MIPS Technologies, Inc., May 1997. http://www.mips.com/pressReleases/051297A.html.

[86] Inc. MIPS Technologies. Silicon graphics introduces enhanced MIPS architecture to lead the interactive digital revolution. Technical report, MIPS Technologies, Inc., October 1996. http://www.mips.com/pressReleases/100196B.html.

[87] Inc. MIPS Technologies. MIPS64 architecture product brief, 1999. http://www.mips.com/products/s2p2.html.

[88] Inc. MIPS Technologies. MIPS-3D graphics extension, 2000. http://www.mips.com/products/s2p13.html.

[89] Motorola Semiconductor Products Sector. *MPC7400 RISC Microprocessor Technical Summary*. Motorola, Inc., Denver, Colorado, August 1999. http:e-www.motorola.com/brdata/PDFDB/docs/MPC7400TS.pdf.

[90] Sun Microsystems, Inc. *UltraSPARC User's Manual*. Sun Microsystems, Inc., Palo Alto, California, July 1997. Formerly at http://www.sun.com/microelectronics/manuals/ultrasparc/802-7220-02.pdf.

[91] G. Goldman and P. Tirumalai. UltraSparc-II: The advancement of UltraComputing. In *Proceedings of Compcon '96, Technologies for the Information Superhighway, Digest of Papers*, pages 417–423, Los Alamitos, California, February 1996. IEEE Computer Society Press.

[92] Inc. Sun Microsystems. An overview of the UltraSPARC III Cu processor. Technical report, Sun Microsystems, Inc., June 2002. http://www.sun.com/processors/UltraSPARC-III/USIIICuoverview.pdf.

[93] Alex Peleg and Uri Weiser. MMX technology extension to the Intel architecture. *IEEE Micro*, 16(4):42–50, August 1996.

[94] Rise Technology Company. Welcome - Rise Technology Homepage. Technical report, Rise Technology Company, http://www.rise.com, May 2000.

[95] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual: Vol. 2 Instruction Set Reference*. Intel Corporation, 2001. http://developer.intel.com/design/pentium4/manuals/245471.htm.

[96] Intel Corporation. *Willamette Processor Software Developer's Guide.* Intel Corporation, February 2000. http://developer.intel.com/design/processor/index.htm.

[97] Advanced Micro Devices, Inc. *AMD Athlon Processor Model 4 Data Sheet, Rev. I.* Advanced Micro Devices, Inc., Sunnyvale, California, June 2001. http://www.amd.com/products/cpg/athlon/techdocs/pdf/23792.pdf.

[98] Rhett Dillingham. The new AMD Athlon processor — enhancing the world's most powerful x86 microprocessor. Technical report, Advanced Micro Devices, Inc., Sunnyvale, California, May 2001. http://www.amd.com/products/cpg/mobile/athlon/newathlonwhitepaper.pdf.

[99] Jack Huynh. *The AMD Athlon XP Processor.* Advanced Micro Devices, Inc., Sunnyvale, California, June 2002. http://www.amd.com/us-en/assets/content_type/-white_papers_and_tech_docs/26485A_Athlon_XP_white_paper.pdf.

[100] Cyrix Corporation. *Cyrix CPU Detection Guide, Preliminary Revision 1.01.* Cyrix Corporation, 1997.

[101] Inc. VIA Technologies. VIA Cyrix MII processor. Technical report, VIA Technologies, Inc., 2002. http://www.viatech.com/en/viac3/cyrix_MII.jsp.

[102] VIA Technologies, Inc. *VIA C3 Processor Datasheet.* VIA Technologies, Inc., January 2002. http://www.viatech.com/en/viac3/-VIA%20C3%20Samuel%201%20datasheet%

[103] VIA Technologies, Inc. *VIA Cyrix III Datasheet.* VIA Technologies, Inc., 2000. http://www.cyrix.com/viacyrixIIIdatasheet_1_2.pdf.

[104] Motorola, Inc. *AltiVec Technology Programming Environments Manual, Rev. 0.1.* Motorola, Inc., Denver, Colorado, November 1998. http://www.mot.com/SPS/PowerPC/teksupport/teklibrary/manuals/-altivec_pem.pdf.

[105] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual: Vol. 3 System Programming Guide.* Intel Corporation, 2001. http://developer.intel.com/design/pentium4/manuals/245472.htm.

[106] Randall J. Fisher and Henry G. Dietz. Compiling for SIMD Within A Register. In S. Chatterjee, J. F. Prins, L. Carter, J. Ferrante, Z. Li, D. Sehr, and P.-C.Yew, editors, *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing*, Chapel Hill, North Carolina, August 1998. Springer-Verlag.

[107] MasPar Computer Corporation. *MasPar Programming Language (ANSI C compatible MPL) Reference Manual, Software Version 2.2.* MasPar Computer Corporation, Sunnyvale, California, November 1991. Document Part Number 9302-0001.

[108] Craig Hansen. Architecture of a Broadband Mediaprocessor. In *Proceedings of Compcon '96, Technologies for the Information Superhighway, Digest of Papers*, pages 334–340, February 1996. http://www.microunity.com/www/broadband/library/Architecture.pdf.

[109] Analog Devices, Inc. *ADSP-21160 SHARC DSP Hardware Reference*. Analog Devices, Inc., Norwood, Massachusetts, 01062-9106, 2nd edition, March 2002. http://www.analog.com/library/dspManuals/21160_HR.html.

[110] Analog Devices, Inc. *ADSP-21161 SHARC DSP Hardware Reference*. Analog Devices, Inc., Norwood, Massachusetts, 01062-9106, May 2002. http://www.analog.com/library/dspManuals/pdf/Hardware/21161_ed3.zip.

[111] Analog Devices, Inc. *TigerSHARC DSP Hardware Specification Part ADSP-TS101S*. Analog Devices, Inc., Norwood, Massachusetts, 01062-9106, March 2002. http://www.analog.com/library/dspManuals/pdf/-tsdsp_hardware/ts_hw_sp_1_0_2.zip.

[112] Cindy Wang. Broadband signal processor suits multimedia applications. In *International IC — China, Conference Proceedings*, pages 159–173, March 2001.

[113] 3DSP Corporation. *UniPHY — Universal Physical Layer Signal Processor*. 3DSP Corporation, 2002. http://www.3dsp.com/uniphy.shtml.

[114] K. Vissers, J. T. J. van Eijndhoven, G. J. Hekstra, E. J. D. Pol, and A. K. Riemans. TriMedia CPU64, October 1999. Presented at special session at the ICCD conference.

[115] Incorporated Texas Instruments. *TMS320C80 Digital Signal Processor Data Sheet*. Texas Instruments, Incorporated, October 1997.

[116] Karl Guttag, Robert J. Grove, and Jerry R. Van Aken. A single-chip multiprocessor for multimedia: The MVP. *IEEE Computer Graphics & Applications*, 12(6):53–64, November 1992.

[117] Daniel L. Slotnick, W. Carl Borck, and Robert C. McReynolds. The SOLOMON computer. In *Proceedings of the AFIPS Fall Joint Computer Conference*, pages 97–107, Boston, Massachusetts, December 1962. American Federation of Information Processing Societies, Spartan Books.

[118] D. L. Slotnick. The conception and development of parallel processors – a personal memoir. *Annals of the History of Computing*, 4(1):20–30, January 1982.

[119] George H. Barnes, Richard M. Brown, Maso Kata, David J. Kuck, Daniel L. Slotnick, and Richard A. Stokes. The ILLIAC IV computer. *IEEE Transactions on Computers*, C-17(8):746–757, August 1968.

[120] Greg Wison et al. Parallel computing history, version 2.1. Technical report, University of Alberta, Alberta, Canada, May 1993. Submitted to comp.parallel, comp.sys.super, comp.arch, comp.compilers newsgroups.

[121] S. F. Reddaway. DAP, a distributive array processor. In *Proceedings of the first Annual Symposium on Computer Architecture*, pages 61–65, 1973.

[122] Kenneth E. Batcher. MPP - a massively parallel processor. In *1979 International Conference on Parallel Processing*, 1979.

[123] Robert J. Baron and Lee Higbie. *Computer Architecture Case Studies*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1992.

[124] John Beetem, Monty Denneau, and Don Weingarten. The GF11 supercomputer. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 108–115. IEEE Computer Society Press, 1985.

[125] Harold S. Stone. *High-Performance Computer Architecture*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1987.

[126] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1989.

[127] V. Benes. Optimal rearrangeable multistage connecting networks. *Bell System Technical Journal*, 43(4):1641–1656, July 1964.

[128] W. Daniel Hillis. *The Connection Machine*. MIT Press, Cambridge, Massachusetts, 1985.

[129] W. Daniel Hillis. The Connection Machine. *Scientific American*, 256:108–115, June 1987.

[130] Lewis W. Tucker and George G. Robertson. Architecture and applications of the Connection Machine. *IEEE Computer*, 21(8):26–38, August 1988.

[131] Thinking Machines Corporation. *Connection Machine Model CM-2 Technical Summary, Version 6.0*. Thinking Machines Corporation, Cambridge, Massachusetts, November 1990.

[132] Tom Blank. The MasPar MP-1 architecture. In *Proceedings of the 35th IEEE Computer Society International Conference – Spring Compcon 90*, pages 20–24, San Francisco, California, February – March 1990. IEEE Computer Society.

[133] MasPar Computer Corporation. *MasPar MP-1 Architecture Specification, Rev. A5*. MasPar Computer Corporation, Sunnyvale, California, July 1991. Part Number 9300-5001.

[134] MasPar Computer Corporation. *MasPar System Overview, Rev. A5*. MasPar Computer Corporation, Sunnyvale, California, July 1992. Part Number 9300-0100.

[135] Richard M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, January 1978.

[136] Ronald D. Levine. Supercomputers. *Scientific American*, 246(1):118–135, January 1982.

[137] T. Watanabe. Architecture and performance of NEC supercomputer SX system. *Parallel Computing*, 5:247–255, 1987.

[138] Rod A. Fatoohi. Vector performance analysis of the NEC SX-2. In *Proceedings of the 4th International Conference on Supercomputing*, pages 389–400. ACM Press, 1990.

[139] W. J. Watson. The TI ASC — a highly modular and flexible super computer architecture. In *Proceedings of the AFIPS Fall Joint Computer Conference*, volume 41, pt. 1, pages 221–228, Montvale, New Jersey, 1972. AFIPS Press.

[140] Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell. *Computer Structures: Principles and Examples*. McGraw-Hill Book Company, New York, New York, 1982.

[141] Roger Espasa, Mateo Valero, and James E. Smith. Vector architectures: Past, present and future. In *Proceedings of the 1998 International Conference on Supercomputing*, pages 425–432, Melbourne, Australia, July 1988. ACM.

[142] K. Shimizu and K. Uchida. System overview of the Fujitsu VP2000. *Fujitsu*, 41(1):3–11, January 1990. In Japanese.

[143] S. Kawabe et al. Supercomputer HITAC S-820 system. *Hitachi Hyoron*, 69(12):7–12, December 1987. In Japanese.

[144] R. G. Hintz and D. P. Tate. Control Data STAR-100 processor design. In *Proceedings of the 6th Annual IEEE Computer Society International Conference (Compcon 72)*, pages 1–4. IEEE Computer Society, September 1972.

[145] Robert E. Morley, Jr. and Thomas J. Sullivan. A massively parallel systolic array processor system. In K. Bromley, Sun-Yuan Kung, and E. Swartzlander, editors, *Proceedings of the International Conference on Systolic Arrays*. IEEE, May 1988.

[146] Jed Deame. Parallel processing solves the DTV format conversion problem. In *33rd AMIC*, Orlando, Florida, February 1999. SMPTE.

[147] Randy Schonhoff. Video pre-compression processing & bit conservation. Technical report, Teranex, Inc., 2000.

[148] D. W. Blevins, E. W. Davis, R. A. Heaton, and J. H. Reif. BLITZEN, a highly integrated massively parallel machine. In *Proceedings of the 2nd Symposium on the Frontiers of Massively Parallel Computation*. IEEE, October 1988.

[149] Dorothy Wedel. Fortran for the Texas Instruments ASC system. *ACM SIGPLAN Notices*, 10(3):119–132, March 1975.

[150] Burroughs Corporation. *Array Processing System Fortran IV, Change No. 3*. Defense, Space, and Special Systems Group, Burroughs Corporation, Paoli, Pennsylvania, August 1971.

[151] K. Stevens. CFD – a FORTRAN-like language for the ILLIAC IV. *ACM SIGPLAN Notices*, 10(3):72–80, March 1975.

[152] R. Michael Hord. *The ILLIAC IV, The First Supercomputer*. Computer Science Press, Inc., Rockville, Maryland, 1982.

[153] R. H. Perrott. A language for array and vector processors. *ACM Transactions on Programming Languages and Systems*, 1(2):177–195, October 1979.

[154] Mark D. Guzzi, David A. Padua, Jay P. Hoeflinger, and Duncan H. Lawrie. Cedar Fortran and other vector and parallel Fortran dialects. In *Proceedings of Supercomputing '88*, pages 114–121, Orlando, Florida, November 1988.

[155] G. Paul and M. Wilson. An introduction to Vectran and its use in scientific computing. In *Proceedings of the 1978 LASL Workshop on Vector and Parallel Processors*, pages 176–204, 1978.

[156] Burroughs Corporation. *Burroughs Scientific Processor Vector Fortran Specification.* Defense, Space, and Special Systems Group, Burroughs Corporation, Paoli, Pennsylvania, 1978.

[157] Information Technology Industry Council (ITI). *American National Standard for Programming Language — Fortran - Extended.* New York, New York, 1992. http://www.ansi.org.

[158] High Performance Fortran Forum. *High Performance Fortran Language Specification.* Rice University, Houston, Texas, November 1994.

[159] Robert E. Millstein. Control structures in Illiac IV fortran. *Communications of the ACM*, 16(10):621–627, October 1973.

[160] Thinking Machines Corporation. *Getting Started in CM Fortran.* Thinking Machines Corporation, Cambridge, Massachusetts, November 1991.

[161] Eugene Albert, Joan D. Lukas, and Guy L. Steele, Jr. Data parallel computers and the FORALL statement. In *Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 390–396. IEEE, October 1990.

[162] Kenneth E. Iverson. *A Programming Language.* John Wiley & Sons, Inc., New York, New York, 1962.

[163] D. H. Lawrie, T. Layman, D. Baer, and J. M. Randal. GLYPNIR – a programming language for Illiac IV. *Communications of the ACM*, 18(3):157–164, March 1975.

[164] Alan J. Perlis. The American side of the development of Algol. In *The First ACM SIGPLAN Conference on History of Programming Languages*, pages 3–14, 1978.

[165] Peter Naur. The European side of the last phase of the development of ALGOL 60. In *The First ACM SIGPLAN Conference on History of Programming Languages*, pages 15–44, 1978.

[166] Thinking Machines Corporation. Introduction to data level parallelism. Technical report, Thinking Machines Corporation, Technical Report 86.14, April 1986.

[167] John R. Rose and Guy L. Steele, Jr. C*: An extended C language for data parallel programming. Technical report, Thinking Machines Corporation, Technical Report PL87-5, April 1987.

[168] John R. Rose. C*: A C++-like language for data-parallel computation. Technical report, Thinking Machines Corporation, Technical Report PL87-8, December 1987.

[169] Thinking Machines Corporation. *C* Programming Guide.* Thinking Machines Corporation, Cambridge, Massachusetts, November 1990.

[170] MasPar Computer Corporation. MasPar data-parallel programming languages, 1990. Document number PL007.0190.

[171] Peter Christy. Software to support massively parallel computing on the MasPar MP-1. In *Digest of Papers Spring Compcon 90*, pages 29–33, San Francisco, California, February – March 1990. IEEE Computer Society.

[172] R. G. Zwakenberg. Vector extensions to LRLTRAN. *ACM SIGPLAN Notices*, 10(3):77–86, March 1975.

[173] Sergey Gaissaryan and Alexey Lastovetsky. An ANSI C superset for vector and superscalar computers and its retargetable compiler. *Journal of C Language Translation*, 5(3), March 1994.

[174] Anar Jhaveri and Hank Dietz. A simple vector language and its portable implementation. Technical report, School of Electrical Engineering, Purdue University, TR-EE 90-41, West Lafayette, Indiana 47907, May 1990.

[175] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. In *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–111, San Diego, May 1993. http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/public/papers/nesl-ppopp93.ps.gz.

[176] Guy E. Blelloch, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. *VCODE Reference Manual*, February 1994. http://www-2.cs.cmu.edu/afs/-cs.cmu.edu/project/scandal/public/code/nesl/nesl/doc/vcode-ref.ps.

[177] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Margaret Reid-Miller, Jay Sipelstein, and Marco Zagha. CVL: A C vector library. Technical Report CMU-CS-93-114, School of Computer Science, Carnegie Mellon University, February 1993. http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/public/papers/CMU-CS-93-114.ps.gz.

[178] L. Hamet and J. Dorband. A generic fine-grained parallel C. In *Proceedings of the 2nd Symposium on the Frontiers of Massively Parallel Computation*, pages 625–628. IEEE, October 1988.

[179] Thomas H. Lee. Vertical leap for microchips. *Scientific American*, 286(1):53–59, January 2002.

[180] AltiVec Working Group. AltiVec.README.txt. Technical report, SIMDtech.org, October 2002. http://www.simdtech.org/apps/group_public/documents.php.

[181] American National Standards Institute. *Information systems - coded character sets - 7-bit American National Standard Code for Information Interchange (7-BIT ASCII).* American National Standards Institute, Sunnyvale, California, 1997. http://www.ansi.org.

[182] George Boole. *The Mathematical Analysis of Logic.* Thoemmes Press, Sterling, Virginia, 1998. Reprinted from 1847 edition published by Macmillan, Barclay, & Macmillan with an introduction by John Slater.

[183] George Boole. *An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities.* Dover Publications, Inc., New York, New York, 1958. Unabridged reproduction of original edition published in 1854 by Macmillan, with corrections made in the text.

[184] Claude E. Shannon and Warren Weaver. *The Mathematical Theory of Communication.* University of Illinois Press, Urbana, Illinois, 1949.

[185] Randall Fisher. libMMX. Technical report, School of Electrical and Computer Engineering, Purdue University, December 1997. No longer available on-line.

[186] Hank Dietz and Randall Fisher. The libmmx homepage. Technical report, School of Electrical and Computer Engineering, Purdue University, May 1998. http://min.ecn.purdue.edu/~rfisher/Research/Libmmx/libmmx.html.

[187] P. J. Hatcher, A. J. Lapadula, R. R. Jones, M. J. Quinn, and R. J. Anderson. A production quality C* compiler for hypercube multicomputers. In *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, Williamsburg, Virginia, April 1991.

[188] A. Nisar and H. G. Dietz. Optimal code scheduling for multiple pipeline processors. In *1990 International Conference on Parallel Processing*, volume II, pages 61–64, Saint Charles, Illinois, August 1990.

[189] Jack J. Dongarra. Performance of various computers using standard linear equations software. Technical report, Computer Science Department, University of Tennessee and Mathematical Sciences Section, Oak Ridge National Laboratory, CS-89-85, October 2002. http://www.top500.org.

[190] Harvey G. Cragon and W. Joe Watson. A retrospective analysis: The TI Advanced Scientific Computer. *IEEE Computer*, 22(1):55–64, January 1989.

[191] Christopher Eoyang, Raul H. Mendez, and Olaf M. Lubeck. The birth of the second generation: The Hitachi S-820/80. In *Proceedings of Supercomputing '88 [Vol.1]*, pages 296–303, Orlando, Florida, November 1988.

[192] Steven J. Wallach. The CONVEX C-1 64-bit supercomputer. In *Proceedings of Spring Compcon '85*, February 1985.

[193] Kenneth E. Batcher. Flexible Parallel Processing and STARAN. In *1972 WESCON Technical Papers, Session 1*, volume 16, 1972.

[194] Active Memory Technology, Inc. *DAP Series Technical Overview.* Active Memory Technology, Inc., Irvine, California, 1988.

[195] Robert Schreiber. An assessment of the Connection Machine. Technical report, Research Institute for Advanced Computer Science, NASA Ames Research Center, June 1990. RIACS Technical Report 90.40.

[196] Marshall Brain. Blitzen tutorial one — what is Blitzen? Technical report, North Carolina State University, 1991. http://www.eos.ncsu.edu/eos/info/-eos_info/other_tutorials/blitzen/tutorials/bt1.

[197] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith. PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Transactions on Computers*, C-30:934–947, December 1981.

[198] T. Schwederski, W. G. Nation, H. J. Siegel, and D. G. Meyer. The implementation of the PASM prototype control hierarchy. In *Proceedings of the Second International Conference on Supercomputing*, volume i, pages 418–427, 1987.

[199] G. Jack Lipovski and Miroslaw Malek. *Parallel Computing: Theory and Comparisons*. John Wiley & Sons, Inc., New York, New York, 1987.

[200] G. Jack Lipovski and Anand Tripathi. A reconfigurable varistructure array processor. In *1977 International Conference on Parallel Processing*, pages 165–174. IEEE Press, 1977.

[201] James Gleick. *Genius: The Life and Science of Richard Feynman*. Random House, Incorporated, November 1993.

[202] Marc Tremblay, J. Michael O'Connor, Venkatesh Narayanan, and Liang He. VIS speeds new media processing. *IEEE Micro*, 16(4):10–20, August 1996.

[203] L. Kohn and N. Margulis. Introducing the Intel i860 64-bit microprocessor. *IEEE Micro*, 9(4):15–30, August 1989.

[204] K. Diefendorff and M. Allen. Organization of the Motorola 88110 superscalar RISC microprocessor. *IEEE Micro*, 12(2):40–63, April 1992.

[205] Leonard Gilman and Allen J. Rose. *APL: An Interactive Approach*. Krieger Publishing Company, Malabar, Florida, reprint of 3rd edition, 1992. Copyright 1984 John Wiley & Sons, Inc.

[206] Adin D. Falkoff and Kenneth E. Iverson. The evolution of APL. In *The First ACM SIGPLAN Conference on History of Programming Languages*, pages 47–57, 1978.

[207] Author Unknown. APL language summary. In *The First ACM SIGPLAN Conference on History of Programming Languages*, page 45, 1978.

[208] Marvin Schaefer. An APPLE tutorial. Technical report, System Development Corporation, SDC-TM5074/100/100, September 1973.

[209] David Gries. ACM SIGPLAN history of programming languages conference ALGOL 60 language summary. In *The First ACM SIGPLAN Conference on History of Programming Languages*, page 1, 1978.

[210] John Backus. The history of FORTRAN I, II, and III. *Annals of the History of Computing*, 1(1):21–37, July 1979.

[211] Unidentified text. Sections 12.3–12.5, pages 254–266, n.p., n.d.

[212] Walter S. Brainerd, Charles H. Goldberg, and Jeanne C. Adams. *Programmer's Guide to Fortran 90*. Unicomp, Inc., Albuquerque, New Mexico, 2nd edition, 1994.

[213] American National Standards Institute. *Fortran 8X, X3J3/S8 (X3.9-198x).* New York, New York, 1986. http://www.ansi.org.

[214] Guy L. Steele, Jr. Languages for massively parallel computers. In *Proceedings of the 2nd Symposium on the Frontiers of Massively Parallel Computation*, pages 3–13. IEEE, October 1988.

[215] Seema Hiranandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer, and Chau-Wen Tseng. An overview of the Fortran D programming system. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, pages 18–34, Santa Clara, California, 1991. Springer-Verlag.

[216] N. Wirth. The programming language PASCAL. *Acta Informatica*, 1:35–63, 1971.

[217] K. V. Nori, U. Ammann, K. Jensen, and H. Nägeli. The Pascal (P) compiler: Implementation notes. Technical report, Institut für Informatik, Eidgenössische Technische Hochschule, Zürich, 1975.

[218] A. S. Tanenbaum, M. F. Kaashoek, K. G. Langendoen, and C. J. H. Jacobs. The design of very fast portable compilers. *ACM SIGPLAN Notices*, 24(11):125–131, 1989.

[219] Inc. Sun Microsystems. The source for Java technology. Technical report, Sun Microsystems, Inc., 2002. http://java.sun.com.

[220] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1978.

[221] Dennis M. Ritchie. The development of the C language. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, pages 201–208. ACM Press, 1993.

[222] James T. Kuehn and Howard Jay Siegel. Extensions to the C programming language for SIMD/MIMD parallelism. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 232–235. IEEE Computer Society, August 1985.

[223] David Beech. A structural view of PL/I. *Computing Surveys*, 2(1):33–64, March 1970.

[224] Robert F. Rosin. ACM SIGPLAN history of programming languages conference PL/I language summary. In *The First ACM SIGPLAN Conference on History of Programming Languages*, pages 225–226, 1978.

[225] George Radin. The early history and characteristics of PL/I. In *The First ACM SIGPLAN Conference on History of Programming Languages*, pages 227–241, 1978.

[226] Digital Equipment Corporation. *Digital Semiconductor Alpha 21164PC Microprocessor Hardware Reference Manual.* Digital Equipment Corporation, September 1997.

[227] John H. Edmondson, Paul Rubinfeld, Ronald Preston, and Vidya Rajagopalan. Superscalar instruction execution in the 21164 Alpha microprocessor. *IEEE Micro*, 15(2):33–43, April 1995.

[228] M. Tremblay, D. Greenley, and K. Normoyle. The design of the microarchitecture of UltraSPARC-I. *Proceedings of the IEEE*, 83(12):1653–1663, December 1995.

[229] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual: Vol. 1 Basic Architecture*. Intel Corporation, 2002. http://developer.intel.com/design/pentium4/manuals/245470.htm.

[230] Inc. Advanced Micro Devices. *AMD Athlon MP Processor Model 6 Data Sheet*. Advanced Micro Devices, Inc., Sunnyvale, California, June 2001. http://www.amd.com/products/cpg/athlon/techdocs/pdf/24685.pdf.

[231] Doug Beard. MXi: A high-performance x86 processor with integrated 3D graphics. Technical report, Advanced Micro Devices, Inc., 1998.

[232] Cyrix Corporation. *Cyrix M II Data Book*. Cyrix Corporation, April 1998. http://www.national.com/cyrix/mii/mi-all.pdf.

[233] Corinna G. Lee and Mark G. Stoodley. Simple vector microprocessors for multimedia applications. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 23–36, December 1998.

[234] Intel Corporation. MMX techonology developer's guide. Technical report, Intel Corporation, 1999. http://developer.intel.com/drg/mmx/manuals/dg/devguide.htm.

[235] Advanced Micro Devices, Inc. *AMD Athlon Processor Technical Brief*. Advanced Micro Devices, Inc., August 2001. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/22054.pdf.

APPENDIX A
HISTORICAL PERSPECTIVE

In order to develop this new abstract model for modern SWAR architectures, we need to have a good understanding of related architectures and programming models. In this appendix, we discuss some of these architectures and the languages developed for programming them in relation to SWAR processing.

Vector Architectures

SWAR architectures are closely related to vector architectures in that both are designed to perform identical operations on sets of related data. Knowledge of these architectures, their features, and the issues traditionally associated with them should provide insight into how SWAR architectures may be best used, and may give clues as to the future of SWAR architectures.

In general, a typical vector processor has one or more sets of function units. These may be contained within a single processor (a uniprocessor) or spread across a group of connected processors (a multiprocessor). Each set contains one or more individual function units, some or all of which may be redundant. In this discussion, we will refer to a set of function units as a "processing element" (PE), regardless of the number of processors involved.

Having multiple function units allows multiple instructions to be issued at one time, as long as no two instructions require the same function unit simultaneously. For example, the execution of an addition and a multiplication in the same clock cycle can take place if separate adder and multiplier units are available. This is often referred to as *superscalar* operation. Almost all of the vector processors discussed below had superscalar PEs.

Redundancy of function units within a PE allows instructions of the same type to be issued simultaneously. These may or may not be part of the same vector instruction, depending on how the function units are used and controlled. In some cases these units are used independently, in superscalar fashion, such as when executing two unrelated additions in the same clock cycle. In other cases, they are used together to execute the same instruction on different parts of the same set of data. In this case, they are acting in a SIMD manner.

Most of the vector processors discussed below have multiple identical PEs. Such a system, whether a uniprocessor or multiprocessor, is essentially a parallel processing system. If these are driven by a single instruction, then they act as a SIMD system. If they are driven as independent sections, they act as a MIMD system instead. In this discussion, we are primarily concerned with pipelined and SIMD vector systems, which are similar to SWAR architectures, rather than MIMD vector systems, which are not.

STAR-100

Built in the early-1970s, the Control Data Corporation (CDC) STring ARray STAR-100 [144, 125] was one of the first vector supercomputers. Its superscalar vector unit consisted of two dissimilar pipelined function units. These were an adder/-divider/logical unit and a separate adder/multiplier. Both could produce a result during each clock cycle, so the vector unit could complete up to two vector element operations per cycle.

According to [141], each of the STAR-100's vector function units also had a SWAR-like feature: they could process one 64-bit operation or two simultaneous 32-bit operations. Special logic inserted between the two halves of the 64-bit datapath broke the carry chains between them. This effectively separated the datapath into two independent parts which performed identical operations. This method of partitioning the processor is essentially the same method used in modern SWAR architectures.

Special hardware was incorporated into the STAR-100 to handle sparse vectors, which consist mostly of zero-valued elements. These were stored as two separate vectors: one which held a bit mask indicating the non-zero elements and a second which actually stored those elements. When a sparse vector was accessed, the bit vector was checked for each element to determine if it needed be loaded or stored from the vector of non-zero elements.

The STAR-100 had several other innovative features. One was the use of bit masks for controlling conditional operations. Another was the ability to use "...stride[s] and gather/scatter memory accesses...." [141] This last point is contradicted, however, in [135] which claims that the STAR-100 could only handle single-strided accesses. Unfortunately, neither [141] nor [135] is a first-hand source, and I have been unable to obtain a copy of [144].

Operands were drawn from a fast main memory and results were stored there. Any result that was to be used in a subsequent operation was first written to memory, then read back from it. This memory-to-memory architecture resulted in slower than necessary inter-operation times. The STAR-100 ultimately failed due to this and its poor scalar performance. However, it was the beginning of the CDC line of vector processors which survived into the late 1980s.

TI-ASC

The Texas Instruments Advanced Scientific Computer (TI-ASC) [139, 190, 141] had an architecture similar to that of its contemporary, the CDC STAR-100. Both were first-generation pipelined, memory-to-memory, vector processors. Both employed bit masks for conditional execution and were capable of providing SWAR-like functionality.

There were some differences between the TI-ASC and the STAR-100. One was the apparent lack of sparse vector handling in the TI-ASC. The other was the functionality and expandability of the TI-ASC's pipelines.

The TI-ASC's central processor contained of a set of up to four identical, microprogrammed vector pipelines, each of which consisted of an combined adder/multiplier unit and was serviced by a combined load/store path. This allowed up to four elementwise operations to be applied simultaneously. These pipelines were driven by a single instruction processing unit and could be used in a SIMD manner [139, 140].

The ASC was TI's only venture into large scale computers [141] and failed for reasons similar to those of the STAR-100. Many of the features of the TI-ASC can be found in the digital signal processing chips currently produced by TI.

CRAY-1

Cray Research, Incorporated introduced the CRAY-1 [135] in 1976. Unlike the TI-ASC and CDC STAR-100 memory-to-memory architectures, it had a set of vector registers for storing vector operands and results. It also had a larger number of vector function units and was designed to allow better data flow between them.

A set of eight 64-word registers were used to store vector operands, which consisted of elements of 64-bits each. This allowed lower-latency access to data than memory-based architectures could achieve. Maintaining high-performance depended in part on making good use of these registers. The data path between memory and the vector register file was only a single word wide and thus could only supply one 64-bit word per clock cycle. Thus, the register file was necessary to provide data at a high enough rate to keep the multiple vector units sufficiently supplied.

The CRAY-1 had twelve independent, non-redundant function units which could be thought of as a single superscalar PE. Its function units were grouped into vector, floating point, scalar, and address units. Six of these, the vector and floating-point units, could be used to operate on vector data. These included integer and floating-point addition units, floating-point multiplication and reciprocal approximation units, and integer logical and shift units.

One of the strengths of the CRAY-1 was the ability to "chain" function units together to form a pipeline. As the individual elemental results of a vector operation left one function unit, they could be immediately forwarded to another for use as an operand before the first vector operation completed. This allowed a series of vector operations to be performed in an overlapped manner that is similar to the operation of a pipelined scalar processor.

The CRAY-1's memory system had 16 banks of 72 modules. Each 64-bit word was stored across the modules of a bank along with an 8-bit SECDED (single-error correction, double-error detection) code. The memory address space cycled through the banks so that sequential addresses were stored in neighboring banks and every 16th address occurred in the same bank. This allowed up to 16 sequential data words to be accessed with no two accessing the same bank of memory.

The CRAY-1 was descended from a line of processors developed by Seymour Cray at CDC including the 6600 and 7600 and was the first of the Cray line which continues today. The CRAY-1 was a significant improvement over the TI-ASC and CDC STAR-100 systems. However, as a non-parallel vector system, it was not able to fully make use of the data parallelism available in vector programs, and thus was not able to achieve the full potential of vector processing.

Cyber 205

The vector processor of the CDC Cyber 205 [136, 126] was a SIMD processor with up to four identical PEs. These were pipelined floating-point ALUs driven by a control unit which read a single instruction stream. Like its predecessor, the STAR-100, the Cyber 205 was a memory-based architecture. Its PEs had no registers and operated on data stored in the central memory.

The Cyber 200 series of computers, including the Cyber 205, had improved scalar processing over the STAR-100 and incorporated SIMD processing. However, its memory-based architecture was not a match for later register-based systems. The Cyber 200 line continued with the ETA-10 and eventually died out in the late 1980s [141].

VP200

Fujitsu, Limited introduced the VP200 in 1982 [141]. It had up to two identical sets of pipelines operating as a SIMD system. Each of these consisted of an adder/logic unit, a multiplier, and a separate divider. Like the CRAY-1, the VP200 was a register-based system. Each PE had a large register file supplied by two combined load/store units. The VP200 was part of the "first generation" of Japanese vector processors. Later generations would eventually dominate vector processing.

S810/20

Hitachi introduced the S810/20 in 1983 [141]. It had up to two identical vector PEs operating in SIMD mode. Each of these consisted of two adder/logic units, a multiplier with a cascaded adder, and a multiplier/divider which also had a cascaded adder. The S810/20 had 32 vector registers of 256 elements each. These were supplied with data via a set of three load units and a separate load/store unit for every PE.

SX-2

The NEC SX-2 [137, 138] vector parallel processor was introduced in 1984. Its vector unit had "four identical sets of functional units" [138] which worked in SIMD parallel fashion. Each set consisted of adder, multiplier, logical, and shift units. These could be chained to increase performance.

The SX-2 had forty 256-element vector registers which were connected to memory via a four word wide load path and a separate four word wide load/store path. The main memory could store up to 1GB of memory organized in 512 banks of 2MB each. Extended memory of up to 8GB was also available.

S820/80

The Hitachi S820/80 [143], introduced in 1987, had "the same basic architecture as its predecessor, the S-810." [191] According to [141], the S820/80 had a maximum of four identical vector PEs which were simpler than those of the S810/20. These consisted of a combined adder/logical unit, a multiplier with a cascaded adder, and a separate divider. Two of the load units were eliminated, leaving the S820/80 with a single load unit and one combined load/store unit. The S820/80 was part of the second generation of vector processors from Japan which were uniprocessor, SIMD vector machines.

VP2600

The VP2600 was introduced by Fujitsu in 1989 [142, 141]. Architecturally, it was primarily a refinement of the VP200 SIMD vector architecture. It had four identical PEs which were more advanced than those of the VP200. Each consisted of two multipliers, which each had an adder/logical unit cascaded behind them, and a separate divide unit. The memory paths were unchanged from the VP200 design. The VP2600 had a large register file with 2048 vector registers of 64 elements each.

Other Vector Machines

Other companies developed slower, less powerful, low-cost vector processors called "mini-supercomputers" [141]. These were intended to be affordable yet relatively powerful systems. They were often scaled-down versions of high-end vector processors and thus typically provided little in the way of architectural innovation. For this reason, they have been excluded from this discussion. One example of a mini-supercomputer was Convex Computer Corporation's C-1 [192] which was introduced in 1985.

In this discussion we have also avoided multiprocessor systems such as the Cray 2, X-MP, and Y-MP, and the later Japanese models such as the NEC SX-3 and

SX-4 and the Fujitsu VX and VPP series systems. The most significant features of these machines are related to multiprocessor and MIMD processing issues rather than SIMD vector processing. Thus, they are of less interest than the earlier models with respect to the subject of this thesis.

Summary

The purpose of this discussion was to develop an understanding of historical vector architectures so that we may better understand the relationship between them and modern SWAR architectures. Having knowledge of past vector systems gives us a baseline for comparing the capabilities and limitations of current SWAR processors.

SWAR processors are most closely related to pipelined SIMD vector systems. The latter are, as a general rule, uniprocessor systems which consist of one or more identical sets of pipelined function units driven by a single instruction stream. Each set of function units in a pipelined SIMD vector system can thus be thought of as a single pipelined, superscalar PE in a SIMD system.

A typical SWAR system is a pipelined uniprocessor with a data path that has been split into multiple parallel sections. These systems can be considered SIMD processors which are comprised of a linear array of identical pipelined, superscalar PEs. Conceptually, each PE consists of one section of the microprocessor's data path.

To better understand how these types of architecture are related, we can compare various aspects of their design including instruction fetch and decoding systems, function units, register files, memory systems, and conditional execution mechanisms.

A typical vector system has an instruction fetch unit which decodes a single instruction stream from a common memory and generates a set of control signals to drive the system's function units. This is essentially what happens in SWAR architectures. Here, a single multimedia instruction is fetched from main memory and decoded to generate a set of control signals. These determine which operation will be performed and how the data path will be partitioned into parallel sections. Thus, a

SWAR system is a SIMD parallel system whose configuration is determined by, and may change with, each instruction executed.

Most traditional vector processors have superscalar architectures, as do modern microprocessor systems. In each case, the number and type of function units available varies between architectures and determines their capabilities. Early vector processors had one or two function units per processing element while the second generation tended to have significantly more. Later generations were more balanced and tended to have a moderate number of function units. Similarly, SWAR architectures can be expected to undergo an evolutionary process as their use becomes more refined.

A typical vector processor's function units were pipelined to allow a result to be generated with every clock cycle. This is also true of the typical SWAR architecture, in which the data path of a pipelined CPU is split into multiple independent sections.

Chaining, in which function units are connected to form a pipeline, is similar to data forwarding techniques used in modern pipelined scalar processors. Some multimedia-enhanced architectures may allow for this type of forwarding as a natural consequence of using existing pipelines for SWAR processing.

The earliest vector processors were memory-based architectures whose performance suffered from their long memory latencies. Later vector processors incorporated multi-element vector register files. These registers were capable of storing multiple data words as a single entity and allowed intermediate results to be stored internally. This reduced the effective latency of these processors' memory systems, which in turn allowed them to have shorter clock cycles and achieve better performance than memory-based vector processors.

Modern SWAR architectures are register-based machines in which existing or especially-designed registers are used with multimedia instructions. These registers are typically one to four words wide and few in number. For example, a register file containing 32 registers of 128 bits each (4kb) would be very large by SWAR standards. This is an eighth of the size of the CRAY-1's register file, which had eight registers of 64 64-bit elements (32kb), and less than one percent of the size of the SX-2's 40 registers of 256 64-bit elements (640kb).

Traditional SIMD vector uniprocessors typically had a common memory shared by each of their processing elements. To increase performance, later vector processors were equipped with banked memories which allowed multiple simultaneous accesses. They were also often designed with a set of memory access pipelines for each independent set of function units in the system. This allowed each set to obtain data and store results at a rate independent of the other processors as long as there were no addressing conflicts.

SWAR processors are similar with one caveat — each memory access touches a contiguous set of bits in a common main memory. No individual addressing is possible because these systems use memory data paths which are split in the same way as their function units. Thus, each word in memory can be thought of as being spread out across several banks of memory which are accessed simultaneously. The degree of interleaving depends on the precision of the data stored and the word size of the architecture.

More complex memory accesses such as strides and gather/scatters are difficult to implement on SWAR systems. Strided accesses are often used to access elements of an array along one of its minor axes. Gathers and scatters are typically used to compress and expand sparse matrices or vectors. Some SWAR architectures allow strided accesses to occur, but gathers and scatters require a level of indirection that current SWAR systems cannot provide.

The first generation of vector processors had only one or two function units per independent section. The throughput of these systems was thus limited to a few scalar operations per clock cycle. Hence, these architectures could be served by a data path that was at most a few words wide.

Later vector processors had a moderate number of function units and large vector register files. Due to their prohibitive costs, the memory pipelines of these systems were often kept narrow in comparison to the size of their vector registers, but wide enough to keep the function units supplied with data.

Thus, the bandwidth of the full memory-to-memory data path was typically limited to the data rate of the functional pipelines. This meant that to maintain peak performance, the vector registers had to be filled at the same rate, and at the same time, that they were being emptied.

Over time, as the number of processing elements has increased, the bandwidth required to maintain high efficiency has also increased. Because of this, later vector processors had some of the highest bandwidth memory systems ever built.

Because SWAR systems have modified microprocessor architectures, their memory systems are often pre-defined by their underlying architectures. Generally, they are able to load or store an entire multimedia register with each clock cycle. Depending on the width of the data path and the precision of the data being accessed, each memory access may move between one and several vector elements. For example, a microprocessor with 64-bit registers and a 64-bit memory path can load the register in one clock cycle. If the data loaded is 8-bit data, then this single load brings in up to eight vector elements in one clock.

SIMD processors must also deal with the issue of conditional execution. When a conditional branch is encountered in a program, the condition may be true for some of the processor's PEs, but not for others. Normally, every SIMD instruction is executed by all of the PEs in the system, but when a condition does not hold for some set of PEs, there needs to be some mechanism to prevent them from executing the related instructions or to block or undo their effects.

Early SIMD vector processors used bit masks to track which of their PEs were enabled to execute instructions and which were disabled due to failing some conditional test. These were typically used with masked stores to prevent side effects from occurring. Modern SWAR processors may use a variety of methods to perform this basic task. These are discussed in more detail later in this chapter.

From this discussion, certain trends can be recognized in vector processor architecture. While the number of function units within each processing element has tended to level off, the number of processing elements themselves has increased significantly. The size of their vector register files has gone from zero to well into the hundreds of kilobits range, and their memories and bandwidths have also increased dramatically.

Current vector processors are significantly more complex than any of those discussed above. This complexity makes them less like current commodity SWAR architectures than were earlier vector systems. For this reason, we have avoided discussing them in this section; however, the trends they exhibit are still worth briefly noting.

These machines are generally multiprocessor MIMD systems with a large number of identical PEs. These typically have a few well-designed, pipelined function units which are used in superscalar or VLIW (Very Long Instruction Word) mode, depending on whether the parallelization is performed in hardware or by a compiler. They are also increasingly connected to allow data to be transferred directly between them. In many ways, vector processors are becoming increasingly like the parallel array processors which will be discussed next.

SIMD Array Architectures

Modern SWAR architectures are also related to traditional SIMD array architectures. Brief descriptions of several of these are presented here with an emphasis on their relationship to SWAR processing. This should provide an understanding of the evolution of these processors and of how modern SWAR architectures are constrained in comparison.

SOLOMON

The SOLOMON [117, 118] prototypes were early SIMD processors built for the Western Electric Company in the early 1960s. Their design was inspired by the physical appearance of a magnetic drum used as storage for the IAS machine built at Princeton University.

The original design called for 512 byte-wise "processing elements" connected in a 2-dimensional toroidal mesh, and controlled from a "central source." In the final design for the original prototype, the PEs were to be grouped into up to eight 32x8 subarrays for a maximum of 2048 PEs. The actual prototypes used significantly fewer PEs in various configurations.

The PEs were essentially single bit full adders which used operands that were either stored in local core memory frames, broadcast from the central source, or read over the link from any of the PE's nearest neighbors. An "L-buffer" was used to convert word-sized data from the control unit into a serial bit stream for the PEs. The length of this stream was variable and determined by the value of a settable register.

The central source was responsible for program storage and supplying immediate data, but its primary task was to act as a controller for the rest of the system. This it did by providing the PEs with control signals to select operations, enable or disable PEs, and activate connections between the PEs.

Each PE had a 2-bit "mode" register whose value determined which of four possible modes the PE was operating in. Each instruction carried a 4-bit field (one per state) which specified a set of possible modes. Only the PEs that were in one of these modes were allowed to execute the instruction. This allowed PEs to be effectively disabled for a given instruction. While "off", a PE could supply operands to its neighboring PEs but was not allowed to change state.

Later SOLOMON prototypes replaced the bit-serial PEs with byte-sliced processors. These had 24-bit registers and 8-bit arithmetic hardware. As PEs with wider data paths began being used, the number of PEs in successive prototypes was scaled down. A full-scale model was never built; however, the design led directly to the ILLIAC IV.

SOLOMON provides us with a basic model of a SIMD array processor: an array of processing elements, controlled by a single control unit, with local memories, and connected via some form of interconnection network.

ILLIAC IV

The first operational SIMD machine was the ILLIAC IV [119], built at the University of Illinois. It was an extension of the SOLOMON prototypes, and was built by a group led by D. L. Slotnick and which included others from the SOLOMON project. Both [152] and [123] contain case studies of the ILLIAC IV, and some of the following material is drawn from these sources.

The ILLIAC IV was contracted by the Department of Defense's Advanced Research Projects Agency (ARPA) in about 1965. A quarter-sized prototype was developed and used at Illinois until the early 1970s when it was decided that it should be moved to a government facility. The prototype was delivered to the NASA Ames Research Center in 1972, but was not fully operational until 1975. The ILLIAC IV was decommissioned in 1982.

As delivered, the ILLIAC IV's processing array consisted of one quadrant of 64 processing elements (PEs). Each of these had an arithmetic/logic unit (ALU), various registers, and a local processing element memory (PEM).

The ALU could perform arithmetic, logical, and comparison operations on data in its four 64-bit data registers. These could be loaded from local memory or with a value broadcast by the control unit (CU). The operation applied depended on the control signals from the CU and the values stored in the PE's flag registers. These contained status and control values and were accessible by both the PE and the CU. This allowed the various PEs to behave differently while executing the same instruction, and allowed conditional execution based on an individual PE's computational results.

Attached to each PE was a local bank of memory from which its data stream was normally drawn during parallel operations. These banks held 2k words of 64 bits each, and were accessible by both the PE and the CU. Each PE could access a local memory location that differed from that of the other PEs. This was done by indexing the address by the value in the PE's 16-bit index register.

ILLIAC IV's control unit drove the processor array by issuing control signals to the PEs over a “nanoinstruction” bus. It could set the PEs' flag registers and mode bits with different values to conditionally enable or disable sets of PEs. The CU could also broadcast data and addresses over a 64-bit common data bus. This allowed it to transmit scalar values and constants to the PE array. A mode “flip-flop” bus collected a single bit from each PE and delivered the set to the CU as a 64-bit word. This word could then be tested to determine global conditions.

The CU could access the PEs' local memories directly over a separate 512-bit bus. This allowed it to use all of memory and treat the PEs' memory banks as a single global store. The CU read its instructions from this memory and fetched them into an instruction cache. Data also could be loaded from this memory and stored in a private buffer.

The PE interconnect was an 8x8 mesh, with each column connected as a separate torus, and the rows connected together as a single torus. This allowed data to be rotated through the columns of the mesh or through the entire set of PEs. It also allowed nearest neighbor communications in any of four directions. This was useful for moving data vectors and arrays which had been mapped onto the processor array.

The ILLIAC IV was capable of a form of variable-width processing. Each PE could operate as a single 64-bit floating-point element, as two 32-bit floating-point elements, or as eight 8-bit fixed point elements. Whether this was implemented in a manner similar to that of modern SWAR architectures is unclear from [118]. This variable-width processing made the architecture more flexible in its ability to support various data types; however, the languages used to program the ILLIAC IV tended not to take advantage of this capability.

ICL DAP

The first commercial massively parallel computer was the Distributed Array of Processors (DAP) built by International Computers Limited (ICL) [120]. It was developed in the mid 1970s based on a "design study" by S. F. Reddaway [121]. Work on a prototype took place during the rest of the decade, and the original system was delivered to Queen Mary College, London University in 1979. In the mid 1980s, Active Memory Technology (AMT), Inc. was spun-off from ICL to develop DAP systems.

The idea behind the ICL DAP was to use bit-serial processors to simplify the logic design and provide these with local memories to closely integrate the logic and storage systems. With enough processors, the entire problem could theoretically be mapped onto the processing array.

The paper design consisted of a main control unit (MCU) and a processor array, and was "somewhat similar to SOLOMON 1" [121]. The DAP was to be connected to, and supported by, a "parent" computer system which provided it with data and instructions.

The MCU was to consist of a "conventional" instruction fetch system, an instruction buffer, and a set of registers which could be connected to a row or column of the array. These registers would allow data to be loaded to, or retrieved from, the array along either of its sides, and was apparently intended for use in processing scalar data.

The processor array was to be two-dimensional with one side connected to the "store highway" of the parent system. The parent could then load data and instructions via this bus and hence could use the array for storage or computation. A word of data was normally to be stored along a column of PEs in what was called "main store mode". Words would also be stored in a single PE in "array mode" for more efficient processing in some circumstances. Conversion between these two modes would occur within the PE array.

The PEs were to be connected in a rectangle via a nearest neighbor network with each PE also connected to the PE "half a row away in the same row". Independent, program controlled edge connections were to allow the PEs to be connected as a linear array, a ring, a mesh, a toroid, or any of up to 32 geometries when half-row connections were used.

Each PE would have a 4kb local memory. Storage to these memories could be blocked by the MCU on a row or column basis to allow operations on array subsections. Each PE also had a set of single-bit registers which were to be used to hold operands and buffer incoming and outgoing bits. One of these was used as a mask bit to control conditional execution according to [126].

The design emphasized connectivity and allowed several input and output connections to be made with the MCU, the parent machine's store highway, and neighboring PEs. Multiplexers were to be used to activate connections between the registers and the various sources and destinations.

MPP

Goodyear Aerospace Corporation's Massively Parallel Processor (MPP) [122, 123] was developed in the late 1970s and built in the early 1980s. It was the first so-called "massively parallel processor," which meant that it contained thousands of PEs. It consisted primarily of an array unit which housed the PEs and an array control unit which directed them. The MPP was descended from the STARAN [193] bit-serial associative processor, and was similarly intended for image processing using bit-slices.

The array unit (ARU) was a 128x128 array of bit-serial PEs. Each PE had a set of six bit registers, a programmable shift register, a full-adder, and a Boolean logic/routing unit. Arithmetic operations were performed bit-serially, with the result stored in either the shift register or local memory. Most instructions could be prevented from executing on a particular PE by resetting a "mask bit" in that PE's status register.

Each PE had 1kb of local memory. Because the MPP was designed to operate on bit planes, these were used collectively with the planes stored across the entire set. During a memory operation, all of the PEs accessed the same local address, and thus the same bit plane. Thus, these memories could be thought of as a set of 1024 bit-planes, with each PE controlling one bit in the same position of each 128x128 plane. Data was typically from 1 to 32 bits in length and was stored across multiple, consecutive bit-planes. Thus, a set of consecutive bit-planes could be used to represent an array of multi-bit items.

Because the depth of the memory array was fixed at 1028 bits, the MPP could store 128 8-bit images or 32 32-bit images. The more precise the pixel data, the fewer pixels the MPP could store. This trade-off is similar to one found in SWAR architectures in which a fixed number of bits are available in the CPU's registers. This fixed number must be traded off between data precision and parallelism width.

Instructions for the MPP's ARU were handled by the array control unit (ACU) by placing them in a "call queue" to be read by the ACU's PE control section. Each of these instructions was executed as a microprogram by the PE control unit which generated one stream of control signals which it broadcast to the entire PE array. For memory accesses, these signals included a single address that was used by all of the PEs simultaneously.

The MPP had multiple interconnects including a reconfigurable mesh, a global OR network, and an aggregate word network. The inter-PE mesh network allowed nearest neighbor communications in any of four directions called north, east, west, and south. For this reason, it was called the *NEWS network*. It connected the PEs in a 128x128 mesh whose topology was controlled by the ACU. This was done by controlling the connections of the PEs at the edges of the mesh. The PE at the edge of a column could be connected with the PE at the other edge of the same column or left disconnected. The same was true of the rows, except that the PE at one edge of a row could be connected to the PE at the other edge of the next row. This flexibility allowed the MPP to be connected as a mesh, a vertical or horizontal cylinder, a torus, an open or closed spiral, or a spiral torus. This level of flexibility can only be achieved in SWAR architectures which support permutations.

A second network, called the "sum-or" network, performed a bitwise ORing of the bits sent by the PEs to the control unit. The SWAR equivalent to this global OR network would be a test of the CPU register for a non-zero value. In both cases, this allows aggregate data to be collected and tested easily.

A single bit could also be collected from each of the 16 PEs in the southeast corners of the ARU's 32x32 subarrays. These formed a 16-bit aggregate value that the ACU could access and manipulate as a single word. This third network was more general than the sum-or network.

The MPP could be difficult to use for higher-dimensional problems and for problems which did not match its dimensions. While the NEWS network was more flexible than the mesh of the ILLIAC IV, it still required all PE data communications to follow the same pattern. For example, for one PE to send data to its northern neighbor, all the other PEs had to do the same.

The ILLIAC IV and MPP represented opposite ends of the SIMD continuum. The ILLIAC IV had a relatively small number of fairly powerful multiple-bit processors, while the MPP had a large number of very simple one-bit processors. This was a result of the two architectures having been designed for different purposes. ILLIAC IV was intended to be a number-cruncher, operating primarily on 32- and 64-bit measured data, while MPP was intended to be an image processor, operating primarily on 8-bit, or at most 32-bit, pixel data.

AMT DAP

The AMT DAP [194] was a successor to the ICL DAP and was built in the late 1980s by Active Memory Technology (AMT), Inc. The 500 series had a 32x32 array of 1-bit processing elements while the 600 series had a 64x64 array. In both systems, each PE had between 32k and 1Mbit of local private memory. Thus, both versions were somewhat smaller than the MPP, but had significantly more memory.

The PEs were bit-serial and consisted of a full adder/logical unit, three 1-bit registers, and two multiplexers. One of these was used to choose the operand sources, which could be any of the registers or interconnection networks. The other was used to choose the source of the result sent to the memory and interconnects.

The PEs were arranged in a two-dimensional mesh with each connected to its four nearest neighbors in a NEWS network, and also to all of the PEs in its row and to all of the PEs in its column via buses. This interconnection was more flexible than the MPP's NEWS network, allowing a PE's data to be broadcast within a row or column, or even to the entire array.

The master control unit (MCU) was a 32-bit CPU. It read instructions from the code memory and issued control signals to the PEs in the processing array. It also performed scalar operations and could broadcast data to the array.

One unique feature was a hardware DO instruction which could encompass other instructions. These instructions could then access various sections of the array in an incremental manner, with the index automatically incremented for each iteration.

Later versions of the DAP had an 8-bit co-processor which was used for computation while the 1-bit processors were used for communication. This medium-sized collection of moderately powerful processing elements represented a trade-off between the ILLIAC IV and MPP architectures. This allowed it to be more commercially acceptable on a price/performance basis.

GAPP

The NCR Geometric Arithmetic Parallel Processor (GAPP) [1] was a single chip SIMD processor which consisted of a control unit and a 6x12 array of PEs connected

[1] Part number NCR45CG72.

by a two-dimensional NEWS network. A brief description of the GAPP can be found in [145] which explains its use in a particular application.

The PEs were bit-serial full adder/subtracters which could perform basic arithmetic and logical operations. Operands were drawn from a set of four 1-bit registers which buffered data that was either drawn from a set of common memory lines or the NEWS network, or was generated as carries or borrows during arithmetic operations. The generated output included the sum (SM), carry out (CY), and borrow (BW) bits.

A set of five multiplexers were used to choose the source of the data latched during the execution cycle. These were chosen from any of the registers, the ALU outputs, or the data incoming over the interconnect. Data could also be moved between the host processor and the PEs' local memories via a set of common data lines, called CMN and CMS.

Each PE had a relatively small 128-bit local memory, addressed by a 7-bit immediate field in the instruction opcode. This meant that all PEs addressed the same location in their local memories during memory accesses.

GAPP chips had a set of I/O ports which allowed them to be connected into larger processing arrays. This allowed the chip to be used by others to develop larger systems. One example is the systolic array constructed by Morley and Sullivan [145].

While the GAPP processor is over a decade old as of this writing, it is still in use. A current video processing/conversion system, the TeraNex video computer [146] is based on the sixth generation of the SIMD microprocessor which was introduced in 1998 by Lockheed Martin Electronics and Missiles. This processor, called the GAPP VI, is implemented as a single chip with 1k PEs in a 32x32 mesh. These can be combined in a 32x32 array for a total of over one million PEs.

The GAPP is actually a full SIMD architecture on a chip, and represents a more powerful architecture than the SWAR architectures we are concerned with in this research. If multimedia, especially image processing, continues to be a driving force in computing design, then the GAPP and similar architectures may move into the commodity market. For now, they are used in specialized architectures.

GF11

The IBM GF11 [124, 125, 126] was a pipelined SIMD parallel processor designed for verifying research in quantum chromodynamics. It had 576 pipelined PEs with high-speed register files. Each of these had 64KB of high-speed memory and 256KB of lower-speed memory. The lower-speed memory was expandable to 2MB per processor, allowing up to 1.125GB in total.

Possibly the most interesting feature of the GF11 was that the PEs were fully interconnected via a non-blocking Beneš network [127]. This network had three stages of 24x24 crossbars and allowed the PEs to be connected in any arbitrary permutation in order to share data. Thus, the PEs could be connected in a large number of various multi-dimensional shapes.

CM-1

The Thinking Machines Corporation's first "Connection Machine", the CM-1 [128, 129, 130], was another massively parallel SIMD system consisting of a parallel processing unit which contained a very large array of PEs, a front-end host computer which read instructions from its memory and issued nanoinstructions to the PEs, and interconnection networks between the PEs and between the PEs and the front end.

The CM-1 had up to 64k PEs – significantly more than previous architectures. These were bit-serial ALUs which could perform any of 2^{32} functions on their inputs. Each PE had a private memory from which two input bits were taken and a set of flags from which a third input bit was drawn. Output consisted of one bit which was stored in memory and another which was stored in the flags register.

Conditional instructions on the CM-1 were executed based on the value of a specified processor flag. For each PE, if this flag had a specified value, then the instruction was executed; otherwise, it was skipped. Thus, like the ILLIAC IV, conditionals were performed on an instruction-by-instruction basis, with no sense of nesting.

The CM-1 had multiple interconnection networks. A NEWS network connected the PEs in a two-dimensional mesh and moved bits between neighboring PEs' flags. This was used for short distance and regular pattern communications. A global OR network combined data from the PEs into a single scalar value that was passed to the front-end. This provided aggregate data to the controlling system.

The most interesting of the CM-1's interconnects was, however, an adaptive packet-switched hypercube network. It allowed any set of PEs to communicate with any other set in an irregular pattern. This network was significantly more complex and powerful than those of earlier systems, whose interconnects did not allow such general communication.

Messages on this network passed through a packet-switched adaptive router. Each set of 16 PEs was connected to a single router node, and these nodes were connected to form a hypercube. Collisions within the hypercube were resolved by using other paths; thus, the router network adapted to internal loading. However, because multiple PEs were connected to a single router node, contention for access to the router was possible, and blocking could occur as a result.

In relation to SWAR architectures, use of the router network is analogous to executing a generalized permutation instruction. These instructions allow any type of permutation of the data fields in a CPU register to be selected including replications. Thus, they can be used to perform broadcasts and generalized communications between fields just as the CM-1's router network could be used for interprocessor communication.

The CM-1 represented a return to the ideas behind the Goodyear MPP, but with the addition of the hypercube network to facilitate the types of communication that the MPP was weak at. This allowed the CM-1 to be used for problem types that the MPP performed poorly on.

CM-2 / CM-2a / CM-200

The Thinking Machines CM-2 [131, 130], CM-2a, and CM-200 were updates of the CM-1 with various sizes and options [2]. The CM-2 could have 16k, 32k, or 64k PEs, while the the CM-2a could have 4k or 8k PEs. "CM-200" may have been the name assigned to a version with floating-point co-processors. The basic architecture was essentially the same as that of the CM-1, although there were some significant modifications.

One difference was the addition of a sequencer between the front end system and the PE array. It read instructions from the front end system and issued nanoinstructions to the PEs, thus taking over this part of the duties of the CM-1's front-end. The broadcast and scalar memory buses which had previously connected the PEs to the front-end now connected them to the sequencer instead.

A second difference was the modification of the global OR network to a more general combinatorial network which connected the PE array to the sequencer. This network could perform global reductions such as maximum, summation, and logical AND on the PE data to form a single value which the sequencer then received. This was a significant advancement over the previous global network, and could be used to gather more diverse information about the system's aggregate state. In relation to SWAR architectures, this was the equivalent of adding advanced reductions to the instruction set.

Another difference was the modification of the NEWS network. First, it was extended from a two-dimensional mesh to an n-dimensional mesh implemented on top of the existing hypercube. This upgrade allowed regular communications patterns in multiple directions. Second, it was modified to perform scans and spreads. *Scans*, which are also known as *parallel prefix* operations, are reductions in which the running subresults are retained. *Spreads* are operations which replicate a value throughout

---

[2] While the literature is somewhat contradictory with respect to the features of the various Connection Machine implementations, the specifics are not crucial to the understanding of this thesis.

the PE array. These operations are often used in data parallel processing, especially when a calculation is split-up between PEs.

In terms of SWAR architectures, no current system performs scans, although they can usually be emulated rather easily, but expensively, using shifts. Spreads are found on some architectures, while others attempt to obviate them by including instructions which use a single data field as a scalar operand to each of the elemental operations which comprise the vector instruction.

Operations on multibit data were executed bit-serially within the CM-2's 1-bit ALUs, while single- and double-precision floating-point operations were processed on an optional floating-point accelerator. This consisted of one floating-point memory unit and one floating-point processing unit per pair of processor chips (i.e. one accelerator per 32 PEs). The memory unit acted as a glue chip which converted data between a collection of 32 single bits or 32 pairs of single bits and a single- or double-precision floating-point value that the processing unit could operate on. Thus, it worked similarly to the MPP's bit-slice processor.

An analysis of the CM-2 and its use at the Research Institute for Advanced Computer Science (RIACS) at NASA Ames Research Center, written by Robert Schreiber [195], provides a more in-depth analysis of the system's utility, strengths, and weaknesses.

BLITZEN

The goal of the BLITZEN [148] project at the Microelectronics Research Center of North Carolina was to develop a miniaturized massively parallel processor. Such a processor was expected to be economical and easily attached to, or embedded in, other systems. While the chip layout was submitted for fabrication, it appears that a prototype system was never built [196]. Despite this, the architectural definition is instructional.

Each chip contained an 8x16 array of bit-serial PEs driven by an on-chip control unit. The control unit converted microcoded routines into the control signals for the array. These routines were stored in a control memory and could be loaded from the host machine via an external interface. This interface could also be used to transfer data between the chip and the host's peripherals.

The PEs had essentially the same design as those of the MPP, but employed various modifications. One was a redesign of the shift register to make it bidirectional and to limit the shifted bits to a selected set, thus protecting the data in the unselected bits. This made the register more generally useful and allowed parts of it to be used for temporary storage and address indexing. Another modification was the extension of masking to all memory accesses. A third was the addition of complementary conditional operations. These performed either the specified operation or its complement depending on the value of a control bit on each of the PEs. This allowed the PEs to take opposite actions simultaneously, and could be used to simplify certain control structures.

Each PE had 1kb of local on-chip memory which was individually addressable using the contents of the PE's shift register as an offset to the globally supplied address. This was done by bitwise ORing them together, and required that the global address be aligned on a 10 bit boundary. This memory organization had two advantages over that of the MPP. First, it was more flexible because it allowed its PEs to access different locations in memory. Second, the BLITZEN design was theoretically faster because memory accesses were on-chip and thus didn't suffer from off-chip delays.

Data could be transferred over a set of 4-bit buses, each of which connected a row of 16 PEs and provided a port for memory accesses. This allowed memory to be accessed in 4-bit blocks during row I/O operations. An innovative interconnection network called the X-grid was also incorporated in the design. This network connected each PE with eight nearest neighbors: its four NEWS neighbors and its four diagonal neighbors. The X-grid was more flexible than a NEWS grid, yet was significantly smaller than a full routing network and required fewer I/O pins.

Each PE had four connections — one leaving each of its four corners. Every four neighboring PEs which formed a square were connected via their corners within the square in an X connection. By choosing which corners the PEs would send data out, and from which they would read data in, the X-grid could be used to connect the vertical, horizontal, or diagonal pairs of PEs. The unused lines would be effectively disconnecting by placing them in a high-impedance state.

Like the GAPP, the BLITZEN architecture was an attempt to place a full SIMD architecture on a single chip and represents a possible future direction for commodity processors.

MP-1 and MP-2

The compute engine of MasPar Computer Corporation's MP-1 [132, 133, 134] was called the data processing unit (DPU). The DPU consisted of a PE array of between 1k and 16k nodes, an Array Control Unit (ACU) which also performed scalar arithmetic, and multiple interconnection networks.

While the arrayed PEs were 4-bit ALUs, microcode was used to make them behave, from a programming perspective, as 32-bit processors. Thus, the MP-1 was another compromise architecture, falling between the massively parallel 1-bit machines and those with fewer, more powerful PEs. Each of the MP-1's PEs had forty 32-bit registers and was connected to its own local memory of between 16 and 64 kB.

Floating-point support consisted mainly of fast normalization hardware which decreased the time needed to normalize the integer mantissa and exponent parts of the operands. This sped-up what is often the slowest part of a floating-point operation. Floating-point data could be single- or double-precision, and could be in VAX or IEEE formats.

Communication between the PEs could be accomplished in two ways. First, an X-net provided straight-line communication in any of 8 directions. This may have been a re-invention of the BLITZEN X-grid or an independent invention by MasPar. Either way, it provided the same level of interprocessor communication and had the same advantages as the BLITZEN X-grid.

Second, a three-stage global router network, similar to that of the CM-1, allowed simultaneous, independently-indirected, duplexed communications between pairs of PEs. The PEs were grouped in clusters of 16, with each cluster having a single connection to the router network. This connection was multiplexed between the PEs in the cluster, and operated in a bit-serial fashion.

As in earlier architectures, communication between the PE array and the control unit was also provided for. Communication from the ACU to the PEs took place over a broadcast network, and communication from the PEs to the ACU took place over a global OR network.

As a later SIMD array architecture, the MP-1 benefitted from many of the lessons learned from previous architectures. While similar to the CM-1 and CM-2, the MP-1's architecture was more of a compromise, combining a fairly large number of processors with a reasonable amount of memory and multiple types of interconnection networks. The MP-2 was essentially a scaled-up version of the MP-1 which had thirty-two 32-bit PEs per chip with a floating-point unit attached to each PE. Thus, it too represented a compromise between the two extremes in SIMD array architecture.

Summary

The purpose of this discussion was to develop an understanding of historical SIMD array architectures so that we may better understand the relationship between them and modern SWAR architectures. This should make it easier to set reasonable goals and avoid pitfalls while designing a SWAR processing model.

SWAR processing is a limited form of SIMD implemented within a single microprocessor. A traditional SIMD array architecture consists of a control unit, a processing array, memory, and an interconnect. Each of these has a SWAR architecture counterpart. We will briefly discuss the relationships between them.

The primary task of a SIMD control unit is to read instructions and decode them into control signals for the processor array. In a SWAR architecture, the analogue of the control unit is the normal CPU instruction issue mechanism. An instruction is read from a single instruction stream in memory and decoded into a set of control signals. These signals specify a single operation to be performed by the ALU or other functional units. They also turn off logic such as the carry and borrow chains to ensure that the operation acts independently within each of the fields of the affected registers.

In a SWAR architecture, each register field can be thought of as a small, complete register residing on one of the PEs of a SIMD system. The set of fields located in the same position across the set of CPU registers can then be thought of as a particular PE's register set. That PE consists not only of this set of register fields, but also of that part of the CPU's data path which operates on them. Thus, a SWAR system can be thought of as a one-dimensional linear array of PEs.

Thus, a SWAR system is really a vector parallel processor in which vector elements are stored in the fields of the CPU registers. In contrast, traditional SIMD systems were usually multi-dimensional array processors with each PE holding one array element in each of its registers. This implies that many of the problems that map easily to SIMD array processors will not map easily to SWAR architectures.

In a typical SIMD array processor, each PE had a local memory which was often shared by, or at least accessible to, the control unit. On a SWAR system, data is loaded or stored in chunks that are often larger than a single field. For example, a load that matches the size of the partitioned register is equivalent to all of the SWAR PEs loading a value from the same address of a banked memory. In this sense, most SWAR memory systems are similar to that of the ILLIAC IV.

On most SIMD architectures, a PE's local memory was not available to the other PEs. Continuing this analogy, on most SWAR architectures a PE cannot directly access data from another PE's part of memory. This is because loads and stores are usually performed on word-sized entities and preserve the bit ordering of the data. An access of another PE's memory slice would be equivalent to performing such a load with a simultaneous shift of the data to the desired position.

In some SIMD architectures, the control unit also acted as a scalar unit. In SWAR processors, non-SIMD instructions treat the contents of the registers as single values regardless of their origins or any earlier partitionings. If the ALU is thought of as a scalar unit when executing normal instructions, it is one with direct access to the global memory consisting of the PEs' local memories. This is true only if the architecture supports instructions which operate on the entire register contents. Often, this is not the case, and is a weakness of several of the current SWAR architectures.

One of the weaknesses in early SIMD array architectures that was addressed in later generations was the lack of sufficient communications capabilities. Early mesh systems were sufficient for regular communications patterns, but it became clear that more generalized capabilities were needed. As SIMD systems evolved, more complex interconnects were developed to provide these capabilities.

Most SWAR architectures have the one-dimensional equivalent of a NEWS network which can be emulated using logical shifts and rotates; but few have any equivalent to the general communication capabilities of a full router network, which requires some form of permutation instruction. Because of this, a good portable model should probably avoid this generality, at least until SWAR architectures mature a little more.

One other aspect of SIMD processing requires discussion. As with SIMD vector processors, SIMD array processors had to incorporate some means of allowing separate control paths to be taken by different PEs. In most systems this was done by turning the PE off on an instruction-by-instruction basis. Usually, this was done by the control unit, but in some cases the PEs could turn themselves off based on the status of an executed instruction. SWAR architectures do not have equivalent functionality.

SWAR instructions are executed across an entire register; thus, all SWAR PEs execute the same instruction.

Other SIMD systems allowed the PE to execute the instruction, but prevented the side-effects of execution from occurring. Some SWAR architectures employ masked stores to accomplish this. These operations store only those register fields which are selected by some type of mask. As long as the data precision used matches one of the hardware-supported field sizes, masked stores can be used to block unwanted side-effects during conditional execution.

Where no hardware support for conditional execution is available, arithmetic nullification must be used to block the effects of execution on those PEs for which the condition doesn't hold. This was used on some SIMD systems, and can also be used on SWAR architectures. Arithmetic nullification is also necessary if the data precision doesn't match any supported field size.

It is clear that SWAR architectures, while similar to traditional SIMD systems, also differ from them in significant ways. SWAR architectures are less mature and more restricted than the later SIMD systems. As we discuss the specifics of commodity SWAR architectures in the next chapter, we will be able to do so with a perspective gained from knowledge of past SIMD architectures.

Reconfigurable Architectures

SWAR architectures are also related, though less closely, to *reconfigurable architectures*. These are architectures whose processing model or logical configuration can be changed without actually changing the hardware, either as the machine is running or between runs. A detailed study of these architectures is unnecessary; however, we will briefly discuss two in order to compare and contrast them to SWAR architectures.

PASM

The PArtitionable SIMD/MIMD (PASM) system [197, 198] was a dynamically repartitionable architecture in which the system could be partitioned, while running, into several smaller SIMD and/or MIMD systems to perform separate parallel tasks. As the needs of the tasks changed, the system could be reconfigured on the fly. This allowed multiple processes to use the array simultaneously, and in a manner that best fit their needs.

SWAR architectures, by contrast, are much less flexible than was PASM. SWAR systems are always SIMD and are not partitionable into separate parallel subsystems. They can, however, dynamically change precision and parallelism by changing how their data paths are partitioned into logical PEs.

Reconfiguration on PASM was explicit, meaning that a program executed a separate instruction to set the configuration of the system, then executed other instructions under that configuration. Reconfiguration on a typical SWAR architecture occurs implicitly with every multimedia instruction executed. The multimedia instruction determines the configuration of the system, but only during is own execution — no state is maintained between instructions.

While modern SWAR architectures share some hardware aspects with PASM, the focus of this research is the development of a programming model for systems in which the partitioning of individual registers is dynamic. SWAR architectures are dynamically partitionable not in the sense of tasks, like PASM, but in the sense of the layout of fields in the register set. Thus, while a study of architectures such as PASM's can provide insight into the design of modern microparallel architectures, they are not particularly relevant to the current work.

TRAC

The Texas Reconfigurable Array Computer (TRAC) prototype consisted of four 8-bit processing elements connected to nine memory modules via a Banyan network [126, 199]. To perform an operation, the network switches were set to form an *instruction tree* rooted at one of the memory modules which would be used to send instructions to a set of PEs. A set of separate *data trees* rooted at each of these PEs were also formed. These were used to access data during the operation.

A more important feature of the TRAC from the perspective of SWAR processing was its *varistructure*. This allowed PEs to be ganged together to perform higher-precision operations. The TRAC's PEs were byte-slice (i.e. 8-bit) processors which could be combined to perform operations on data sizes which were multiples of eight bits. Because the prototype had only four processors, it was limited to 8-, 16-, 24-, and 32-bit operations, but would allow any combination of data precision and set size whose product was limited to 32.

TRAC was an extension of the Reconfigurable Varistructure Array Processor [200]. For this architecture, the precision and vector sizes were specified by the programmer via dimension declarations. The trees were then built, with the PEs ganged together via exposed carry networks. By passing the carry signals between PEs, multi-byte precision objects were formed, and by blocking the carry signals multiple elements of a vector were formed. This is similar to modern SWAR architectures which also control the carry chain to create sets of equivalent elements of various sizes.

A later version, TRAC 2.0, was built at a time when 32-bit microprocessors were affordable enough to use as the PEs. Varistructure, which combined byte-slice processors to form multi-byte objects, was no longer needed. Each PE in the TRAC 2.0 design could handle 32-bit and smaller objects itself. Because of this, the TRAC 2.0 design is not particularly relevant to SWAR processing.

Early Forms of SWAR Processing

SWAR-like processing is not a new concept. Various forms have been used to exploit limited machine resources such as memory and register space for some time.

As demonstrated in this chapter, both the ILLIAC IV and the MPP could perform datapath partitioning to operate on data in a SWAR-like fashion.

In fact, James Gleick indicates in "Genius" [201] that Stanley Frankel, a mathematician at Los Alamos during the second World War, modified IBM 601 multipliers, which performed a single ten digit multiplication, to perform three separate three or four digit multiplies simultaneously. This was clearly a form of SIMD processing, and, depending on the design of the multipliers, may even have been a form of SWAR processing. I have not been able to obtain more specific information on this work, nor was Mr. Gleick able to guide me to the original source of this information [3], so I cannot confirm this.

Early work in applying this form of processing to microprocessor systems focused on enhancing these processors with on-chip graphical hardware. These were later generalized into the multimedia extensions currently in use. A short history of SWAR-like multimedia extensions is given in [29]. There is also some discussion of early SWAR-like architectures in [202]. In this section, we will discuss some of these early SWAR processors.

Intel i860

In 1989, Intel introduced the i860 microprocessor [203]. This was the first general-purpose microprocessor to incorporate SIMD-style instructions for graphics processing [202]. This functionality was intended to accelerate "back-end rendering operations" such as "shading and hidden surface removal." [203]

The i860 had a three-dimensional graphics processing unit that could operate simultaneously on a set of pixels stored in any of its 64-bit floating-point registers. When used in this manner, these registers were partitioned into sets of eight 8-bit pixels, four 16-bit pixels, or two 24- or 32-bit pixels.

[3] James Gleick, email to author, 19 December 2001.

A set of ten graphics instructions were supported by the i860 which performed operations such as z-buffer checks, pixel intensity interpolation, and z-distance interpolation. These were used for determining which pixels were closest to the viewer, and therefore must be visible, and for rendering unstored, but visible, points between polygon vertices.

Motorola 88110

The 88110 [204], introduced by Motorola in 1992, had a set of about nine SIMD instructions for performing "...fixed-point shading and image processing." These instructions operated on pixel or color intensity data stored in the 88110's 64-bit general registers. The 88110 had separate pack/unpack and arithmetic units and could issue an instruction to each on every clock cycle.

Graphical data was normally stored as "pixels" in packed format. These consisted of four 8-bit integer values stored as a 32-bit entity. It appears that these were normally operated on in an "unpacked" format with four 16-bit fixed-point values stored in a 64-bit register. Instructions for unpacking pixel data into fixed-point form and packing fixed-point data into pixel form were included.

The 88110 allowed modular and signed or unsigned saturation addition and subtraction on 8-, 16-, and 32-bit unpacked fixed-point data. *Modular arithmetic* refers to operations in which only the bits that can fit into the assigned storage space are stored. Overflow bits are ignored, although side effects such as the setting of condition codes may occur. This is equivalent performing a modulus operation after the arithmetic operation, and is how arithmetic is traditionally handled on computing systems.

Saturation arithmetic refers to operations in which overflow is prevented by setting the result of an operation to the maximum storable value of the same sign when an overflow would have occurred and to the minimum storable value of the same sign when a negative overflow would have occurred. *Signed saturation* refers to performing saturation arithmetic while treating the data as signed values. *Unsigned saturation* refers to performing saturation arithmetic while treating the data as unsigned values.

Multiplication of fixed-point data by an 8-bit integer scalar value was also supported. This instruction multiplied each 16-bit unpacked fixed-point field by the same 8-bit value to form a set of 16-bit results stored in unpacked fixed-point form. This allowed color intensity values to be scaled simultaneously by the same amount.

Other instructions included rotation of the fields of a register by a constant or variable amount and z-buffer comparison operation. The rotate could operate on 4-, 8-, 16-, and 32-bit fields, presumably in unpacked form.

Texas Instruments MVP

Introduced in 1992, the Texas Instruments multimedia video processor (MVP) [116] was a single-chip parallel processor intended for "...general integer DSP or bit and pixel manipulation...." The architecture allowed for one to eight parallel processing units controlled by a "master processor".

Each parallel processing unit had a 32-bit ALU which could perform arithmetic operations in a SWAR-like manner. These were referred to as "split ALU" operations and could be performed on either two 16-bit or four 8-bit register fields simultaneously. It is unclear from [116] exactly which operations could be performed in this manner.

The MVP was a highly specialized high-performance architecture intended for DSP and graphics manipulation algorithms. I am unsure if any processor was ever built based on this architecture.

Parallel Programming Languages

Because SWAR architectures implement a limited form of SIMD processing, it makes sense to try to develop a SIMD-like abstract model to program them. However, it is clear that past SIMD architectures differ somewhat from modern SWAR architectures. Because of this, the programming models developed for SIMD machines may not work well with SWAR architectures. Also, while SWAR architectures are similar to SIMD architectures, their operation more closely fits the one-dimensional vector processing model than the multi-dimensional array processing model. In order to develop a good SWAR programming model, it is best to have some understanding of both.

A large number of programming languages have been developed for programming vector and SIMD parallel processors. In this section, several of these are discussed in order to gain an understanding of vector and SIMD programming models and how they have been embodied in these languages. Having an understanding of the relationship between these models and languages will be useful when developing a usable SWAR processing model. We will also borrow from these languages to develop an experimental SWAR programming language.

Most parallel programming languages are based on previously existing computer languages, so it is useful to group them into families of languages which are based on a common ancestor.

APL-based Languages

APL [162, 205] was developed starting in early 1956 "as a tool for describing and analyzing various topics in data processing, for use in teaching classes, and in writing a book...." [206] APL is rooted in mathematics and has a syntax similar to that of algebraic notation. Thus, APL programs are essentially mathematical expressions.

In APL, vectors and arrays are "first-class" objects. This means that the language allows the programmer to concisely describe the task at hand as simple high-level operations on vectors and arrays rather than as a series or loop of low-level operations on their individual elements. This, in turn, makes it easier for a compiler to recognize parallelizable operations.

Vectors and arrays are operated on using a set of primitive "functions" (operations) which are defined in an implementation-independent manner [207]. These include arithmetic, Boolean, and relational operations, and other operations such as array element selection. Operations on scalars are extended in a consistent way to array operands and handle them in elementwise fashion. These operations have no "side effects" which are hidden from the programmer and are thus well-suited to parallelization.

APL also introduced several advanced features. These include reduction and scan (parallel prefix) operators. Reductions combine the elements of a vector or array to form a single scalar result. For example, adding all the elements of an array together. Scans are reductions in which each of the intermediate results is also kept, not just the final result. For example, keeping the running total for the above example as each element is added in. Other features were an "axis" modifier which indicated that an operation was to be applied across the rows or columns of an array, and inner- and outer-product operators which resulted in a scalar or array respectively.

Because of its mathematical basis and consistent treatment of scalar, vector, and array objects, APL might be a good choice for SWAR processing. However, APL has several aspects which make it less desirable as a basis for a SWAR language. For one, it is a dynamic language. Array types and dimensions are often undeclared and must be determined by the compiler [126]. Also, variable types may change during processing. While these features make APL versatile, they also make building a properly working compiler for it a difficult task.

APL also makes use of symbols that are not part of a modern microcomputer's repertoire. Its character set is based on that of the IBM 1050 terminal and includes a number of symbols not available in the ASCII character set which is used on most modern systems. Finally, APL differs significantly from the languages most often used by programmers in the high-performance area. This may be the most damning, as programmers tend not to use unfamiliar languages no matter how well designed they are. For these reasons, the SWAR model presented in this thesis will draw from APL's strengths, but our incarnation of it will be based on a more universally accepted language.

APPLE [208] was intended to be a general-purpose parallel language for the ILLIAC IV. Like APL, it was highly dynamic and allowed operations on vectors and arrays to be described efficiently. These would be performed in parallel on the PE array. While this language may have been useful for SWAR processing, the project was abandoned after proving to be too difficult to implement correctly for the ILLIAC IV [126, 152].

ALGOL-based Languages

ALGOL [164, 165] was developed in the late 1950s. It was intended to be a well-designed, machine- and application- independent language for expressing algorithms with conciseness and structure.

ALGOL was the first block-structured language [126]. It allowed programs to be hierarchical and better organized than those written in earlier languages such as FORTRAN. It also allowed for dynamically allocated local variables, recursive procedures, and call-by-value and call-by-name parameters [209]. This structure had a price in that code written in FORTRAN tended to be compiled to more efficient machine code. Thus, programmers concerned with performance tended to use FORTRAN instead.

ALGOL has been the basis for much theoretical work in computer languages, and has influenced the design of many subsequent languages. ALGOL was a sequential language, but at least one parallel language, GLYPNIR, was based directly on it.

GLYPNIR [163] was a general-purpose language intended to provide a stable, efficient method of programming the ILLIAC IV. Designed in 1968, it was one of the first attempts at the development of a true SIMD language. GLYPNIR was an extension of ALGOL 60 which allowed parallelism to be expressed explicitly in terms of 64-word vectors (the size of the ILLIAC IV's PE array).

GLYPNIR differentiated between what were called CU variables and PE variables. CU variables represented scalars and vectors of scalars that would normally reside on the ILLIAC IV's control unit, while PE variables represented swords or vectors of swords (sword vectors) residing on its PE array. A *sword* was the group of 64 words at the same address in each of the PEs' local memories. A *sword vector* was a collection of swords contiguously allocated on each of the PEs.

PE variables were first-class objects, and operations on them were executed in parallel across the PE array. Using PE variables to index a sword vector allowed a *slice* to be accessed. This was a group of 64 words residing on the PE array at possibly different local addresses in each PE. Thus, GLYPNIR allowed what would later be called "vector-valued indexing" or "vector indexing" of a vector or array.

GLYPNIR allowed data to be stored in a packed format along the same lines as modern SWAR architectures. The partitioned object was represented by an identifier, but could not be operated on in a SWAR manner. An individual piece of data was accessed by modifying the identifier with a bit field specifier which defined the range of bits to be accessed. A sword of bit fields could be operated on in parallel just as with any other type of sword.

IF and *ELSE* statements were parallelized, with implicit PE enabling, if their conditional tests were PE expressions. This was also true for *FOR*, *DO*, and *WHILE* loops. A *FOR ALL* construct was added as an alternative equivalent syntax for the parallelized *IF*. These constructs allowed the programmer to express parallelism using familiar means.

GLYPNIR also included the Boolean quantifiers `SOME` and `EVERY` which could be used to test aggregate conditions and provide a scalar result. These were TRUE if a condition was TRUE for some or all of the tested elements, respectively.

Unfortunately, GLYPNIR was not designed to hide the architecture of the ILLIAC IV from the programmer. In fact, quite the opposite was true. PE variables always defined a set of 64 objects which were spread across the width of the machine's processor array. Operations on larger data sets had to be *strip-mined* (i.e. split into a series of operations on smaller parts of the data set) by the programmer to fit within this limit. This exposure of the architecture makes GLYPNIR unsuitable as a basis for a portable SWAR model.

FORTRAN-based Languages

There are a large number of parallel processing languages based on FORTRAN (the FORmula TRANslation system). This language was developed by a group at IBM led by John Backus in the mid-1950's [210]. It was originally designed as a means to program the 704, a commercial SISD computer, in a manner which more closely represented the scientific problems of the end-users than other languages of the time. In later incarnations, the name was changed to Fortran to signify the acceptance of case-sensitive sources.

Fortran has a long history as a language for scientific and technical computing, and has been in continual use since its inception. The proverbial "dusty decks" of punch cards are typically Fortran sources that few people want to make significant changes to unless there will be sufficient pay-off. As a result, much research has centered on converting sequential source code into vectorized or parallelized machine code. This is typically done by parallelizing the iterations of code loops.

As architectures evolved, so did Fortran. Newer versions of the language treat arrays and vectors as first-class objects. Thus, looping constructs are no longer necessary for describing vector and array operations. Unfortunately, much of the dusty deck code is still written as looped constructs. Thus, while Fortran has grown to allow new paradigms, it has also been forced to continue supporting the old ones.

Because of its history, Fortran is the most widely used language for high-performance computing. This same history has also transformed it into a large and unwieldy language with many archaic features which are only slowly being removed. This makes Fortran a non-optimal choice for the basis of a new programming model. Despite this, much can be learned from its evolution, so it is worth studying. In this section, we will concentrate on describing versions of Fortran used on vector and SIMD processing systems.

ILLIAC IV FORTRAN [150] by Burroughs Corporation was developed sometime before 1968 and was the earliest parallel version of FORTRAN for the ILLIAC IV [154]. This language introduced some simple constructs for supporting parallel processing which were used in later languages.

Parallelism was supported via a notation in which an asterisk was used as an array index. This indicated that operations on the array should be applied to each of its elements in parallel. Thus, arrays could be treated almost as though they were first-class objects. Some later versions of Fortran used a similar notation. We will refer to the use of this notation as *wildcard indexing*.

ILLIAC IV FORTRAN also introduced the use of "control vectors" as array subscripts to indicate conditional execution. Each element of a control vector had either a .true. or .false. value. When used as an array subscript, the value of each control vector element indicated whether or not operations were to be performed on the corresponding element of the array. This allowed elementwise conditional operations to be written as operations on arrays rather than as loops of conditional scalar code.

These constructs provided rudimentary support for parallel processing, but were somewhat restrictive. Later parallel dialects of FORTRAN would build on this starting point and were significantly more complex.

IVTRAN [159] was an extension of ILLIAC IV FORTRAN which allowed more complex parallel operations to be performed on arrays of integer or floating-point data. This was done by adding new looping and data allocation constructs which helped the compiler to find and extract useful parallelism.

The primary mechanism for expressing parallelism was a new `DO FOR ALL` construct which was somewhat similar to a `DO` loop. This construct indicated that certain assignments within the loop were logically simultaneous and could therefore be parallelized. These assignments were denoted as scalar element assignments and were required to be of a certain form.

Rather than having a single index variable as with standard Fortran `DO` loops, `DO FOR ALL` loops could be indexed over a set of variables called a “control multi-index”. Each member of this set represented an axis of the object or objects to be accessed. A related logical expression specified a range of values for each axis to be operated on. The values thus specified were called the “index set”. This allowed a subarray to be selected for parallel operation within the loop body. PE enabling for the selected elements was handled implicitly.

Using IVTRAN required having knowledge of the ILLIAC IV’s architecture. To achieve efficient speedup, the data had to be laid-out so that it could be accessed in parallel. This required the programmer to structure arrays to match the underlying architecture. Two constructs were provided to help in this endeavor. The first, an optional *allocation declaration*, allowed the programmer to specify the layout of an array. The second was an `OVERLAP` specifier which allowed an array to be transformed between multiple layouts in place during processing.

While IVTRAN had certain features which may be useful in a SWAR model, the exposure of the architecture made it non-portable, and thus not useful in the current effort. The language also had a short life, having been replaced by CFD soon after the ILLIAC IV was delivered to NASA [126].

CFD [151, 152] was a FORTRAN-based language designed primarily to allow computational fluid dynamics code to be ported to the ILLIAC IV (hence the name). CFD was not intended to be a general-purpose language and was intentionally tied to the underlying architecture. This allowed programmers in NASA’s CFD research branch to optimize code for the ILLIAC IV target.

Parallelizable “vector-aligned” arrays of up to three dimensions were allowed, but the first dimension was required to be less than or equal to the number of PEs. Parallel operations on these were pseudo-first-class using a wildcard indexing scheme similar to that of ILLIAC IV FORTRAN. This was extended to allow expressions over wildcards to denote rotations of the indexed object.

Scalars and non-parallelizable arrays resided on the CU and operations on them were performed there. These were thus limited to the operations which the CU could perform, while a different set of operations could be performed on the vector-aligned arrays residing on the PEs.

Thus, the language not only required the user to have knowledge of the target architecture, it codified the differences between its functional units. These issues make CFD unsuitable for use as a basis for a portable programming model.

Despite this, CFD did have certain features which could be incorporated into a modern SWAR programming model. Like GLYPNIR, CFD had used `CU` and `PE` storage class modifiers which explicitly indicated where the data should be stored. Logical `IF` statements with parallel conditionals were parallelized and implicitly handled PE enabling, thus hiding these issues from the programmer. Finally, explicit `.ANY.` and `.ALL.` quantifiers, which were similar to GLYPNIR's `SOME` and `EVERY` tests, could be used to obtain aggregate information. CFD expanded upon these with new `.NOT ANY.` and `.NOT ALL.` quantifiers which performed complementary tests.

TI-ASC NX Fortran [149, 154] was a vector Fortran developed for the Texas Instruments Advanced Scientific Computer — a parallel pipelined vector processor. This language was introduced around 1973. The NX compiler was one of the first vectorizing compilers, capable of converting standard Fortran 66 code into vector machine code. To make it easier to make use of the TI-ASC's capabilities, NX Fortran also included some array processing features.

Vectors and arrays were apparently first-class objects in NX Fortran which could be referenced in expressions and assignments by simply using their names. No special indices or loop constructs were necessary to invoke parallel operation on these objects. This was essentially a notational improvement over previous versions of FORTRAN which brought them closer to GLYPNIR or APL.

Elementwise array assignments could be performed as long as the right-hand side of the assignment conformed to the shape and size of the object on the left-hand side. Scalars on the right-hand side were promoted to a conforming shape via replication.

A set of "array generating intrinsics" allowed "the generation of an array cross section from other array cross sections by use of vector instructions...." [149] Various reduction intrinsics were also available which generated an aggregate scalar value from a vector or array argument.

Other features allowed access to subsections of multi-dimensional objects. A `SUBARRAY` statement allowed the dynamic aliasing of an array subsection to another array of the same rank. This allowed multiple operations to be performed on the subsection without requiring the subsection to be specified for each one. Cross-sections of an array could be specified using an asterisk wildcard index. This indicated the full range of possible values for that index from one to the object's length in that dimension. Choosing a particular index value for only one dimension caused a cross-section of rank $n - 1$ to be selected from an object of rank n. A negated asterisk could be used to specify the full range of values from the object's length down to one.

From the perspective of designing a model for vector-based SWAR processing, the most significant contribution of NX Fortran was probably its use of first-class vector objects. Vectorization is used primarily when the source is based on a scalar programming model. Cross-sections are trivial unless the object is a multi-dimensional array — the cross-section of a vector is a single element. Thus, most of NX Fortran's features are better suited to non-SWAR programming models.

Vector LRLTRAN [172, 154] was developed, also around 1973, at the Lawrence Livermore Labs. It was intended to allow programmers to make use of the vector capabilities of the CDC STAR-100 vector processor by extending the LRLTRAN version of Fortran with vector features. Vector LRLTRAN was also used to code programs for the TI-ASC, ILLIAC IV, and CDC 7600 before the STAR system was delivered.

The language supported single-strided (i.e. contiguously allocated), one-dimensional vectors as first-class objects. These vectors could consist of `REAL`, `INTEGER`, or `BIT` data. Vector declarations, assignments, expressions, subscripting, and functions were included to support these objects.

Vector LRLTRAN allowed vector objects to be declared in a manner similar to a Fortran `DIMENSION` statement. Vectors differed from arrays in that they were first-class objects and thus could be operated on as a single object. Related to each vector was a "descriptor". This was essentially an index into a table containing the memory address of the first element and the number of elements in each vector.

If the vector was declared with a scalar dimension, it was allocated statically and assigned a new descriptor which could not be changed during execution. If the vector was declared using a descriptor, but no dimension, the information in the descriptor was used to allocate the vector. If both were given in the vector declaration, then the dimension was assigned to the descriptor and the vector was allocated using this information. In each of the last two cases, the descriptor was available to the programmer during execution; thus, the vector's size and location were dynamically alterable.

LRLTRAN's scalar operators were extended to perform in elementwise fashion on vector operands including sparse vectors. Pure vector and mixed expressions could be written, with promotion and vector extension performed as necessary. If the size of the vector operands did not match, the shorter vector was extended with an identity value for the applied operation.

Available operations included the standard Fortran arithmetic, Boolean, relational, and logical operators. Location and mode (type) operators, and a set of "STAR-specific" vector operators were also included. Arithmetic operations on vectors were parallelized and had vector results. Boolean and logical operations could only be applied to bit vectors and produced a bit vector as a result. Relational operations could be applied to any type of vector, but produced a bit vector as a result.

Assignments could also be performed on vector objects using the same syntax as scalar assignments. When assigning a scalar value to a vector, the scalar was replicated to conform to the vector's size. Vector to vector assignments were performed elementwise to fill the result vector. If the result was longer than the right-hand side vector value, then the remaining elements of the result were undefined.

The language also allowed subvectors to be defined using "dynamic equivalencing" in which a range of vector elements could be assigned an alias and operated on as a single entity. This was similar to vector assignment, but did not create a new object.

Vectors could also be used as function arguments and return values. Although the length of the returned vector had to be specified upon declaration of a user-defined function, this size could be a run-time value. When called, the calling routine was responsible for evaluating the size of the return value and allocating space for it.

Vectors and parenthesized vector expressions could be indexed using several different methods. These allowed a single element, a range of elements, or any set of elements to be selected for use in expressions. Scalars could be used as with arrays to specify a single element. Non-bit vector expressions could be used as an index vector which listed the elements to be selected. This allowed arbitrary permutations of a vector to be generated. Bit vectors could also be used as indices and acted like a control vector, indicating whether or not each element would be used in the current operation.

Semicolon-separated offsets could be used as an index. These specified the number of elements to dismiss at either end of the vector. The result was the remaining elements from the middle of the vector. Either offset could be omitted and defaulted to zero.

Colon-separated limits also could be used as an index. These specified the first and last elements to include in the result. Either limit could be omitted. The lower limit defaulted to zero, while the upper defaulted to the length of the vector minus one.

The STAR-specific operators were `.LGTH.`, `.VEC.`, `.DES.`, `.CTRL.`, `'`, `:`, and `;`. These were used to obtain information about a vector, manipulate it, or select subsections of it. The colon and semicolon index operators were just described. The others can be described briefly.

The length of a vector could be obtained using the unary `.LGTH.` operator. The `.VEC.` and `.DES.` operators were used to manipulate vector descriptors. Conceptually, `.VEC.` converted its scalar argument to a vector descriptor which could be used in expressions and assignments. The `.DES.` operator returned the descriptor of its vector or vector expression argument. This allowed the programmer to obtain, copy, or modify a vector's descriptor.

Similar to ILLIAC IV FORTRAN, control vectors were bit vectors used to implement conditional execution by indicating which elemental results of a vector expression were to be stored. These were used with the binary `.CTRL.` operator which applied the control vector, or an expression which evaluated to one, which preceded it to the vector expression which followed it. Only one `.CTRL.` operator could be used per statement and it was not allowed to be enclosed in parentheses.

A representation for sparse vectors was also included in the language. These were stored as a pair of vectors. The "value vector" contained the non-zero element values, while the "order vector" stored a bit vector indicated which elements contained these non-zero values. Sparse vectors were denoted as an apostrophe-separated value/order vector pair.

A set of inlineable intrinsic vector functions were also included in LRLTRAN. One set of these performed arithmetic reductions on their vector arguments. `Q8SUM()` and `Q8PROD()` performed, respectively, reduce-add and reduce-multiply operations on their vector arguments. Each of these could also take a control vector as a second argument. This specified a subset of the vector's elements to be used in the calculation.

A second set of intrinsics could be used either as functions, which returned a result, or subroutines, which required pre-allocated storage for the result to be made available to them. This second set of intrinsics included `Q8MASK`, `Q8MERG`, `Q8CMPRS`, and `Q8XPND`. The first two of these combined two data vectors using a control vector to select which elements from each data vector would be selected. `Q8MASK` selected one of the two data elements whose index corresponded with that of the result element, while `Q8MERG` treated the data vectors as two stacks and used the control vector to choose from which the next sequential result element would be taken. `Q8CMPRS` was used to compress a vector into sparse form, while `Q8XPND` was used to expand a sparse vector back to normal form.

Vector LRLTRAN had a large number of vector-handling features, some of which are beyond our current needs or the capabilities of current SWAR architectures. However, a number of them can be incorporated into a SWAR model or used to implement a SWAR-based programming language.

VECTRAN [155, 154] was introduced by International Business Machines (IBM) Corporation in 1978 [4]. Triplet notation, `identify` statements, and `where` constructs were introduced by this language and/or **BSP Fortran** which was introduced about the same time by the Burroughs Corporation [156, 154].

Triplet notation allowed the programmer to reference sections of arrays using a concise notation. A triplet was a comma-separated list consisting of the indices of the first and last elements along a particular dimension to be accessed and an optional stride to be used between successive elements. This notation allowed the programmer to describe regular patterns of access without using looping constructs.

Each part of the triplet had a well-chosen default value which made commonly-accessed subsections trivial to describe. If the first index was omitted from a triplet, the first element in the array was used. Similarly, if the last index was omitted, the last element was used. An omitted stride was set to one.

Triplet and standard index notations could be mixed as long as corresponding dimensions had the same number of elements. When used as an array index, triplet notation allowed the programmer to express regular patterns of access without using looping constructs. However, triplet notation did not allow conditional selection as did ILLIAC IV FORTRAN's control vectors.

The `where` construct allowed conditional assignment in a manner that was more flexible than control vectors. A conditional vector expression was evaluated and

[4] [154] reports this date as 1973, but the cited work is from 1978.

used to decide which elements would be operated on. In a sense, the `where` created the equivalent of a control vector to be applied to its body. This body could only contain array assignment statements which conformed to the shape of the conditional expression. An `otherwise` statement was also included which operated on the set of elements where the condition did not hold.

The `identify` statement was used to allow the expression of operations that accessed memory in regular strides, but were denoted by array indices with irregular strides. For example, the diagonal of an array is typically stored with a regular stride of $n + 1$ for an array with dimensions of length n, but the correct set of indices cannot be described using triplet notation. The `identify` statement applied aliasing to create a smaller-dimensional object with correctly strided element indices. This object could then be used in a separate assignment statement.

VECTRAN handled parallelism somewhat more elegantly than earlier parallel versions of FORTRAN. Subsection selection and conditional execution were denoted using concise notations and language constructs. These features would be copied by several later languages.

DAP Fortran [211] was a variant of Fortran for programming the ICL DAP. It was influenced by CFD, but extended for use with the DAP's bit-serial architecture. It was developed in the late 1970s.

Two parallelizable data structures were defined which were clearly related to the geometry of the DAP's PE array: Two-dimensional arrays equal to the size of the PE array, and one-dimensional vectors equal to the size of one edge of the array. Higher-degree objects could be defined as arrays of lower-level objects. Thus, a programmer could declare an object that was a collection of arrays or vectors.

DAP Fortran, like Vector LRLTRAN, allowed expressions of mixed dimensions. In these expressions, lower-dimensional objects were replicated and promoted to match the dimension of higher-dimensional objects. This allowed the programmer to easily mix vector and array code.

A set of intrinsic functions were included which performed restructuring operations such as vector and array shifts and rotates, element and subarray selection, reductions such as summations and ANY and ALL tests, and a trinary merge which combined two objects based on the elemental values of a third. Masked assignments, which stored elements based on an element-wise conditional, were also available.

Because DAP Fortran was so closely tied to the DAP architecture, it is not a good candidate for a portable SWAR language. However, some of the ideas, such as defining high-level objects as collections of lower-level ones and dimensional promotion via replication, may be useful for a SWAR programming language.

Fortran 90 [157, 212] is an extension of the Fortran 77 language which allows the processing of vectors and multi-dimensional arrays. An interim version, Fortran 8X, was standardized during the late 1980s [213].

Vectors and arrays are treated as first-class objects in Fortran 90; thus, operations on them can be expressed with a simple syntax. A large number of operations and functions can be performed on these objects including elemental operations, conditional tests, array sectioning operations, reductions, and various intrinsic functions.

As with earlier languages, elemental operations behave as though they are applied independently across the elements of their array operands and are often parallelized. Their operands are required to be conformable in shape and size. As with NX Fortran, scalar objects were considered to be conformable to any shape and size, thus making it possible to mix scalars with vectors or arrays within expressions.

Fortran 90 reuses the VECTRAN/BSP Fortran `where` construct with some modifications. As in VECTRAN, `WHERE` operates in parallel on each of the elements of an object for which a specified condition holds. It is equivalent to an `IF` statement enclosed by a `DO` statement, and can therefore be thought of as a parallelized `IF`. An `ELSEWHERE` statement replaces the VECTRAN `otherwise`, and operates on the set of elements where the condition does not hold. It is analogous to a parallel `ELSE`.

Statements in the `WHERE/ELSEWHERE` blocks are restricted to array assignments and were required to conform to the shape of the tested object. The `WHERE` is typically used to avoid singular cases such as dividing by a zero-valued element. To minimize the overhead of tracking the set of enabled PEs, `WHERE`s cannot be nested.

Fortran 90 also reuses VECTRAN's triplet notation for referencing sections of arrays, and allows vector and arrays to be indexed using vector subscripts. These are used to select elements in an independent and variable manner. This allows the programmer to specify complex data movement and rearrangement such as replications, permutations, and gathers and scatters of the elements of a sparse array.

These notational capabilities can be used on either side of an assignment statement to reorganize data, and are typically mapped to interprocessor communications on the hardware. By assigning one array section to another, the data is effectively moved between PEs. Section assignments specified by triplet have regular communications patterns, while those specified by vector subscripting are equivalent to generalized permutations. The latter is a powerful feature that is only reasonable to use on architectures with generalized interconnection networks such as the routers found on the Connection Machines and MasPar systems.

Fortran 90 also has a large number of intrinsics which perform various array operations. These intrinsics include construction, transposition, multiplication, reductions, geometric location of elements with specific properties, and structure inquiries. A conditional MASK can be applied to some of these to limit the operation to a subset of elements.

Fortran 90's reductions include SUM, PRODUCT, MAXVAL, MINVAL, COUNT, ANY, and ALL. These can be applied across the rows of an array in any dimension to form an array of one less dimension, forming a scalar in the limiting case. Conversely, data can also be spread (replicated) along a new axis to expand an array by one dimension.

A limited amount of operator and intrinsic function overloading is possible, as are user-defined operators. These features let the programmer define short-hand notations for specific tasks, but can also make the source less understandable.

Fortran 90 is a large and complex language which has evolved to handle array processing on SIMD and MIMD architectures. However, SWAR architectures are not particularly well suited to multi-dimensional array processing. Thus, Fortran 90 is more complex than is necessary for a SWAR processing model.

A number of parallel variations on Fortran were developed concurrently with the Fortran 90 standard. These languages have features which are similar to those of Fortran 90. Often these were intended to match the (then proposed) standard. Because of their concurrent development, and because several dialects of C were developed at about the same time, it is difficult to determine which of these languages implemented which features first. We will not be concerned with this, but will introduce some of these languages and point out salient features regardless of their origins.

Fortran-Plus [194] was a high-level language for programming the AMT DAP. It had features that were later included in the then proposed Fortran 8X language. These included extensions and intrinsic routines intended to allow the programmer to easily take advantage of data parallelism.

As with DAP Fortran, parallel data types were limited to vectors and two-dimensional matrices. These were first-class objects, but were limited to the size of the DAP array. Later versions of the language were expected to allow arbitrarily-sized objects.

Fortran-Plus had selection operators which could conditionally operate on sections of a vector or matrix. This was similar to the proposed Fortran 8X standard. It also had a set of aggregate functions such as reductions which operated on both vectors and matrices.

CM Fortran [160, 131] was essentially Fortran 77 with Fortran 90 and Connection Machine-specific array extensions for specifying potential data parallelism. These extensions were automatically parallelized by the compiler for execution on the parallel unit of the Connection Machine.

Generally, CM Fortran source code could be divided into Fortran 77 code and parallel-extended code. Fortran 77 operations were executed on the front end system and applied to scalar data and arrays whose elements were only accessed individually. These data objects were stored on the front end. All other arrays were stored on the PE array, and were operated on in parallel by Fortran 90 and CM Fortran-specific operations.

CM Fortran allowed vector subscripting which was only reasonable to use because of the presence of the CM's router interconnect. The Fortran 90 `WHERE` construct was supported to allow parallel conditional execution. Also, a few CM-specific extensions were supported by the language including a `FORALL` [161] statement (which had been dropped from the Fortran 8X proposal) and various advanced array processing intrinsics.

The `FORALL` construct was similar to a `FOR` loop in which the iterations were known to be parallelizable. This allowed the programmer to explicitly indicate array assignments which could be parallelized and made it easier for the compiler to find and exploit this code. To ensure the independence of its iterations, the body of a `FORALL` loop was restricted to containing "...a single array assignment statement." [214]

Array elements to be operated on could be selected by value or position within the array. The `FORALL` was typically used for array initialization and elemental assignment, but it was also useful for performing various data movements such as scans and generalized permutations.

From the programmer's point of view, the elemental operations denoted by a `FORALL` executed simultaneously, although this was not necessarily the case. This guaranteed that elemental assignments which would overwrite a value used in another assignment would not destroy the old value before it was used. Thus, the programmer did not have to take extra steps to protect values from the execution of other iterations.

The `FORALL` was the equivalent of the VECTRAN `identify`, except that it avoided the aliasing step by combining the separate aliasing and assignment statements into a single construct. Syntactically, it was similar to IVTRAN's `DO FOR ALL` construct.

As in Fortran 90, intrinsic functions were modified to work in parallel on the elements of the object. Also, many of the Fortran 90 array intrinsics were implemented in CM Fortran including those for construction, location, manipulation, inquiry, and multiplication. Reduction intrinsics were also supported, but were extended by allowing them to be used with a `FORALL` to specify scans (parallel prefix operations) to be performed on the PE array.

A number of other intrinsics beyond those in Fortran 90 were available for performing a variety of transformations on vectors and arrays. These included several inquiry and location intrinsics, a `DIAGONAL` constructor which placed a vector in the diagonal of a matrix filled with an optionally specified value, and a `REPLICATE` which extended an array along one of its dimensions.

Compiler directives which controlled the layout of arrays in the memory of the PE array were also available. These allowed the programmer to attempt to optimize the placement of the data on the CM. A directive to allow the programmer to specify where common data should be stored was also provided.

MPF [170, 171] (MasPar Fortran) was a subset of Fortran 77 which included some of the array-handling extensions of Fortran 8X. It was intended to allow the programmer to write code in a familiar manner by hiding the details of the MasPar architecture. This made the compiler responsible for finding and automatically parallelizing operations on vector and array objects.

MPF implemented a subset of the proposed Fortran 8X standard. It treated vectors and arrays as first-class objects. It allowed array sections to be referenced and operated on using triplet notation or vector subscripts. It included the `WHERE/ELSEWHERE` construct for describing parallel conditionals. It also supported a subset of Fortran 8X's array intrinsics. Layout directives which allowed the programmer to specify how data was to be stored on the MasPar's DPU were also included.

Fortran D [215] was a post-Fortran 90 attempt to develop a portable, parallel version of Fortran that could replace the variety of dialects which were around at the time. These had been developed for various processing models and architectures including SIMD, MIMD, and vector systems. It was believed that they tended to expose the underlying model, and thus programs written in them were often hard to port to systems based on other models.

The important aspect of Fortran D was decomposition: separating a problem into a problem mapping and a machine mapping. The problem mapping was an expression of the inherent, target-independent parallelism of the problem. The machine mapping was an expression of how the problem was to be mapped onto the specific target architecture. Thus, the problem was decomposed into a portable, machine-independent part and a non-portable, machine-dependent part.

Fortran D operates at a higher level than this research is concerned with. The purpose of the current work is to develop a new SWAR processing model and related programming methods, while Fortran D was developed to promote the portability of Fortran code between multiple processing models.

High Performance Fortran [158] (HPF) is a later dialect of Fortran 90 with extensions intended to better support data-parallel processing, primarily on MIMD and SIMD computers with non-uniform memory access costs.

New directives, implemented as Fortran 90 comments, allow the programmer to suggest parallelization strategies or to make assertions about the program. An `INDEPENDENT` directive indicates that statements in a `DO` loop can be parallelized. An `ALIGN` directive indicates that an object should be co-located with another object. Also, `DISTRIBUTE` and `REDISTRIBUTE` directives allow the programmer to suggest data layouts.

Other additions include support for extrinsic functions which allow the programmer to tailor algorithms to the target system. Also, certain of Fortran 90's capabilities have been eliminated to remove associated problems.

As with Fortran D, HPF can be rejected for our purposes. HPF is basically Fortran 90 with a CM Fortran-style `FORALL` and some mark-up. The `FORALL` should not be necessary in a well-designed programming language compiled with a smart vectorizing compiler, and the work of the mark-up directives should be unnecessary in a SWAR environment and should probably be handled in some other manner.

Various vectorizing compilers for Fortran [154] included Cray CFT, Fujitsu Fortran 77, IBM VS Fortran, Alliant FX/8 Fortran and NEC SX Fortran. These were developed between about 1979 and 1987. As automatic vectorizors for standard Fortran, they did not add much in the way of interesting language constructs or programming concepts for parallel processing. Their purpose was to avoid doing this so that the programmer could reuse sequential Fortran code without change or, at most, with the addition of a few directives to provide the compiler with hints about how to vectorize parts of the code.

PASCAL-based Languages

PASCAL [216] is an ALGOL-based language that was designed as a portable teaching tool sometime around 1971. This was done by compiling the source to a simplified, portable intermediate language called P-code [217, 218], then using an interpreter to execute this code on the target machine. This method was very successful. In fact, the highly portable JAVA [219] uses a remarkably similar method to obtain its portability.

Because of its portability, PASCAL became widely used and well-known, and has influenced a number of later languages. This ubiquity makes it a reasonable choice as a basis for parallel programming languages. One parallel language that was based on PASCAL was Actus.

Actus [153] was a SIMD-parallel language developed just after NX Fortran and Vector LRLTRAN and at about the same time as VECTRAN and BSP Fortran. It was a structured language intended to provide a target-independent programming model for vector and array processors which allowed for the direct, natural expression of data parallelism. Actus was originally targeted to the ILLIAC IV using a PASCAL P-code compiler.

In Actus, the maximum parallelism width that could be applied to an array or vector was specified upon declaration of the object. This was done using a dimension notation in which the starting and ending indices along one dimension of the object were separated with a colon instead of a pair of periods. This indicated both the maximum "extent of parallelism" and the dimension across which it should be applied.

For example, the declaration `var a: array[1:4, 1..5] of integer;` would declare `a` to be a two-dimensional array and indicate that it should be arranged in memory so that accesses across its first dimension could occur in parallel. The maximum extent of parallelism for this array would be four (the length of its first dimension).

The extent of parallelism to be applied for a particular access could also be explicitly specified when that access occurred. This allowed subvectors and subarrays to be described and operated on in a parallel fashion. Suppose that the array `a` above was accessed as `a[2:3, 1]` in an expression. This would indicate that the elements `a[2, 1]` and `a[3, 1]` should be accessed in parallel. The extent of parallelism thus controlled the enabling of PEs which held selected elements.

Actus was one of the first languages to allow vector subscripting. It also allowed named "parallel constants" which were a set of strided values that were defined using a notation similar to that of VECTRAN triplets. These could be used as array indices or as initial values for vectors. They had the form: `const` *id* `=` *start*`:(`*stride*`)`*finish*, where the stride was optional and defaulted to one.

Actus introduced a general form of index sets which were similar to parallel constants. These allowed the programmer to specify a set of indices that would be involved in an operation. For example, the code `index ind = 1:10, 11:(2)99;` created an index set containing all values from 1 to 10 and the odd values from 11 to 99. The identifier `ind` could then be used to indicate the indices involved in a particular operation.

These sets could be operated on using set operators to obtain their union, intersection, difference, or complements. Vector shift and rotate operations could also be applied to index sets or to explicit extents of parallelism. For example, the code segments `a[1:10 shift 2];` and `index idx=1:10 shift 2; a[idx];` each represented the first ten elements of the vector `a` shifted by two positions.

To allow an extent of parallelism to be reused within a section of code without forcing the programmer to repeatedly supply the same information, Actus had a `within` construct which defined an extent of parallelism to be used throughout its body. Within the body, the current extent of parallelism was represented by a pound sign (#).

Like GLYPNIR, Actus had parallelized `if`, `while`, and `for` constructs and `any` and `all` tests that were equivalent to its `SOME` and `EVERY` tests. It also had a parallelized `case` statement which embodied multiple jump targets given a single conditional test.

Actus also allowed vectors to be passed to functions and procedures as arguments and used as return values from functions.

While Actus allowed virtualized vector and array dimensions (i.e. dimensions that did not match the underlying architecture), it only allowed standard data precisions. As a language which allowed and promoted the use of multidimensional arrays, it is not a good match for current SWAR architectures which are all one-dimensional. Under the assumption that future SWAR architectures will be multi-dimensional; that is, something more akin to the GAPP or BLITZEN processors, they may benefit from an Actus-like programming model.

C-based Languages

The C programming language was developed in the mid 1970s by Dennis Ritchie and others at AT&T Bell Laboratories [220, 221]. It was co-developed with the UNIX operating system and was its primary source language.

C is a well-defined language that is useful for writing portable applications code. Its real strength, however, lies in its low-level nature. This allows the programmer a high degree of flexibility and access to the target system.

Because of the wide-spread use of UNIX on high-performance, multi-user systems, most of these systems have a working C compiler available to their programmers. Because of this ubiquity and its power, C has become a favorite of systems-level programmers, and the basis for several parallel programming languages. Among these are PASM Parallel-C, C*, and MPL. The language developed as part of this research, SWARC, is also based on C.

PASM Parallel-C [222] was developed for the dynamically reconfigurable PASM. It allowed any data type to be parallelized, and treated objects of these types as first-class entities.

Conditional tests such as `if` statements were modified for use with parallel expressions, and a *selector* type was added which allowed subarrays to be specified.

Assignment of parallel data objects used a syntax similar to that of C and operated in an element-wise fashion. Mixed-sized parallel assignments were allowed, but were executed only for corresponding elements. Parallel to scalar assignments were not allowed, so the the programmer was required to convert the parallel data to a single value. This was done by using the value of a single selected element from the parallel object. No reduction operations or reduction-assignments were available in the language.

Because PASM could be partitioned into sections which used SIMD and MIMD modes simultaneously, the Parallel-C language was primarily geared toward allowing this type of usage. Later languages were targeted to more SWAR-like architectures.

C* (pronounced C-star) was an extension of the C language intended to help the programmer exploit data parallelism on the SIMD Connection Machine. There were actually two major versions of C*. One was introduced in the mid-1980s and was modified slightly soon afterward. A second was introduced around 1990 which was significantly different from the earlier versions. It is instructive to look at each of these.

The original version of C* [166] was developed for use on the CM-1. It used two storage class identifiers to explicitly indicate parallel versus scalar data, and implicitly indicate where in the system a data object would reside. `mono` objects were scalars that were placed in the memory of the host computer and were typically operated on there. `poly` objects were placed in the memories of the processors in the parallel array, and were operated on in parallel.

PEs were represented in the language via the `processor` type. The programmer could declare an array of `processors` to represent a subset of the available PEs where a parallel data object would reside. By declaring different `processor` objects, the programmer could create different sets of PEs to hold different data sets.

C* had a selection statement modifier which allowed the programmer to choose an "active set" of PEs to be enabled during the execution of the modified statement. Upon completion of this statement, the PEs were returned to their previous enable state.

The format of this selection statement was `[selector].statement`. The selector could be a `processor` variable, an array of `processors`, an indexed value representing a consecutive series of `processors`, or a list of any of the above. This allowed any subset of processors to be chosen at any time to execute a statement, thus providing a great deal of flexibility.

The standard C control constructs retained their C syntax, but were modified semantically to match the SIMD processing model using active sets. These were split and recombined as necessary to handle conditional execution. The bodies of `if`, `else`, and `while` statements were only executed if, and while, the test condition held for at least one active PE. This was later called the "rule of local support". Nested constructs were allowed. These recursively divided the set of active PEs into smaller sets which recombined as each level of nesting completed. Once a construct was completed, the active set before it was entered was restored.

The language supported the full set of standard C operators including its various assignment operators. New operators were also included to represent the minimum (<>) and maximum (><) binary operations. These operators provided a concise means of denoting these often used operations, and could represent scalar or parallel operations depending on the types of their operands. These operations could also be combined with assignment to form comparison-assignment operators.

Purely `mono` expressions were executed as in C, but `poly` and mixed expressions required the semantics of the standard C operators to be modified for use with the SIMD model. Both `poly` and mixed expressions were required to follow the "as-if-serial" rule. This stated that the result was determined as if the parallel parts had been executed in some undetermined serial order.

In mixed expressions, `mono` values were promoted to `poly`s as needed via replication. Assignment of a `mono` value to a `poly` object implied replication of the value to each of the members of the active set. Assignment of a `poly` to a `mono` implied some form of reduction operation to form the single assigned value.

The standard C assignment operators, and those formed from the minimum and maximum operators, could be used for both assignment and unary reduction. When used as assignments, they acted as described above. When used as unary reductions, the result was a `mono` value which could be used in an expression.

Under the as-if-serial rule, reductions were performed as if the elemental assignments occurred in some unspecified serial order. This ensured that reduction-assignments to a `mono` object resulted in the correct value being stored without the loss of any parts of the reduction.

C* had a `this` keyword which could be used in `poly` expressions to represent the currently executing `processor`. It could be dereferenced to access data on the local processor; but more importantly, it could also be indexed to access data on another processor, thus allowing a form of interprocessor communication.

Daniel Hillis' dissertation [128] describes the theory behind the use of the Connection Machine. It was based on mapping data onto the PE array in any of several representations called *xectors*. Xectors were domain/range pairings of the indexed PEs with values determined by applying a function to these indices. The original C* language was modified to incorporate the domain concept soon after its introduction.

The **modified C* language**, described in [167] and [168], included a C++ class-like construct called a *domain*. Each instance of a domain represented data residing on a single PE. An array of some domain represented a set of data (i.e. a xector) which was distributed with one element per virtual processor. Each processor on which an instance resided was said to belong to the domain. Using this concept, the PEs could be divided into groups for performing different tasks on different sets of data.

Similar to classes in an object-oriented language, domains consisted of a data structure and a set of functions which could access it. The data structure described the xector data and its layout in the memory of each of the PEs on which it resided. These PEs were said to "belong" to the domain.

A domain's data elements were treated as first-class objects. A references to any of them referred to the entire set of same-named elements across all of the instances of the domain. This allowed the programmer to specify an entire parallel data object concisely.

Parallel execution was performed by calling the member functions of the domain related to the xector to be operated on. These functions were executed simultaneously across all the PEs belonging to the domain. Thus, domains were used to specify the active set of PEs as used in the original version of the language. A domain's member functions could only be called on a particular PE if that PE belonged to the domain. This ensured that the processor had the correct data layout for the called function.

For this version of C*, the meanings of `mono` and `poly` were modified slightly to work with domains. `mono` domain members were scalars stored on the front-end, while `poly` members were allocated across the PEs belonging to the domain.

Other changes included the replacement of the minimum and maximum operators with (<?) and (>?), respectively, and the addition of a (,=) assignment operator which indicated that a single, arbitrary element should be chosen as the result.

The use of the selection statement was modified to activate the processors belonging to a particular domain for a single statement (which could be a block). This was done by modifying the format of a selection to: `[domain tag].statement`. The effect of this change was to make selection less flexible, thus making it harder for the programmer to violate the semantics of the language's control structures.

Selection deactivated the current active domain before selecting the new one, and reactivated it once the statement completed. Indexing could still be done with selection, and the `this` keyword had the same meaning, except that the index referred to a PE in the active domain. Selection could also be used to initiate parallel execution from within serial code.

The programmer could still do something along the lines of the original C*'s selection statement using a dot operator. This was interpreted by evaluating the left-hand side as an lvalue which specified a set of PEs. These PEs would evaluate the right-hand side based on the type of the left-hand side. If the right-hand side evaluated to a value, it was used as the value of the dot operation. In this sense, C*'s selection statement was an extension of its dot operator.

Function overloading was available and allowed multiple variations of same-named functions to be written for various combinations of `mono` and `poly` parameter and return types. Resolution was done using an algorithm which tried to find the best match between the argument and return types of the call and the parameter and return types of the available functions. C* also had a `typeof` keyword which was used to allow function parameters to be polymorphous.

This version of C* allowed interprocessor communication to be denoted concisely. As in the original C*, the `this` keyword could be used with the dot operator to denote interprocessor communication between a PE and its neighbors. For example, `x=(this+1)->x;` sets the local PE's value of `x` to that of its nearest neighbor's `x`. Similarly, C* pointers could be used to denote communications between the processors in a domain. This was accomplished by simply pointing at an object in another processor's memory. This notation supported permutations, multiple parallel broadcasts, and multiple parallel reductions.

Around 1990, after about three years of use in this second form, the language was again redesigned [169, 131]. This **second major version of C*** was somewhat cleaner than the previous two, and was based on the concept of data "shapes".

Shapes were used to specify multi-dimensional spaces on a virtual PE array. Once specified, these shapes could be associated with data objects as part of their declaration. The syntax for a shape specification was similar to that of a multi-dimensional array declaration, with the size of each dimension specified by the number of positions along its axis. This allowed a shape to be described concisely and easily applied to multiple data objects.

Shaped objects could be simple variables, arrays, `struct`s, or any other C type construct. Pointers to shapes were also available, and shapes could be passed between functions. Thus, the new C* provided a significant level of flexibility in dealing with objects of different sizes and dimensions.

One aspect of this version of C* was the concept of a "current shape". This was specified using a `with` statement. In general, objects had to be of the current shape in order to be operated on in parallel. The addition of `with` allowed multiple layouts to be specified and used within a single program. This allowed parallel data objects to be independent of not only the architecture, but also of other parallel objects.

Parallelism was expressed in terms of the positions in a data shape that were to be acted on. A `where` statement, similar to that of VECTRAN, allowed the set of active data positions to be conditionally determined. This was referred to as "setting the context". The standard C constructs were modified to work with the `where` statement to provide conditional execution. These included the `else` statement, which was modified to activate the set of positions opposite to that of the `where`.

An `everywhere` construct was also added to allow all positions, active or not, to be enabled for the execution of an embodied statement. Nested `where`s operated as expected, possibly making the set of active positions smaller as each was entered, and returning to the previous set as each exited.

Functions could take parallel objects as arguments and also return them. They could be written with the shape of their parameters explicitly specified or left unspecified, in which case the `current` shape would be used during the call. As with earlier versions of C*, overloading could be used to specify multiple functions with the same name but various parameter shapes.

C*'s expression syntax was made concise through the use of operator overloading. Overloading allowed the standard C operators to be used on shaped objects in a first-class manner. Operations on these objects could then by parallelized and modified with replications or reductions as necessary. Thus, C* shapes were similar to object-oriented classes with overloaded operators.

The `this` keyword was replaced by the `pcoord` intrinsic function which returned an identifier for the current data element along a specified axis. This could be used in a manner similar to `this`, allowing regular communication along one axis of the data structure.

"Left indexing" was used with assignments to access data in irregular parallel patterns. Indexing a parallel object on the right-hand side was equivalent to performing a "get" operation. In this case, the operation assigned the instance of the parallel object on the indexed virtual PE to the left-hand side. Indexing a parallel object on the left-hand side was equivalent to performing a "send" operation. In this case, the operation assigned the value on the right-hand side to the instance of the parallel object on the virtual PE indexed on the left. These operations allowed generalized communication to be described using a syntax similar to that of element access and assignment.

This version of C* also differed from the previous versions by the inclusion of a `bool` Boolean type. This type closely matched the bitwise architecture of the parallel array, and allowed the programmer to make use of this aspect of the system more easily than the previous versions of C* allowed.

Obviously, C* was changed significantly over time as experience was gained with its use. The original version focused on the PEs as the parallel entities whose activity needed to be described and controlled. This was replaced by the second version, which focused more on describing the data sets to be parallelized. This version was more complex in its handling of selection and domains, but more closely matched Hillis' thesis. Both of these exposed the virtual processor array via selection, a mechansim which was promoted for communications purposes.

These versions of C* can be rejected as the basis for a SWAR model, just as they were ultimately rejected by Thinking Machines. In each case, the language was a mix of a data-oriented programming model and one with explicit control over PE selection and inter-virtual PE communications. This made each of these languages more complex than necessary. While these versions of C* could be thought of as failures, certain of their aspects were very well-designed and deserve to be remembered by anyone trying to design a new parallel programming model.

The last version of C* was, semantically, the cleanest of the three. Processor selection was limited to the conditional `where` and the unconditional `everywhere` constructs. Few new constructs were added beyond those of the C language, and the semantics of the C operators were extended to handle parallelism through operator overloading. This version of the language also allowed the programmer to focus on describing the data sets, and the operations to be performed on them, rather than on the control of parallel execution.

This last version of C* might be a good choice for the basis of a SWAR programming model. However, "shapes" are more useful for multi-dimensional data structures than for the vectors which more closely fit the SWAR model. Thus, a SWAR model should probably avoid C*-like shapes. Also, special syntax and intrinsic functions are probably unnecessary for communications in a SWAR environment – simple element accesses and assignments should suffice.

A **generic fine-grained parallel C** [178] was developed by scientists at NASA's Goddard Space Flight Center in the late 1980s. It was intended to be a common interface language to multiple types of architectures including serial processors [178].

At the time [178] was published, the language was only partially implemented for the Apple Macintosh II — a serial processor.

This language extended C with a `parallel` storage class which indicated that the declared object was multi-valued. Conceptually, parallel objects were stored in a parallel memory and serial objects stored in a separate serial memory. When mapped to a target architecture, these memories may or may not have been separate.

As with previous parallel languages, the standard C operators were extended to operate on parallel objects. Arithmetic operators were extended to perform in an elementwise manner. Bit shifts were implemented such that shifting by a parallel value resulted in each element being shifted by a (possibly) different number of bits. Logical operators were implemented using a parallel `if-else` structure, apparently to maintain the short-circuit semantics of C's logical operators. Mixed expressions were allowed, with scalar values replicated to match the dimensions of parallel objects.

Mixed assignments were also allowed. Assignment of a parallel value to a scalar object resulted in a reduce-OR of the parallel elements, while scalar to parallel assignments resulted in replication of the scalar. C's assignment operators were also parallelized with reduction or replication of values taking place as necessary.

The C control constructs, `if`, `while`, `for`, and `switch` were modified for use with parallel conditionals. If the conditional was a parallel expression, each body would be executed if the condition held for at least one element. Each `case` in a `switch` was executed only if at least one element was directed to that `case`.

Parallel pointers were disallowed, but serial pointers to parallel objects were legal. Arithmetic on these pointers could be used to denote interprocessor communication by shifting values between elements. Thus, the language hid communication behind its normal syntax.

This language also allowed all variables, including parallel objects, to be assigned a bit size. This was primarily intended for use with bit-sliced target architectures, such as the MPP, which allowed variable data lengths. It is unclear from [178] if this feature allowed all bit sizes to be applied. To ease portability to more restrictive architectures, the compiler was allowed to use larger bit sizes than were specified in the program source. Given the assertion that a general-purpose SWAR model should support any data precision, this aspect of the language deserves further examination.

This language had several interesting features that may be of value for a SWAR-based programming language. Unfortunately, I could find no further references to this language, so it is probably safe to assume that it either was abandoned or evolved into another language. The SWARC language described in this thesis has some similarities to this language, but is more fully developed.

MPL [107, 170, 171], the MasPar Programming Language, was another SIMD variant of C developed around 1990. Semantically, it was similar enough to C to allow it to be compiled with a simple variation of the GNU C Compiler (GCC). MPL was also known as the MasPar Parallel Application Language.

To allow the programmer to specify data parallel algorithms, a `plural` type modifier was used which indicated that the object was multi-valued and distributed across the PE array. An operation on a `plural` object was executed simultaneously on the enabled PEs and resulted in another `plural` object. This allowed the programmer to specify data parallel operations in a manner semantically similar to C.

A scalar data object in MPL was referred to as a *single*. These objects had one value and resided on the MasPar's ACU. Operations on single objects took place in the ACU and resulted in single values. This allowed the programmer to specify scalar operations simply, again using C-like semantics.

MPL also allowed mixed-mode operations and assignments, with reductions and replications performed as necessary. As with C*, the semantics of control constructs such as `while` loops and `if` statements were modified for proper operation under the SIMD processing model.

MPL allowed for synchronous inter-PE communication via the addition of three new constructs: `proc`, `router`, and `xnet`. These allowed non-local data to be accessed by the PEs. They also allowed expressions to be executed where their operands resided, with only their results passed over the interconnect. Using these constructs, communication occurred synchronously with all active PEs sending and receiving data on the same instruction.

The `proc[`*ex1*`].`*ex2* construct allowed the programmer to specify the execution of an expression, *ex2* on a single PE chosen by another expression, *ex1*. In the simplest case, this allowed the extraction of elements from `plural` objects.

The `router[`*ex1*`].`*ex2* construct was a `plural` operation in which the result on each of the PEs was the result of evaluating expression *ex2* on PE number *ex1* with communication occurring over the three-stage router network. The expression *ex1* was a `plural` object. This allowed independently indexed communications to be specified.

Similarly, the `xnet`*dir*`[`*ex1*`].`*ex2* construct was a `plural` operation in which the result on each of the PEs was the result of evaluating expression *ex2* on the PE which is *ex1* steps away in direction *dir* with communication occurring over the Xnet. The expression *ex1* was a single value; thus, all PEs executed the same communications pattern.

While the names of these constructs are taken directly from the MP-1's major interconnection networks, they are really more generally applicable. For example, the PE numbering used in the router construct is linear, but these numbers may be mapped onto an N-dimensional array where N is any non-negative integer. Also, the `xnet` construct could be mapped to smaller-dimensional PE arrays by ignoring dimensions, or to larger-dimensional ones by adding new directions.

MPL code was callable from other languages used on MasPar systems to ease code migration to the parallel model. This allowed the programmer to incrementally rewrite existing code to take advantage of the parallel architecture.

MPL was well-designed and semantically clean. It allowed the programmer to express parallelism and operations such as reductions in a manner which did not expose the properties of the underlying architecture. It also allowed communications using language constructs that were applicable to other types of architectures. MPL would be a good choice for the basis of a SWAR programming model, with the caveat that most SWAR architectures cannot easily support its highly-generalized communications constructs.

C[ ] (C brackets) [173], developed in the early 1990s, is an extension of the ANSI C language. It was intended to allow the programmer to write efficient code that was portable between the SIMD architectures then available without incorporating non-portable features.

C[ ] is vector-based, treating vectors as first-class objects with a declarable fixed stride between elements. Multidimensional arrays are allowed, and are treated as vectors of vectors. This is an approach that may work well for allowing array-based processing on vector-based SWAR architectures.

C[ ] was defined in a manner that ensured that pointer arithmetic has a consistent interpretation which followed the basic intent of the then current ANSI C standard. Accesses of vector and array elements obey an arithmetic which takes the declared stride into account. Subarrays can be specified using either pointer arithmetic or a notation similar to C's array indexing.

C[ ] extends the C language's bit fields by allowing vectors of these to be assigned values via a gather operation on an integer vector of fixed stride. However, it appears that this is the only first-class operation allowed on bit field vectors, and that the language does not allow SWAR-like operations to be performed on them.

Along with the standard C operators, C*-like scalar maximum (?>) and minimum (?<) operators were included in C[ ], as were operators for bitwise population (?), leading zero count (%) and word reversal (@). Unary operators can be applied to vectors and operate in elementwise fashion, while binary operators can operate on vector or mixed operands. These same operations can be performed as unary reductions using a set of "unary linear operators" which are denoted by enclosing the corresponding C operator in a bracket pair. For example reductive addition is denoted by the symbol `[+]`.

Vectors could be converted in length or type via casting or on assignment, but vector to scalar conversions were not allowed. Binary operations between vectors of differing lengths had undefined results. Vectors and arrays could also be passed to functions as first-class objects and return values could be of vector type. All of the attributes of a vector or array parameter were required to be defined as part of the function's formal declaration. Thus, functions using these parameters could not be written to accept objects of some other size without resorting to pointer arithmetic.

The goals of C[ ] are similar to those of the SWAR model of processing, but the language was intended to provide efficiency and portability at the level of array and vector processing of standard data types. While not the best model for the current set of SWAR architectures, this language has features that may be useful in future SWAR-like languages targeting array-based architectures.

Other Languages

There are a few other languages that are worth mentioning because they have some feature or features which are related to SWAR processing; however, for various reasons, are not languages that we wish to model.

PL/I [223, 224, 225] was developed in the mid-1960s and was originally intended to be an update of FORTRAN IV that was referred to by the name of FORTRAN VI. After it was decided that it would be incompatible with FORTRAN IV, the name NPL (New Programming Language) was given to it. This name happened to conflict with the name of a laboratory in England, so the name of the language was finally changed to PL/I.

PL/I allowed the programmer to specify arbitrary precisions to be used for storing individual data objects "by declaring the total number of digits and the number of digits to the right of the decimal (or binary) point." [225] This allowed the programmer to specify data precisions that closely matched those of the application. The compiler could then attempt to preserve precision when possible. As a practical matter, using precisions that differed significantly from those supported by IBM's S/360 series of computers, PL/I's primary target, resulted in unexpected results and were thus rarely used.

AJL (Anar Jhaveri's Language) [174] was developed around 1990 and was intended to provide a simple vector programming model which could be easily ported to various target architectures. AJL was a calculator language which provided basic arithmetic and trigonometric operations and functions on either scalar (`mono`) or vector (`poly`) objects. It was similar in certain respects to both C and Pascal.

Arithmetic operations provided by AJL included addition, subtraction, negation, multiplication, division, and power. Intrinsic functions included sine, cosine, tangent, floor, and ceiling. Mixed expressions were allowed for some of these operations and functions, but each could be applied to purely scalar or vector expressions.

A set of predefined values was also provided, including `pi` and `e`, and a shorthand for the number of elements in any vector (`#`). A set of intrinsic functions were also included which provided limited support of the input and output of scalar values.

AJL provided operations related to layout and rearrangement of vector data. These included vector value definitions (i.e. the ability to assign the values of a vector's elements from a list), generation of linearly ranging vectors, left and right vector shifts, shuffle, and inverse shuffle.

Only a "less than" comparison operator was available in the language. It operated on either scalars or on vectors in an element-wise fashion. A C-like trinary operator was also provided which operated on vectors by element.

Source code written in AJL was translated into a pseudo-assembly language for a non-existent stack-based machine. This code was actually a list of macros which were then converted into native C code for the target machine. Thus, porting AJL-compiled code consisted of defining the pseudo-assembly macros for the new target. This method of translation allowed AJL to be very portable and to take advantage of the optimization capabilities of the native C compiler.

AJL was a limited language which dealt neither with vectors of unequal lengths nor with vector element precisions. However, many of its features are useful for developing a new SWAR programming language, and lessons learned in its development can be applied to the development of a new SWAR language. In fact, the language developed as part of this thesis has some similarities to AJL, but AJL itself is not particularly suited to SWAR processing.

NESL/VCODE/CVL NESL [175] is a "nested data-parallel language". This means that it allows data to be described using recursive data structures and allows operations to be applied to sets of data described by these structures. Its primary benefit is the description of irregular data sets. Like APL, NESL differs significantly from the programming languages which are most commonly used in the high-performance computing community.

NESL is built on top of the stack-based VCODE vector language [176]. VCODE allows operations on the primitive data types: `int`, `bool`, `float`, `char`, and `segdes`, where `segdes` "specifies a partitioning of one of more vectors into segments." The language allows basic arithmetic operations, conditional tests, intra-element shifts, logical operations, and conversions. It also allows higher-level mathematics such as exponentials and trigonometric functions. A limited set of reductions and scans are also available. Various operations allow data manipulation such as permutations, extractions, and packing. Operations for manipulating the stack and performing I/O are also included.

While VCODE allows a large range of useful functions which can be included in a SWAR model, it is a stack language for a rather powerful, theoretical machine. As such, it does not match the current set of multimedia-enhanced targets very well.

VCODE itself is built on top of CVL [177], a low-level vector library for the C language. CVL functions include elementwise operations, reductions, scans, permutations, vector-scalar conversions, management, and some higher-level functions.

CVL functions operate on an area of memory set aside exclusively for the storage of vectors. Vectors are laid-out within this memory in an implementation-dependent manner. Vector elements may be stored in larger than necessary locations in memory in order to simplify processing or provide portability.

Functions are passed a "handle" for each of their vector operands. This handle may be a pointer or a more complex structure which indicates the position and layout of the vector. Functions must also be passed the length of their vector operands and, in some cases, a handle to a previously allocated scratch space in vector memory.

CVL's functions operate on vectors of type `int` and `double`, which have native precision, and `cvl bool` which may be stored in any useful form such as `char`s or bits. A vector may be *segmented*, meaning that it is actually a collection of smaller vectors, or *unsegmented* which means that it consists of a single vector (i.e. it has one segment). Operations performed on a segmented vector are applied to each segment independently.

CVL was intended to provide portability between massively-parallel processors such as the Connection Machines CM-2 and MasPar MP-1. It was not intended to provide functionality for non-standard data precisions. CVL's use of a private vector memory allows vectors to be laid-out in the most efficient manner without regards to issues such as pointer arithmetic although it provides similar functionality via vector handles.

CVL provides much of the functionality that one would hope to have in a good SWAR model. However, it is limited to standard data precisions and provides certain functionality, such as trigonometric functions, which should not be included in a general-purpose SWAR model.

APPENDIX B
SUPPORTED SWAR EXTENSIONS IN COMMODITY CPUS

Table B.1
Supported SWAR Extensions in Commodity CPUs

Processor Name	Year [1]	Aliases	MVI	MAX-1	MAX-2	MIPS-V	MDMX
DEC Alpha 21264 [60]	1997?	EV6	Yes	-	-	-	-
DEC Alpha 21164PC [226]	1997	PCA56	Yes	-	-	-	-
DEC Alpha 21164A	1995?	EV56	-	-	-	-	-
DEC Alpha 21164 [227]	1994	EV5	-	-	-	-	-
HP PA-8000 [84]	1996		-	Yes	Yes	-	-
HP PA-7100LC [61]	1994		-	Yes	-	-	-
MIPS MIPS64 [87]	1999		-	-	-	Optional	-
MIPS H1 Arch. [85][2]	1999?		-	-	-	Yes	Yes
MIPS R12000 [85]	1998?		-	-	-	-	-
MIPS R10000 [66]	1994?		-	-	-	-	-
Motorola MPC7400 [89]	1999	G4	-	-	-	-	-
Sun UltraSparc III Cu [92]	2001		-	-	-	-	-
Sun UltraSparc III [92]	2000		-	-	-	-	-
Sun UltraSparc II [202, 91]	1996?		-	-	-	-	-
Sun UltraSparc I [202, 228]	1995		-	-	-	-	-
Intel Pentium 4 [229]	2000	Willamette	-	-	-	-	-
Intel Pentium III [229]	1999	Katmai	-	-	-	-	-
Intel Pentium II [229]	1997		-	-	-	-	-
Intel Pentium w/MMX [229]	1996		-	-	-	-	-
Intel Pentium Pro [229]	1995		-	-	-	-	-
Intel Pentium [229]	1993	80586	-	-	-	-	-
AMD Athlon XP [99]	2002	Thoroughbred	-	-	-	-	-
AMD Athlon MP [230]	2001	Palomino	-	-	-	-	-
AMD Athlon 4 [98]	2001	Palomino	-	-	-	-	-
AMD Athlon [76]	1999	K7	-	-	-	-	-
AMD K6-III [75]	1999		-	-	-	-	-
AMD K6-2 [75]	1998	Model 8	-	-	-	-	-
AMD K6 [73]	1996	Models 6-7	-	-	-	-	-
VIA C3 [103]	2000	Cyrix MIII	-	-	-	-	-
Cyrix MXi [231][2]	1998?	Cayenne	-	-	-	-	-
Cyrix M-II [232, 77]	?	M2	-	-	-	-	-
Cyrix MediaGXm [100]	?		-	-	-	-	-
Cyrix 6x86Mx [100]	1997		-	-	-	-	-
Cyrix MediaGX [100]	?		-	-	-	-	-
Cyrix 6x86 [100]	?		-	-	-	-	-

[1] Approximate year of introduction or implementation.
[2] I'm not sure that this was ever implemented.

Table B.1 cont'd.
Supported SWAR Extensions in Commodity CPUs

Processor Name	AltiVec	VIS	MMX	SSE	SSE2
DEC Alpha 21264	-	-	-	-	-
DEC Alpha 21164PC	-	-	-	-	-
DEC Alpha 21164A	-	-	-	-	-
DEC Alpha 21164	-	-	-	-	-
HP PA-8000	-	-	-	-	-
HP PA-7100LC	-	-	-	-	-
MIPS MIPS64	-	-	-	-	-
MIPS H1 Arch.	-	-	-	-	-
MIPS R12000	-	-	-	-	-
MIPS R10000	-	-	-	-	-
Motorola MPC7400	Yes	-	-	-	-
Sun UltraSparc III Cu	-	2.0	-	-	-
Sun UltraSparc III	-	2.0	-	-	-
Sun UltraSparc II	-	1.0	-	-	-
Sun UltraSparc I	-	1.0	-	-	-
Intel Pentium 4	-	-	Yes	Yes	Yes
Intel Pentium III	-	-	Yes	Yes	-
Intel Pentium II	-	-	Yes	-	-
Intel Pentium w/MMX	-	-	Yes	-	-
Intel Pentium Pro	-	-	-	-	-
Intel Pentium	-	-	-	-	-
AMD Athlon XP	-	-	Yes	-	-
AMD Athlon MP	-	-	Yes	-	-
AMD Athlon 4	-	-	Yes	-	-
AMD Athlon	-	-	Yes	-	-
AMD K6-III	-	-	Yes	-	-
AMD K6-2	-	-	Yes	-	-
AMD K6	-	-	Yes	-	-
VIA C3	-	-	Yes	-	-
Cyrix MXi	-	-	Yes	-	-
Cyrix M-II	-	-	Yes	-	-
Cyrix MediaGXm	-	-	Yes	-	-
Cyrix 6x86Mx	-	-	Yes	-	-
Cyrix MediaGX	-	-	-	-	-
Cyrix 6x86	-	-	-	-	-

Table B.1 cont'd.
Supported SWAR Extensions in Commodity CPUs

Processor Name	3DNow!	E3DNow!	3DNow!Pro	EMMX	MMFP
DEC Alpha 21264	-	-	-	-	-
DEC Alpha 21164PC	-	-	-	-	-
DEC Alpha 21164A	-	-	-	-	-
DEC Alpha 21164	-	-	-	-	-
HP PA-8000	-	-	-	-	-
HP PA-7100LC	-	-	-	-	-
MIPS MIPS64	-	-	-	-	-
MIPS H1 Arch.	-	-	-	-	-
MIPS R12000	-	-	-	-	-
MIPS R10000	-	-	-	-	-
Motorola MPC7400	-	-	-	-	-
Sun UltraSparc III Cu	-	-	-	-	-
Sun UltraSparc III	-	-	-	-	-
Sun UltraSparc II	-	-	-	-	-
Sun UltraSparc I	-	-	-	-	-
Intel Pentium 4	-	-	-	-	-
Intel Pentium III	-	-	-	-	-
Intel Pentium II	-	-	-	-	-
Intel Pentium w/MMX	-	-	-	-	-
Intel Pentium Pro	-	-	-	-	-
Intel Pentium	-	-	-	-	-
AMD Athlon XP	Yes	Yes	Yes	-	-
AMD Athlon MP	Yes	Yes	Yes	-	-
AMD Athlon 4	Yes	Yes	Yes [1]	-	-
AMD Athlon	Yes	Yes	-	-	-
AMD K6-III	Yes	-	-	-	-
AMD K6-2	Yes	-	-	-	-
AMD K6	-	-	-	-	-
VIA C3	Yes	-	-	-	-
Cyrix MXi	-	-	-	-	Yes
Cyrix M-II	-	-	-	Yes	-
Cyrix MediaGXm	-	-	-	Yes	-
Cyrix 6x86Mx	-	-	-	-	-
Cyrix MediaGX	-	-	-	-	-
Cyrix 6x86	-	-	-	-	-

[1] Available on later models.

APPENDIX C

SWAR INSTRUCTION MNEMONICS

The following tables show the instruction mnemonics for the SWAR multimedia support tabulated in section 2.1. Except for table C.1, each table corresponds to the table in section 2.1 with the same number.

Table C.1
Comparison of Multimedia Instruction Set Extensions

Architectural Feature	DEC MVI	HP MAX-1.0	HP MAX-2.0	SGI MIPS-V	SGI MDMX
Typical Processor	Alpha 21164PC	PA-7100LC	PA-8000	H1 Arch.	H1 Arch.
# MM Pipelines[1]	2[226]	2 ALUs [63]	2 ALUs, 2 SMUs [63] [2]	Unknown	Unknown
Year Announced [233]	1996	1993	1995	1996	1996
Year Shipped [233]	1997	1994	1996	1999	1999?

Architectural Feature	Motorola AltiVec	Sun VIS	Intel, AMD MMX	Intel SSE
Typical Processor	MPC7400	UltraSparcI	Pentium w/MMX	Pentium III
# MM Pipelines[1]	1 ALU, 1 VPU [89] [3]	2 in GRU [90][4]	2 (U and V) [234]	2?
Year Announced	1998	1994	1996	1998?
Year Shipped	1999	1995	1996	1999

Architectural Feature	Intel SSE2	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Typical Processor	Pentium4	K6-2	Athlon	Athlon XP	M-II
# MM Pipelines[1]	2?	2 (X and Y)	2 (excluding L/S) [235]	2 [99]	1? [232]
Year Announced	1999?	1997?	1998?	2001	1997
Year Shipped	2000	1998	1999	2002	?

[1]Independent pipelines may not necessarily be equivalent.
[2]SMU=Shift Multiply Unit.
[3]VPU=Vector Permute Unit.
[4]GRU=Graphics Unit

Table C.2
SWAR Addition Operations

Operation Types	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX	Motorola AltiVec
Modular Addition						
Part/Part	-	hadd	- hadd	add.ps	-	vaddubm, vadduhm, vadduwm
Immd/Part	-	-	-	-	-	-
Part/Part w/Acc (w/ or w/o Init)	-	-	-	-	add[la].ob, add[la].qh	-
Scalar/Part w/Acc (w/ or w/o Init)	-	-	-	-	add[la].ob, add[la].qh	-
Immd/Part w/Acc (w/ or w/o Init)	-	-	-	-	add[la].ob, add[la].qh	-
Element/Element	-	-	-	-	-	-
Saturation Addition						
Part/Part	-	hadd,ss, hadd,us	hadd,ss, hadd,us	-	add.ob,add.qh	vaddsbs,vaddubs, vaddshs,vadduhs, vaddsws,vadduws,vaddfp
Scalar/Part	-	-	-	-	add.ob,add.qh	-
Immd/Part	-	-	-	-	add.ob,add.qh	-
Modular Add. High						
Part/Part	-	-	-	-	-	vaddcuw
Sat. RedAdd w/El.	-	-	-	-	-	vsumsws
Sat. Part. RedAdd w/Even	-	-	-	-	-	vsum2sws
Sat. Part. RedAdd w/Part	-	-	-	-	-	vsum4sbs, vsum4ubs, vsum4shs
Sat. RedAdd and Pack	-	-	-	-	-	-
Sat. RedAdd/Sub and Pack	-	-	-	-	-	-

Table C.2 cont'd.
SWAR Addition Operations

Operation Types	Sun VIS	Intel MMX	Intel SSE	Intel SSE2
Modular Addition				
Part/Part	fpadd16s,fpadd16, fpadd32s,fpadd32	paddb, paddw, paddd	addps	paddb, paddw, paddd, paddq,addpd
Immd/Part	-	-	-	-
Part/Part w/Acc (w/ or w/o Init)	-	-	-	-
Scalar/Part w/Acc (w/ or w/o Init)	-	-	-	-
Immd/Part w/Acc (w/ or w/o Init)	-	-	-	-
Element/Element	-	-	addss	addsd
Saturation Addition				
Part/Part	-	paddsb,paddusb, paddsw,paddusw	-	paddsb,paddusb, paddsw,paddusw
Scalar/Part	-	-	-	-
Immd/Part	-	-	-	-
Modular Add. High				
Part/Part	-	-	-	-
Sat. RedAdd w/El.	-	-	-	-
Sat. Part. RedAdd w/Even	-	-	-	-
Sat. Part. RedAdd w/Part	-	-	-	-
Sat. RedAdd and Pack	-	-	-	-
Sat. RedAdd/Sub and Pack	-	-	-	-

Table C.2 cont'd.
SWAR Addition Operations

Operation Types	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Modular Addition				
Part/Part	-	-	addps	-
Immd/Part	-	-	-	-
Part/Part w/Acc (w/ or w/o Init)	-	-	-	-
Scalar/Part w/Acc (w/ or w/o Init)	-	-	-	-
Immd/Part w/Acc (w/ or w/o Init)	-	-	-	-
Element/Element	-	-	addss	-
Saturation Addition				
Part/Part	pfadd	-	-	paddsiw
Scalar/Part	-	-	-	-
Immd/Part	-	-	-	-
Modular Add. High				
Part/Part	-	-	-	-
Sat. RedAdd w/El.	-	-	-	-
Sat. Part. RedAdd w/Even	-	-	-	-
Sat. Part. RedAdd w/Part	-	-	-	-
Sat. RedAdd and Pack	pfacc	-	-	-
Sat. RedAdd/Sub and Pack	-	pfpnacc	-	-

Table C.3
SWAR Subtraction Operations

Operation Types	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX	Motorola AltiVec
Modular Subtraction						
Part/Part	-	hsub	- hsub	sub.ps	-	vsububm, vsubuhm, vsubuwm
Part/Part w/Acc Diff (w/ or w/o Init)	-	-	-	-	sub.ob, sub.qh	-
Scalar/Part w/Acc Diff (w/ or w/o Init)	-	-	-	-	sub.ob, sub.qh	-
Immd/Part w/Acc Diff (w/ or w/o Init)	-	-	-	-	sub.ob, sub.qh	-
Element/Element	-	-	-	-	-	-
Saturation Subtraction						
Part/Part	-	hsub,ss, hsub,us	hsub,ss, hsub,us	-	sub.ob,sub.qh	vsubsbs,vsububs, vsubshs,vsubuhs vsubsws,vsubuws,vsubfp
Scalar/Part	-	-	-	-	sub.ob,sub.qh	-
Immd/Part	-	-	-	-	sub.ob,sub.qh	-
Subtraction High						
Part/Part	-	-	-	-	-	vsubcuw
Sat. RedSub and Pack	-	-	-	-	-	-
RedAdd of Abs. Diffs	perr	-	-	-	-	-
Sum Abs Diffs; Sat Acc.	-	-	-	-	-	-

Table C.3 cont'd.
SWAR Subtraction Operations

Operation Types	Sun VIS	Intel MMX	Intel SSE	Intel SSE2
Modular Subtraction				
Part/Part	fsub16s,fsub16, fsub32s,fsub32	psubb, psubw, psubd	subps	psubb, psubw, psubd, psubq,psubq,subpd
Part/Part w/Acc Diff (w/ or w/o Init)	-	-	-	-
Scalar/Part w/Acc Diff (w/ or w/o Init)	-	-	-	-
Immd/Part w/Acc Diff (w/ or w/o Init)	-	-	-	-
Element/Element	-	-	subss	subsd
Saturation Subtraction				
Part/Part	-	psubsb,psubusb psubsw,psubusw	-	psubsb,psubusb, psubsw,psubusw
Scalar/Part	-	-	-	-
Immd/Part	-	-	-	-
Subtraction High				
Part/Part	-	-	-	-
Sat. RedSub and Pack	-	-	-	-
RedAdd of Abs. Diffs	pdist	-	psadbw	psadbw
Sum Abs Diffs; Sat Acc.	-	-	-	-

Table C.3 cont'd.
SWAR Subtraction Operations

Operation Types	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Modular Subtraction				
Part/Part	-	-	subps	-
Part/Part w/Acc Diff (w/ or w/o Init)	-	-	-	-
Scalar/Part w/Acc Diff (w/ or w/o Init)	-	-	-	-
Immd/Part w/Acc Diff (w/ or w/o Init)	-	-	-	-
Element/Element	-	-	subss	-
Saturation Subtraction				
Part/Part	pfsub(r)	-	-	psubsiw
Scalar/Part	-	-	-	-
Immd/Part	-	-	-	-
Subtraction High				
Part/Part	-	-	-	-
Sat. RedSub and Pack	-	pfnacc	-	-
RedAdd of Abs. Diffs	-	psadbw	-	-
Sum Abs Diffs; Sat Acc.	-	-	-	pdistib

Table C.4
Maximum and Minimum Operations

Operation Types	DEC MVI	HP MAX	SGI MIPS-V	SGI MDMX	Motorola AltiVec	Sun VIS
Maximum						
Part/Part	maxsb8,maxub8, maxsw4,maxuw4	-	-	max.ob, max.qh	vmaxsb,vmaxub, vmaxsh,vmaxuh, vmaxsw,vmaxuw,vmaxfp	-
Scalar/Part	-	-	-	max.ob, max.qh	-	-
Immd/Part	maxsb8,maxub8, maxsw4,maxuw4	-	-	max.ob, max.qh	-	-
Element/Element	-	-	-	-	-	-
Minimum						
Part/Part	minsb8,minub8, minsw4,minuw4	-	-	min.ob, min.qh	vminsb,vminub, vminsh,vminuh, vminsw,vminuw,vminfp	-
Scalar/Part	-	-	-	min.ob, min.qh	-	-
Immd/Part	minsb8,minub8, minsw4,minuw4	-	-	min.ob, min.qh	-	-
Element/Element	-	-	-	-	-	-
Magnitude Part/Part	-	-	-	-	-	-
Abs. Value Part/Part	-	-	abs.ps	-	-	-
Negate Part/Part	-	-	neg.ps	-	-	-
Generate Sign Mask	-	-	-	-	-	-

Operation Types	Intel MMX	Intel SSE	Intel SSE2	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Maximum							
Part/Part	-	pmaxub, pmaxsw, maxps	pmaxub, pmaxsw, maxpd	pfmax	pmaxub, pmaxsw	maxps	-
Scalar/Part	-	-	-	-	-	-	-
Immd/Part	-	-	-	-	-	-	-
Element/Element	-	maxss	maxsd	-	-	maxss	-
Minimum							
Part/Part	-	pminub, pminsw, minps	pminub, pminsw, minpd	pfmin	pminub, pminsw	minps	-
Scalar/Part	-	-	-	-	-	-	-
Immd/Part	-	-	-	-	-	-	-
Element/Element	-	minss	minsd	-	-	minss	-
Magnitude Part/Part	-	-	-	-	-	-	pmagw
Abs. Value Part/Part	-	-	-	-	-	-	-
Negate Part/Part	-	-	-	-	-	-	-
Generate Sign Mask	-	pmovmskb, movmskps	pmovmskb, movmskpd	-	pmovmskb	movmskps	-

Table C.5
Multiplication Operations

Operation Types	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX	Motorola AltiVec
Modular Multiplication						
Part/Part	-	-	-	mul.ps	-	vmulesb, vmuleub, vmulosb, vmuloub, vmulesh, vmuleuh, vmulosh, vmulouh
Immd/Part	-	-	-	-	-	-
Part/Part w/Acc (w/ or w/o Init)	-	-	-	-	mul[la].ob, mul[la].qh	-
Scalar/Part w/Acc (w/ or w/o Init)	-	-	-	-	mul[la].ob, mul[la].qh	-
Immd/Part w/Acc (w/ or w/o Init)	-	-	-	-	mul[la].ob, mul[la].qh	-
Part/Part w/Acc Subt (w/ or w/o Init)	-	-	-	-	mul[la].ob, mul[la].qh	-
Scalar/Part w/Acc Subt (w/ or w/o Init)	-	-	-	-	mul[la].ob, mul[la].qh	-
Immd/Part w/Acc Subt (w/ or w/o Init)	-	-	-	-	mul[la].ob, mul[la].qh	-
Part/Element	-	-	-	-	-	-
Element/Element	-	-	-	-	-	-
Modular Mul. High						
Pt/Pt Store in Enh.	-	-	-	-	-	-
Pt/Pt Store in Implied	-	-	-	-	-	-
Pt/Pt Acc. w/Implied	-	-	-	-	-	-
Sat. Multiplication						
Part/Part	-	-	-	-	mul.ob,mul.qh	-
Scalar/Part	-	-	-	-	mul.ob,mul.qh	-
Immd/Part	-	-	-	-	mul.ob,mul.qh	-
Mult. by Sign (-,0,+)						
Part/Part	-	-	-	-	msgn.qh	-
Scalar/Part	-	-	-	-	msgn.qh	-
Immd/Part	-	-	-	-	msgn.qh	-
Average	-	have	havg	-	-	vavgsb,vavgub, vavgsh,vavguh, vavgsw,vavguw

Table C.5 cont'd.
Multiplication Operations

Operation Types	Sun VIS	Intel MMX	Intel SSE	Intel SSE2
Modular Multiplication				
Part/Part	fmul8x16, fmul8sux16, fmul8ulx16, fmuld8sux16, fmuld8ulx16	pmullw	mulps	pmullw, pmuludq, pmuludq, mulpd
Immd/Part	-	-	-	-
Part/Part w/Acc (w/ or w/o Init)	-	-	-	-
Scalar/Part w/Acc (w/ or w/o Init)	-	-	-	-
Immd/Part w/Acc (w/ or w/o Init)	-	-	-	-
Part/Part w/Acc Subt (w/ or w/o Init)	-	-	-	-
Scalar/Part w/Acc Subt (w/ or w/o Init)	-	-	-	-
Immd/Part w/Acc Subt (w/ or w/o Init)	-	-	-	-
Part/Element	fmul8x16au, fmul8x16al	-	-	-
Element/Element	-	-	mulss	mulsd
Modular Mul. High				
Pt/Pt Store in Enh.	-	pmulhw	pmulhuw	pmulhuw,pmulhw
Pt/Pt Store in Implied	-	-	-	-
Pt/Pt Acc. w/Implied	-	-	-	-
Sat. Multiplication				
Part/Part	-	-	-	-
Scalar/Part	-	-	-	-
Immd/Part	-	-	-	-
Mult. by Sign (-,0,+)				
Part/Part	-	-	-	-
Scalar/Part	-	-	-	-
Immd/Part	-	-	-	-
Average	-	-	pavgb, pavgw	pavgb, pavgw

Table C.5 cont'd.
Multiplication Operations

Operation Types	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Modular Multiplication				
Part/Part	-	-	mulps	-
Immd/Part	-	-	-	-
Part/Part w/Acc (w/ or w/o Init)	-	-	-	-
Scalar/Part w/Acc (w/ or w/o Init)	-	-	-	-
Immd/Part w/Acc (w/ or w/o Init)	-	-	-	-
Part/Part w/Acc Subt (w/ or w/o Init)	-	-	-	-
Scalar/Part w/Acc Subt (w/ or w/o Init)	-	-	-	-
Immd/Part w/Acc Subt (w/ or w/o Init)	-	-	-	-
Part/Element	-	-	-	-
Element/Element	-	-	mulss	-
Modular Mul. High				
Pt/Pt Store in Enh.	pmulhrw	pmulhuw	-	pmulhrw
Pt/Pt Store in Implied	-	-	-	pmulhriw
Pt/Pt Acc. w/Implied	-	-	-	pmachriw
Sat. Multiplication				
Part/Part	pfmul	-	-	-
Scalar/Part	-	-	-	-
Immd/Part	-	-	-	-
Mult. by Sign (-,0,+)				
Part/Part	-	-	-	-
Scalar/Part	-	-	-	-
Immd/Part	-	-	-	-
Average	pavgusb	pavgb, pavgw	-	paveb

Table C.6
Combined Arithmetic Operations

Operation Types	DEC MVI	HP MAX	SGI MIPS-V	SGI MDMX	Motorola AltiVec
Multiply, then Add Neighboring Fields	-	-	-	-	-
Multiply/Mod. Add	-	-	madd.ps	-	vmaddfp, vmladduhm
Negated Multiply/Mod. Add	-	-	nmadd.ps	-	-
Multiply/Sat. Add	-	-	-	-	vmhaddshs
Multiply(w/Rnd)/Sat. Add	-	-	-	-	vmhraddshs
Multiply/Mod. Subtract	-	-	msub.ps	-	-
Negated Multiply/Mod. Subtract	-	-	nmsub.ps	-	vnmsubfp
Multiply, then Modular Add Neighbor w/Part	-	-	-	-	vmsumubm, vmsumshm, vmsumuhm, vmsummbm
Multiply, then Saturate Add Neighbor w/Part	-	-	-	-	vmsumshs, vmsumuhs

Table C.6 cont'd.
Combined Arithmetic Operations

Operation Types	Sun VIS	Intel MMX	Intel SSE	Intel SSE2	AMD 3DNow! (All families)	Cyrix EMMX
Multiply, then Add Neighboring Fields	-	pmaddwd	-	pmaddwd	-	-
Multiply/Mod. Add	-	-	-	-	-	-
Negated Multiply/Mod. Add	-	-	-	-	-	-
Multiply/Sat. Add	-	-	-	-	-	-
Multiply(w/Rnd)/Sat. Add	-	-	-	-	-	-
Multiply/Mod. Subtract	-	-	-	-	-	-
Negated Multiply/Mod. Subtract	-	-	-	-	-	-
Multiply, then Modular Add Neighbor w/Part	-	-	-	-	-	-
Multiply, then Saturate Add Neighbor w/Part	-	-	-	-	-	-

Table C.7
Division and Advanced Arithmetic Operations

Operation Types	DEC MVI	HP MAX	SGI MIPS-V	SGI MDMX	Motorola AltiVec	Sun VIS	Intel MMX
Divide							
Part/Part	-	-	-	-	-	-	-
Element/Element	-	-	-	-	-	-	-
Square Root							
Part/Part	-	-	-	-	-	-	-
Element/Element	-	-	-	-	-	-	-
Reciprocal Approx.							
Part	-	-	-	-	vrefp	-	-
Element	-	-	-	-	-	-	-
Recip. Sq. Rt. Approx.							
Part	-	-	-	-	vrsqrtefp	-	-
Element	-	-	-	-	-	-	-
$Log_2(x)$ Approx.							
Part	-	-	-	-	vlogefp	-	-
2^x Approx.							
Part	-	-	-	-	vexptefp	-	-

Operation Types	Intel SSE	Intel SSE2	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Divide						
Part/Part	divps	divpd	-	-	divps	-
Element/Element	divss	divsd	-	-	divss	-
Square Root						
Part/Part	sqrtps	sqrtpd	-	-	sqrtps	-
Element/Element	sqrtss	sqrtsd	-	-	sqrtss	-
Reciprocal Approx.						
Part	rcpps	-	-	-	rcpps	-
Element	rcpss	-	pfrcp/pfrcpit1/pfrcpit2	-	rcpss	-
Recip. Sq. Rt. Approx.						
Part	rsqrtps	-	-	-	rsqrtps	-
Element	rsqrtss	-	pfrsqrt/pfrsqit1/fprcpit2	-	rsqrtss	-
$Log_2(x)$ Approx.						
Part	-	-	-	-	-	-
2^x Approx.						
Part	-	-	-	-	-	-

Table C.8
Shift and Rotate Operations

Operation Types	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX	Motorola AltiVec	Sun VIS
Shift Left Logical							
Part by Part	-	-	-	-	sll.ob, sll.qh	vslb, vslh, vslw	-
Part by Scalar	-	-	-	-	sll.ob, sll.qh		-
						vsl	
Part by Single	sll	-	-	-	-		-
						vslo	
Part by Immd	sll	-	hshl	-	sll.ob, sll.qh	-	-
Shift Right Logical							
Part by Part	-	-	-	-	srl.ob, srl.qh	vsrb, vsrh, vsrw	-
Part by Scalar	-	-	-	-	srl.ob, srl.qh	-	-
Part by Single	srl	-	-	-	-	vsro	-
Part by Immd	srl	-	hshr,u	-	srl.ob, srl.qh	-	-
Shift Right Arithmetic							
Part by Part	-	-	-	-	sra.qh	vsrab, vsrah, vsraw	-
Part by Scalar	-	-	-	-	sra.qh	-	-
Part by Single	sra	-	-	-	-	-	-
Part by Immd	sra	-	hshr or hshr,s	-	sra.qh	-	-
Shift Left and Add							
by 1 bit	-	-	-	-	-	-	-
by 2 bits	s4addq	-	-	-	-	-	-
by 3 bits	s8addq	-	-	-	-	-	-
Shift Left and Sat. Add							
by 1,2, or 3 bits	-	hshladd	-	-	-	-	-
Shift Left and Subtract							
by 2 bits	s4subq	-	-	-	-	-	-
by 3 bits	s8subq	-	-	-	-	-	-
Shift Right and Sat. Add							
by 1,2, or 3 bits	-	hshradd	-	-	-	-	-
Rotate							
Part by Part	-	-	-	-	-	vrlb, vrlh, vrlw	-

Table C.8 cont'd.
Shift and Rotate Operations

Operation Types	Intel MMX	Intel SSE	Intel SSE2	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Shift Left Logical							
Part by Part	-	-	-	-	-	-	-
Part by Scalar	-	-	-	-	-	-	-
Part by Single	psllw, pslld, psllq	-	psllw, pslld, psllq	-	-	-	-
Part by Immd	psllw, pslld, psllq	-	psllw, pslld, psllq, pslldq	-	-	-	-
Shift Right Logical							
Part by Part	-	-	-	-	-	-	-
Part by Scalar	-	-	-	-	-	-	-
Part by Single	psrlw, psrld, psrlq	-	psrlw, psrld, psrlq	-	-	-	-
Part by Immd	psrlw, psrld, psrlq	-	psrlw, psrld, psrlq, psrldq	-	-	-	-
Shift Right Arithmetic							
Part by Part	-	-	-	-	-	-	-
Part by Scalar	-	-	-	-	-	-	-
Part by Single	psraw, psrad	-	psraw, psrad	-	-	-	-
Part by Immd	psraw, psrad	-	psraw, psrad	-	-	-	-
Shift Left and Add							
by 1 bit	-	-	-	-	-	-	-
by 2 bits	-	-	-	-	-	-	-
by 3 bits	-	-	-	-	-	-	-
Shift Left and Sat. Add							
by 1,2, or 3 bits	-	-	-	-	-	-	-
Shift Left and Subtract							
by 2 bits	-	-	-	-	-	-	-
by 3 bits	-	-	-	-	-	-	-
Shift Right and Sat. Add							
by 1,2, or 3 bits	-	-	-	-	-	-	-
Rotate							
Part by Part	-	-	-	-	-	-	-

Table C.9
Polymorphic Operations

Operation Types	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX	Motorola AltiVec	Sun VIS
AND							
Part/Part	and	and	and	-	and.ob,and.qh	vand	fands,fand
Part/Imm	and	-	-	-	and.ob,and.qh	-	-
Part/Scalar	-	-	-	-	and.ob,and.qh	-	-
ANDN							
Part/Part	bic	andcm	andcm	-	-	vandc	fandnot[12]s[1],fandnot[12]
Part/Imm	bic	-	-	-	-	-	-
NAND							
Part/Part	-	-	-	-	-	-	fnands,fnand
Part/Imm	-	-	-	-	-	-	-
OR							
Part/Part	bis	or	or	-	or.ob,or.qh	vor	fors,for
Part/Imm	bis	-	-	-	or.ob,or.qh	-	-
Part/Scalar	-	-	-	-	or.ob,or.qh	-	-
ORN							
Part/Part	ornot	-	-	-	-	-	fornot[12]s,fornot[12]
Part/Imm	ornot	-	-	-	-	-	-
NOR							
Part/Part	-	-	-	-	nor.ob,nor.qh	vnor	fnors,fnor
Part/Imm	-	-	-	-	nor.ob,nor.qh	-	-
Part/Scalar	-	-	-	-	nor.ob,nor.qh	-	-
XOR							
Part/Part	xor	xor	xor	-	xor.ob,xor.qh	vxor	fxors,fxor
Part/Imm	xor	-	-	-	xor.ob,xor.qh	-	-
Part/Scalar	-	-	-	-	xor.ob,xor.qh	-	-
XORN							
Part/Part	eqv	-	-	-	-	-	-
Part/Imm	eqv	-	-	-	-	-	-
NXOR							
Part/Part	-	-	-	-	-	-	fxnors,fxnor
Part/Imm	-	-	-	-	-	-	-
Population	ctpop	-	-	-	-	-	-
Leading 0 bits	ctlz	-	-	-	-	-	-
Trailing 0 bits	cttz	-	-	-	-	-	-

[1] “[12]” means “1” or “2”.

Table C.9 cont'd.
Polymorphic Operations

Operation Types	Intel MMX	Intel SSE	Intel SSE2	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
AND							
Part/Part	pand	andps	pand,andpd	-	-	andps	-
Part/Imm	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-
ANDN							
Part/Part	pandn	andnps	pandn,andnpd	-	-	andnps	-
Part/Imm	-	-	-	-	-	-	-
NAND							
Part/Part	-	-	-	-	-	-	-
Part/Imm	-	-	-	-	-	-	-
OR							
Part/Part	por	orps	por,orpd	-	-	orps	-
Part/Imm	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-
ORN							
Part/Part	-	-	-	-	-	-	-
Part/Imm	-	-	-	-	-	-	-
NOR							
Part/Part	-	-	-	-	-	-	-
Part/Imm	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-
XOR							
Part/Part	pxor	xorps	pxor,xorpd	-	-	xorps	-
Part/Imm	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-
XORN							
Part/Part	-	-	-	-	-	-	-
Part/Imm	-	-	-	-	-	-	-
NXOR							
Part/Part	-	-	-	-	-	-	-
Part/Imm	-	-	-	-	-	-	-
Population	-	-	-	-	-	-	-
Leading 0 bits	-	-	-	-	-	-	-
Trailing 0 bits	-	-	-	-	-	-	-

Table C.10
Condition Testing Operations

Operation Types	DEC MVI	HP MAX	SGI MIPS-V	SGI MDMX	Motorola AltiVec
Forms of Result	Bitmask	-	FP CC Bits	FP CC Bits	Field Mask All/None Bits
Equality					
Part/Part	cmpeq	-	c.eq.ps	c.eq.ob, c.eq.qh	vcmpequb, vcmpequh, vcmpequw,vcmpeqfp,
Part/Imm	cmpeq	-	-	c.eq.ob,c.eq.qh	-
Part/Scalar	-	-	-	c.eq.ob,c.eq.qh	-
El/El	-	-	-	-	-
Inequality					
Part/Part	-	-	c.neq.ps	-	-
Part/Imm	-	-	-	-	-
Part/Scalar	-	-	-	-	-
El/El	-	-	-	-	-
Greater Than					
Part/Part	-	-	c.gt.ps	-	vcmpgtsb,vcmpgtub, vcmpgtsh,vcmpgtuh, vcmpgtsw,vcmpgtuw,vcmpgtfp
El/El	-	-	-	-	-
Less Than					
Part/Part	-	-	c.lt.ps	c.lt.ob, c.lt.qh	-
Part/Imm	-	-	-	c.lt.ob,c.lt.qh	-
Part/Scalar	-	-	-	c.lt.ob,c.lt.qh	-
El/El	-	-	-	-	-
Greater or Equal					
Part/Part	cmpbge	-	c.ge.ps	-	vcmpgefp
Part/Imm	cmpbge	-	-	-	-
Part/Scalar	-	-	-	-	-
Less or Equal					
Part/Part	-	-	c.le.ps	c.le.ob, c.le.qh	-
Part/Imm	-	-	-	c.le.ob,c.le.qh	-
Part/Scalar	-	-	-	c.le.ob,c.le.qh	-
El/El	-	-	-	-	-
Not Less nor Equal					
Part/Part	-	-	c.nle.ps	-	-
Element/Element	-	-	-	-	-
Not Less Than					
Part/Part	-	-	c.nlt.ps	-	-
Element/Element	-	-	-	-	-

Table C.10 cont'd.
Condition Testing Operations

Operation Types	Sun VIS	Intel MMX	Intel SSE	Intel SSE2
Forms of Result	Bitmask	Field Mask	Field Mask	Field Mask
Equality				
Part/Part	fcmpeq16, fcmpeq32	pcmpeqb, pcmpeqw, pcmpeqd	cmpps/0	pcmpeqb, pcmpeqw, pcmpeqd, cmppd/0
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
El/El	-	-	cmpss/0	cmpsd/0
Inequality				
Part/Part	fcmpne16, fcmpne32	-	cmpps/4	cmppd/4
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
El/El	-	-	cmpss/4	cmpsd/4
Greater Than				
Part/Part	fcmpgt16, fcmpgt32	pcmpgtb, pcmpgtw, pcmpgtd	-	pcmpgtb, pcmpgtw, pcmpgtd,
El/El	-	-	-	-
Less Than				
Part/Part	-	-	cmpps/1	cmppd/1
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
El/El	-	-	cmpss/1	cmpsd/1
Greater or Equal				
Part/Part	-	-	-	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
Less or Equal				
Part/Part	fcmple16, fcmple32	-	cmpps/2	cmppd/2
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
El/El	-	-	cmpss/2	cmpsd/2
Not Less nor Equal				
Part/Part	-	-	cmpps/6	cmppd/6
Element/Element	-	-	cmpss/6	cmpsd/6
Not Less Than				
Part/Part	-	-	cmpps/5	cmppd/5
Element/Element	-	-	cmpss/5	cmpsd/5

Table C.10 cont'd.
Condition Testing Operations

Operation Types	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Forms of Result	Field Mask	-	Field Mask	-
Equality				
Part/Part	pfcmpeq	-	cmpps/0	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
El/El	-	-	cmpss/0	-
Inequality				
Part/Part	-	-	cmpps/4	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
El/El	-	-	cmpss/4	-
Greater Than				
Part/Part	pfcmpgt	-	-	-
El/El	-	-	-	-
Less Than				
Part/Part	-	-	cmpps/1	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
El/El	-	-	cmpss/1	-
Greater or Equal				
Part/Part	pfcmpge	-	-	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
Less or Equal				
Part/Part	-	-	cmpps/2	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
El/El	-	-	cmpss/2	-
Not Less nor Equal				
Part/Part	-	-	cmpps/6	-
Element/Element	-	-	cmpss/6	-
Not Less Than				
Part/Part	-	-	cmpps/5	-
Element/Element	-	-	cmpss/5	-

Table C.10 cont'd.
Condition Testing Operations

Operation Types	DEC MVI	HP MAX	SGI MIPS-V	SGI MDMX	Motorola AltiVec	Sun VIS
Not (Greater or Equal) Pt/Pt	-	-	c.nge.ps	-	-	-
Greater or Less Than Pt/Pt	-	-	c.gl.ps	-	-	-
Not (Greater or Less) Pt/Pt	-	-	c.ngl.ps	-	-	-
Not Greater Than Pt/Pt	-	-	c.ngt.ps	-	-	-
Greater, Less, or Equal Pt/Pt	-	-	c.gle.ps	-	-	-
Not (Gr., Less, or Eq.) Pt/Pt	-	-	c.ngle.ps	-	-	-
Ordered						
Part/Part	-	-	c.or.ps	-	-	-
Element/Element	-	-	-	-	-	-
Unordered						
Part/Part	-	-	c.un.ps	-	-	-
Element/Element	-	-	-	-	-	-
Unordered or Equal Pt/Pt	-	-	c.ueq.ps	-	-	-
Signaling Equal Pt/Pt	-	-	c.seq.ps	-	-	-
Signaling Not Equal Pt/Pt	-	-	c.sne.ps	-	-	-
Ordered or Greater Than Pt/Pt	-	-	c.ogt.ps	-	-	-
Unordered or Greater Pt/Pt	-	-	c.ugt.ps	-	-	-
Ord. or Greater or Eq. Pt/Pt	-	-	c.oge.ps	-	-	-
Unord. or Grtr. or Eq. Pt/Pt	-	-	c.uge.ps	-	-	-
Ordered or Less Than Pt/Pt	-	-	c.olt.ps	-	-	-
Unordered or Less Than Pt/Pt	-	-	c.ult.ps	-	-	-
Ordered or Less or Eq. Pt/Pt	-	-	c.ole.ps	-	-	-
Unord. or Less or Eq. Pt/Pt	-	-	c.ule.ps	-	-	-
Ord. or Greater or Less Pt/Pt	-	-	c.ogl.ps	-	-	-
Compare Bounds Pt/Pt	-	-	-	-	vcmpbfp	-
Set Cond. Codes						
Ordered El/El	-	-	-	-	-	-
Unord. El/El	-	-	-	-	-	-

Table C.10 cont'd.
Condition Testing Operations

Operation Types	Intel MMX	Intel SSE	Intel SSE2	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Not (Greater or Equal) Pt/Pt	-	-	-	-	-	-	-
Greater or Less Than Pt/Pt	-	-	-	-	-	-	-
Not (Greater or Less) Pt/Pt	-	-	-	-	-	-	-
Not Greater Than Pt/Pt	-	-	-	-	-	-	-
Greater, Less, or Equal Pt/Pt	-	-	-	-	-	-	-
Not (Gr., Less, or Eq.) Pt/Pt	-	-	-	-	-	-	-
Ordered							
Part/Part	-	cmpps/7	cmppd/7	-	-	cmpps/7	-
Element/Element	-	cmpss/7	cmpsd/7	-	-	cmpss/7	-
Unordered							
Part/Part	-	cmpps/3	cmppd/3	-	-	cmpps/3	-
Element/Element	-	cmpss/3	cmpsd/3	-	-	cmpss/3	-
Unordered or Equal Pt/Pt	-	-	-	-	-	-	-
Signaling Equal Pt/Pt	-	-	-	-	-	-	-
Signaling Not Equal Pt/Pt	-	-	-	-	-	-	-
Ordered or Greater Than Pt/Pt	-	-	-	-	-	-	-
Unordered or Greater Pt/Pt	-	-	-	-	-	-	-
Ord. or Greater or Eq. Pt/Pt	-	-	-	-	-	-	-
Unord. or Grtr. or Eq. Pt/Pt	-	-	-	-	-	-	-
Ordered or Less Than Pt/Pt	-	-	-	-	-	-	-
Unordered or Less Than Pt/Pt	-	-	-	-	-	-	-
Ordered or Less or Eq. Pt/Pt	-	-	-	-	-	-	-
Unord. or Less or Eq. Pt/Pt	-	-	-	-	-	-	-
Ord. or Greater or Less Pt/Pt	-	-	-	-	-	-	-
Compare Bounds Pt/Pt	-	-	-	-	-	-	-
Set Cond. Codes							
Ordered El/El	-	comiss	comisd	-	-	comiss	-
Unord. El/El	-	ucomiss	ucomisd	-	-	ucomiss	-

Table C.11
Conditional Flow Control Operations

Operation Types	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX	Motorola AltiVec	Sun VIS
Branch On...			[1]				
None True	beq	-	-	-	-	-	-
Any True	bne	-	-	-	-	-	-
All Equal (Part/Part)	-	combt,=	cmpb,*=	-	-	-	-
All Equal (Part/Immed)	-	comibt,=	cmpib,*=	-	-	-	-
All Inequal (Part/Part)	-	combt,<>	cmpb,*<>	-	-	-	-
All Inequal (Part/Immed)	-	comibt,<>	cmpib,*<>	-	-	-	-
Operate and Null Next On...							
AND/Any True?	-	and,<>	and,*<>	-	-	-	-
AND/None True?	-	and,=	and,*=	-	-	-	-
ANDN/Any True?	-	andcm,<>	andcm,*<>	-	-	-	-
ANDN/None True?	-	andcm,=	andcm,*=	-	-	-	-
OR/Any True?	-	or,<>	or,*<>	-	-	-	-
OR/None True?	-	or,=	or,*=	-	-	-	-
XOR/Any True?	-	xor,<>	xor,*<>	-	-	-	-
XOR/None True?	-	xor,=	xor,*=	-	-	-	-
XOR/Any False?	-	uxor,shz uxor,sbz	uxor,*swz uxor,*shz uxor,*sbz	-	-	-	-
XOR/None False?	-	uxor,nhz uxor,nbz	uxor,*nwz uxor,*nhz uxor,*nbz	-	-	-	-
Add Complement/Any False? $(A + \overline{B})$	-	uaddcm,shz uaddcm,sbz	uaddcm,*swz uaddcm,*shz uaddcm,*sbz	-	-	-	-
Add Complement/None False? $(A + \overline{B})$	-	uaddcm,nhz uaddcm,nbz	uaddcm,*nwz uaddcm,*nhz uaddcm,*nbz	-	-	-	-

[1] 1x32 versions of these tests are also available. For example, “cmpb,<>”.

Table C.11 cont'd.
Conditional Flow Control Operations

Operation Types	Intel MMX	Intel SSE	Intel SSE2	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Branch On...							
None True	-	-	-	-	-	-	-
Any True	-	-	-	-	-	-	-
All Equal (Part/Part)	-	-	-	-	-	-	-
All Equal (Part/Immed)	-	-	-	-	-	-	-
Any Inequal (Part/Part)	-	-	-	-	-	-	-
Any Inequal (Part/Immed)	-	-	-	-	-	-	-
Operate and Null Next On...							
AND/Any True?	-	-	-	-	-	-	-
AND/None True?	-	-	-	-	-	-	-
ANDN/Any True?	-	-	-	-	-	-	-
ANDN/None True?	-	-	-	-	-	-	-
OR/Any True?	-	-	-	-	-	-	-
OR/None True?	-	-	-	-	-	-	-
XOR/Any True?	-	-	-	-	-	-	-
XOR/None True?	-	-	-	-	-	-	-
XOR/Any False?	-	-	-	-	-	-	-
XOR/None False?	-	-	-	-	-	-	-
Add Complement/Any False? $(A + \overline{B})$	-	-	-	-	-	-	-
Add Complement/None False? $(A + \overline{B})$	-	-	-	-	-	-	-

Table C.12
Conditional Data Manipulation Operations

Operation Types	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX	Motorola AltiVec	Sun VIS
Move Reg/Imm On...							
None True	cmoveq	-	-	-	-	-	-
Any True	cmovne	-	-	-	-	-	-
Zero Masked Bytes	zap	-	-	-	-	-	-
Zero UnMasked Bytes	zapnot	-	-	-	-	-	-
Clear Reg & Null Next/All							
Part/Part	-	comclr,=	cmpclr,*=	-	-	-	-
Part/Imm	-	comiclr,=	cmpiclr,*=	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-
Clear Reg & Null Next/Not All							
Part/Part	-	comclr,<>	cmpclr,*<>	-	-	-	-
Part/Imm	-	comiclr,<>	cmpiclr,*<>	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-
Load Reg. On...							
Zero	-	-	-	-	-	-	-
Non-Zero	-	-	-	-	-	-	-
Negative	-	-	-	-	-	-	-
Non-Negative	-	-	-	-	-	-	-
Move Reg. On...							
CC bit TRUE	-	-	-	movt.ps	-	-	-
CC bit FALSE	-	-	-	movf.ps	-	-	-
Pick True							
Part/Part	-	-	-	-	pickt.ob,pickt.qh	vsel	-
Part/Imm	-	-	-	-	pickt.ob,pickt.qh	-	-
Part/Scalar	-	-	-	-	pickt.ob,pickt.qh	-	-
Pick False							
Part/Part	-	-	-	-	pickf.ob,pickf.qh	vsel	-
Part/Imm	-	-	-	-	pickf.ob,pickf.qh	-	-
Part/Scalar	-	-	-	-	pickf.ob,pickf.qh	-	-

Table C.12 cont'd.
Conditional Data Manipulation Operations

Operation Types	Intel MMX	Intel SSE	Intel SSE2	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Move Reg/Imm On...							
None True	-	-	-	-	-	-	-
Any True	-	-	-	-	-	-	-
Zero Masked Bytes	-	-	-	-	-	-	-
Zero UnMasked Bytes	-	-	-	-	-	-	-
Clear Reg & Null Next/All							
Part/Part	-	-	-	-	-	-	-
Part/Imm	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-
Clear Reg & Null Next/Any							
Part/Part	-	-	-	-	-	-	-
Part/Imm	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-
Load Reg. On...							
Zero	-	-	-	-	-	-	pmvzb
Non-Zero	-	-	-	-	-	-	pmvnzb
Negative	-	-	-	-	-	-	pmvlzb
Non-Negative	-	-	-	-	-	-	pmvgezb
Move Reg. On...							
CC bit TRUE	-	-	-	-	-	-	-
CC bit FALSE	-	-	-	-	-	-	-
Pick True							
Part/Part	-	-	-	-	-	-	-
Part/Imm	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-
Pick False							
Part/Part	-	-	-	-	-	-	-
Part/Imm	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-

Table C.13
Data Movement, Replication, and Type Conversion Operations

Operation Types	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX	Motorola AltitVec	Sun VIS
Move Reg.→Enh. Reg.	-	-	-	-	-	-	-
Move Enh. Reg.→Reg.	-	-	-	-	-	-	-
Move Enh. Reg. →Enh. Reg.	-	movb	movb	mov.ps	-	-	fsrc[12]s[1], fsrc[12]
Move Comp. Enh. Reg. →Enh. Reg.	-	-	-	-	-	-	fnot[12]s, fnot[12]
Pack Singles to Part	-	-	-	cvt.ps.s	-	-	-
Modular Move Acc→Reg							
Low Third of Acc.	-	-	-	-	racl.ob, racl.qh	-	-
Middle Third of Acc.	-	-	-	-	racm.ob, racm.qh	-	-
High Third of Acc.	-	-	-	-	rach.ob, rach.qh	-	-
Move Regs. to Low Acc.	-	-	-	-	wacl.ob, wacl.qh	-	-
Move Reg. to High Acc.	-	-	-	-	wach.ob, wach.qh	-	-
Replicate Field (Element/Part)	-	-	-	-	-	vspltb, vsplth, vspltw	-
Replicate Sign-Extended Immediate to Part	-	-	-	-	-	vspltisb, vspltish, vspltisw	-
Shift Rt, Rnd, & Sat Acc							
toward 0	-	-	-	-	rzu.ob, rzs.qh, rzu.qh	-	-
to nearest away from 0	-	-	-	-	rnau.ob, rnas.qh, rnau.qh	-	-
to nearest toward even	-	-	-	-	rne.ob, rnes.qh, rneu.qh	-	-
Convert int. to flt.	-	-	-	-	-	vcfux, vcfsx	-
Convert flt. to int.	-	-	-	-	-	vctuxs, vctsxs	-
Convert flt. to flt.	-	-	-	-	-	-	-
Round flt. value to int.							
to nearest	-	-	-	-	-	vrfin	-
toward zero	-	-	-	-	-	vrfiz	-
toward +infinity	-	-	-	-	-	vrfip	-
toward -infinity	-	-	-	-	-	vrfim	-

[1] "[12]" means "1" or "2".

Table C.13 cont'd.
Data Movement, Replication, and Type Conversion Operations

Operation Types	Intel MMX	Intel SSE	Intel SSE2
Move Reg→Enh. Reg.	movd	-	movd
Move Enh. Reg→Reg.	movd	-	movd
Move Enh. Reg→Enh. Reg.	movq	(movups)movaps	movq, movdq2q,movq2dq, (movdqu)movdqa, (movupd)movapd
Move Comp. Enh. Reg. →Enh. Reg.	-	-	-
Pack Singles to Part	-	-	-
Modular Move Acc→Reg			
Low Third of Acc.	-	-	-
Middle Third of Acc.	-	-	-
High Third of Acc.	-	-	-
Move Regs. to Low. Acc.	-	-	-
Move Reg. to High Acc.	-	-	-
Replicate Field	-	-	-
Replicate Sign-Extended Immediate to Part	-	-	-
Shift Rt, Rnd, & Sat Acc			
toward 0	-	-	-
to nearest away from 0	-	-	-
to nearest toward even	-	-	-
Convert int. to flt.	-	cvtpi2ps, cvtsi2ss	cvtpi2pd, cvtsi2sd, cvtdq2ps, cvtdq2pd
Convert flt. to int.	-	cvt(t)ps2pi[1], cvt(t)ss2si[1]	cvt(t)pd2pi[1], cvt(t)pd2dq[1], cvt(t)sd2si[1], cvt(t)ps2dq[1]
Convert flt. to flt.	-	-	cvtpd2ps, cvtps2pd, cvtsd2ss, cvtss2sd
Round flt. value to int.			
to nearest	-	-	-
toward zero	-	-	-
toward +infinity	-	-	-
toward -infinity	-	-	-

[1] Cvt* uses rounding mode specified in MXCSR. Cvtt* truncates the fractional part.

Table C.13 cont'd.
Data Movement, Replication, and Type Conversion Operations

Operation Types	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Move Reg→Enh. Reg.	-	-	-	-
Move Enh. Reg→Reg.	-	-	-	-
Move Enh. Reg→Enh. Reg.	-	-	(movups)movaps	-
Move Comp. Enh. Reg. →Enh. Reg.	-	-	- -	-
Pack Singles to Part	-	-	-	-
Modular Move Acc→Reg				
Low Third of Acc.	-	-	-	-
Middle Third of Acc.	-	-	-	-
High Third of Acc.	-	-	-	-
Move Regs. to Low. Acc.	-	-	-	-
Move Reg. to High Acc.	-	-	-	-
Replicate Field	-	-	-	-
Replicate Sign-Extended Immediate to Part	-	-	-	-
Shift Rt, Rnd, & Sat Acc				
toward 0	-	-	-	-
to nearest away from 0	-	-	-	-
to nearest toward even	-	-	-	-
Convert int. to flt.	pi2fd	pi2fw	cvtpi2ps, cvtsi2ss	-
Convert flt. to int.	pf2id	pf2iw	cvt(t)ps2pi[1], cvt(t)ss2si[1]	-
Convert flt. to flt.	-	-	-	-
Round flt. value to int.				
to nearest	-	-	-	-
toward zero	-	-	-	-
toward +infinity	-	-	-	-
toward -infinity	-	-	-	-

[1]Cvt* uses rounding mode specified in MXCSR. Cvtt* truncates the fractional part.

Table C.14
Data Extraction, Insertion, and Permutation Operations

Operation Types	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX
Extract Field to Reg.	-	-	-	-	-
Insert Selected Field	-	-	-	-	-
Insert Low Field	-	-	-	-	-
Byte Shft Rt & Extract					
By Immed.	extbl,extwl,extll,extql	-	-	-	-
By Register	extbl,extwl,extll,extql	-	-	-	-
Byte Shft Lt & Extract					
By Immed.	extwh,extlh,extqh	-	-	-	alni.ob, alni.qh
By Register	extwh,extlh,extqh	-	-	alnv.ps	alnv.ob, alnv.qh
Byte Shft Rt & Insert into Zeroed Reg	inswh,inslh,insqh	-	-	-	-
Byte Shft Lt & Insert into Zeroed Reg	insbl,inswl,insll,insql	-	-	-	-
Bit Shft Lt & Extract	-	(v)extrs (v)extru - -	extrw(,s), extrw,u[1] extrd(,s), extrd,u[2]	-	-
Merge, Bit Shft Rt & Extract	-	(v)shd	shrpw, shrpd	-	-
Bit Shift Left & Insert into Zeroed Reg					
from Immed	-	z(v)depi	depwi,z, depdi,z	-	-
from Reg	-	z(v)dep	depw,z, depd,z	-	-
Bit Shift Left & Insert into Unchanged Reg					
from Immed	-	(v)depi	depwi, depdi	-	-
from Reg	-	(v)dep	depw, depd	-	-
Clear Segment Low	mskbl,mskwl,mskll,mskql	-	-	-	-
Clear Segment High	mskwh,msklh,mskqh	-	-	-	-
Permute					
Part/Indexed by Part	-	-	-	-	-
Part/Indexed by Imm	-	-	permh	-	-
Swap Fields	-	-	-	-	-

[1]See table D-13, pD-9 in [82].
[2]See table D-14, pD-9 in [82].

Table C.14 cont'd.
Data Extraction, Insertion, and Permutation Operations

Operation Types	Motorola AltiVec	Sun VIS	Intel MMX	Intel SSE	Intel SSE2
Extract Field to Reg.	-	-	-	pextrw	pextrw
Insert Selected Field	-	-	-	pinsrw	pinsrw
Insert Low Field	-	-	-	movss	movsd
Byte Shft Rt & Extract					
By Immed.	vsldoi	-	-	-	-
By Register	-	faligndata	-	-	-
Byte Shft Lt & Extract					
By Immed.	-	-	-	-	-
By Register	-	-	-	-	-
Byte Shft Rt & Insert into Zeroed Reg	-	-	-	-	-
Byte Shft Lt & Insert into Zeroed Reg	-	-	-	-	-
Bit Shft Lt & Extract	-	-	-	-	-
Merge, Bit Shft Rt & Extract	-	-	-	-	-
Bit Shift Left & Insert into Zeroed Reg					
from Immed	-	-	-	-	-
from Reg	-	-	-	-	-
Bit Shift Left & Insert into Unchanged Reg					
from Immed	-	-	-	-	-
from Reg	-	-	-	-	-
Clear Segment Low	-	-	-	-	-
Clear Segment High	-	-	-	-	-
Permute					
Part/Indexed by Part	vperm	-	-	-	-
Part/Indexed by Imm	-	-	-	pshufw, shufps	pshuflw,pshufhw, pshufd, shufpd
Swap Fields	-	-	-	-	-

Table C.14 cont'd.
Data Extraction, Insertion, and Permutation Operations

Operation Types	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Extract Field to Reg.	-	pextrw	-	-
Insert Selected Field	-	pinsrw	-	-
Insert Low Field	-	-	movss	-
Byte Shft Rt & Extract				
By Immed.	-	-	-	-
By Register	-	-	-	-
Byte Shft Lt & Extract				
By Immed.	-	-	-	-
By Register	-	-	-	-
Byte Shft Rt & Insert into Zeroed Reg	-	-	-	-
Byte Shft Lt & Insert into Zeroed Reg	-	-	-	-
Bit Shft Lt & Extract	-	-	-	-
Merge, Bit Shft Rt & Extract	-	-	-	-
Bit Shift Left & Insert into Zeroed Reg				
from Immed	-	-	-	-
from Reg	-	-	-	-
Bit Shift Left & Insert into Unchanged Reg				
from Immed	-	-	-	-
from Reg	-	-	-	-
Clear Segment Low	-	-	-	-
Clear Segment High	-	-	-	-
Permute				
Part/Indexed by Part	-	-	-	-
Part/Indexed by Imm	-	pshufw	-	-
			shufps	
Swap Fields	-	pswapd	-	-

Table C.15
Interleaving Operations

Operation Types	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX	Motorola AltiVec	Sun VIS
Interleave (Merge)	-	-	-	-	-	-	fpmerge
Interleave odd (left)	-	-	mixh,l, mixw,l	-	-	-	-
Interleave even (right)	-	-	mixh,r, mixw,r	-	-	-	-
Interleave upper							
Part/Part	-	-	-	 puu.ps	shfl.mixh.ob, shfl.mixh.qh	vmrghb, vmrghh, vmrghw	-
Part/Imm	-	-	-	-	shfl.mixh.ob, shfl.mixh.qh	-	-
Part/Scalar	-	-	-	-	shfl.mixh.ob, shfl.mixh.qh	-	-
Part/Zero	-	-	-	-	shfl.upuh.ob	-	-
Interleave lower							
Part/Part	-	-	-	 pll.ps	shfl.mixl.ob, shfl.mixl.qh	vmrglb, vmrglh, vmrglw	-
Part/Imm	-	-	-	-	shfl.mixl.ob, shfl.mixl.qh	-	-
Part/Scalar	-	-	-	-	shfl.mixl.ob, shfl.mixl.qh	-	-
Part/Zero	-	-	-	-	shfl.upul.ob	-	-
Scale, Trunc., Clip & Merge	-	-	-	-	-	-	fpack32
Interleave even w/odd Forward or Reverse							
Part/Part	-	-	-	- plu.ps	shfl.bfl[ab].qh	-	-
Part/Imm	-	-	-	-	shfl.bfl[ab].qh	-	-
Part/Scalar	-	-	-	-	shfl.bfl[ab].qh	-	-
Interleave odd w/even Forward or Reverse							
Part/Part	-	-	-	pul.ps	-	-	-
Part/Imm	-	-	-	-	-	-	-
Part/Scalar	-	-	-	-	-	-	-

Table C.15 cont'd.
Interleaving Operations

Operation Types	Intel MMX	Intel SSE	Intel SSE2
Interleave (Merge)	-	-	-
Interleave odd (left)	-	-	-
Interleave even (right)	-	-	-
Interleave upper			
Part/Part	punpckhbw, punpckhwd, punpckhdq	unpckhps	punpckhbw, punpckhwd, punpckhdq, punpckhqdq,unpckhpd
Part/Imm	-	-	-
Part/Scalar	-	-	-
Part/Zero	-	-	-
Interleave lower			
Part/Part	punpcklbw, punpcklwd, punpckldq	unpcklps	punpcklbw, punpcklwd, punpckldq, punpcklqdq,unpcklpd
Part/Imm	-	-	-
Part/Scalar	-	-	-
Part/Zero	-	-	-
Scale, Trunc., Clip & Merge	-	-	-
Interleave even w/odd Forward and Reverse			
Part/Part	-	-	-
Part/Imm	-	-	-
Part/Scalar	-	-	-
Interleave odd w/even Forward and Reverse			
Part/Part	-	-	-
Part/Imm	-	-	-
Part/Scalar	-	-	-

Table C.15 cont'd.
Interleaving Operations

Operation Types	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Interleave (Merge)	-	-	-	-
Interleave odd (left)	-	-	-	-
Interleave even (right)	-	-	-	-
Interleave upper				
Part/Part	-	-	- unpckhps	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
Part/Zero	-	-	-	-
Interleave lower				
Part/Part	-	-	- unpcklps	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
Part/Zero	-	-	-	-
Scale, Trunc., Clip & Merge	-	-	-	-
Interleave even w/odd Forward and Reverse				
Part/Part	-	-	-	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
Interleave odd w/even Forward and Reverse				
Part/Part	-	-	-	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-

Table C.16
Catenating, Packing, and Unpacking Operations

Operation Types	DEC MVI	HP MAX	SGI MIPS-V	SGI MDMX	Motorola AltiVec
Catenate odd					
Part/Part	-	-	-	shfl.pach.ob, shfl.pach.qh	-
Part/Imm	-	-	-	shfl.pach.ob, shfl.pach.qh	-
Part/Scalar	-	-	-	shfl.pach.ob, shfl.pach.qh	-
Catenate even					
Part/Part	-	-	-	shfl.pacl.ob, shfl.pacl.qh	vpkuhum, vpkuwum
Part/Imm	-	-	-	shfl.pacl.ob, shfl.pacl.qh	-
Part/Scalar	-	-	-	shfl.pacl.ob, shfl.pacl.qh	-
Catenate upper					
Part/Part	-	-	-	shfl.repa.qh	-
Part/Imm	-	-	-	shfl.repa.qh	-
Part/Scalar	-	-	-	shfl.repa.qh	-
Catenate lower					
Part/Part	-	-	-	shfl.repb.qh	-
Part/Imm	-	-	-	shfl.repb.qh	-
Part/Scalar	-	-	-	shfl.repb.qh	-
Unsigned Saturate, Pack, and Catenate	-	-	-	-	vpkshus,vpkuhus, vpkswus,vpkuwus
Signed Saturate, Pack, and Catenate	-	-	-	-	vpkshss, vpkswss
Pixel Pack and Catenate	-	-	-	-	vpkpx
Truncate & Pack Low Byte	pklb, pkwb	-	-	-	-
Scale, Truncate, & Clip	-	-	-	-	-
Unpack Lower & Sign Extend	-	-	-	shfl.upsl.ob	vupklsb, vupklsh
Unpack Upper & Sign Extend	-	-	-	shfl.upsh.ob	vupkhsb, vupkhsh
Unpack Low Bytes & Zero Extend	unpkbl, unpkbw	-	-	-	-
Unpack Lower Pixel	-	-	-	-	vupklpx
Unpack Upper Pixel	-	-	-	-	vupkhpx
Zero Expand	-	-	-	-	-

Table C.16 cont'd.
Catenating, Packing, and Unpacking Operations

Operation Types	Sun VIS	Intel MMX	Intel SSE	Intel SSE2
Catenate odd				
Part/Part	-	-	-	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
Catenate even				
Part/Part	-	-	-	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
Catenate upper				
Part/Part	-	-	movhlps	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
Catenate lower				
Part/Part	-	-	movlhps	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
Unsigned Saturate, Pack, and Catenate	-	packuswb	-	packuswb
Signed Saturate, Pack, and Catenate	-	packsswb, packssdw	-	packsswb, packssdw
Pixel Pack and Catenate	-	-	-	-
Truncate & Pack Low Byte	-	-	-	-
Scale, Truncate, & Clip	fpack16, fpackfix	-	-	-
Unpack Lower & Sign Extend	-	-	-	-
Unpack Upper & Sign Extend	-	-	-	-
Unpack Low Bytes & Zero Extend	-	-	-	-
Unpack Lower Pixel	-	-	-	-
Unpack Upper Pixel	-	-	-	-
Zero Expand	fexpand	-	-	-

Table C.16 cont'd.
Catenating, Packing, and Unpacking Operations

Operation Types	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Catenate odd				
Part/Part	-	-	-	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
Catenate even				
Part/Part	-	-	-	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
Catenate upper				
Part/Part	-	-	movhlps	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
Catenate lower				
Part/Part	-	-	movlhps	-
Part/Imm	-	-	-	-
Part/Scalar	-	-	-	-
Unsigned Saturate, Pack, and Catenate	-	-	-	-
Signed Saturate, Pack, and Catenate	-	-	-	-
Pixel Pack and Catenate	-	-	-	-
Truncate & Pack Low Byte	-	-	-	-
Scale, Truncate, & Clip	-	-	-	-
Unpack Lower & Sign Extend	-	-	-	-
Unpack Upper & Sign Extend	-	-	-	-
Unpack Low Bytes & Zero Extend	-	-	-	-
Unpack Lower Pixel	-	-	-	-
Unpack Upper Pixel	-	-	-	-
Zero Expand	-	-	-	-

Table C.17
Memory Access Operations

Operation Types	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX	Motorola AltiVec
Load Aligned	ldbu, ldwu, ldl, ldq	ldb, ldh, ldw	ldb, ldh, ldw and ldwa, ldd and ldda	luxc1	-	lvebx, lvehx, lvewx, lvx or lvxl[1]
Load Unaligned	ldq_u	-	-	-	-	-
Load Field	-	-	-	-	-	-
Load Immediate	-	ldil	ldil	-	-	-
Load Zeros	-	-	-	-	-	-
Load All Ones	-	-	-	-	-	-
Load Alignment Vector	-	-	-	-	-	lvsl or lvsr
Store Aligned	stb, stw, stl, stq	stb, stb, stw	stb, sth, stw and stwa std and stda	suxc1	-	stvebx, stvehx, stvewx, stvx or stvxl[1]
Store Unaligned	stq_u	stbys	stby, stdby	-	-	-
Store Aligned w/Cache Flush	-	-	-	-	-	-
Masked Store						
by Bitmask	-	-	-	-	-	-
by msb of Part	-	-	-	-	-	-
Store Sync	wmb	-	-	-	-	-
Load Sync	-	-	-	-	-	-
Memory Sync	-	sync	-	-	-	-
Spin-wait Hint	-	-	-	-	-	-

[1]Hints that the reference will probably be the last to this cache block.

Table C.17 cont'd.
Memory Access Operations

Operation Types	Sun VIS	Intel MMX	Intel SSE	Intel SSE2
Load Aligned	lddfa,d0, lddfa,d2, lddfa,[7f]0	-	movaps	movdqa,movapd
Load Unaligned	-	movd, movq	movss, movhps, movlps, movups	movd, movq,movsd, movhpd, movlpd, movdqu,movupd
Load Field	-	-	pinsrw	pinsrw
Load Immediate	-	-	-	-
Load Zeros	fzeros, fzero	-	-	-
Load All Ones	fones, fone	-	-	-
Load Alignment Vector	-	-	-	-
Store Aligned	stdfa,d0, stdfa,d2, stdfa,[7f]0	-	movntq, movaps,movntps	movnti, movdqa,movntdq, movapd,movntpd
Store Unaligned	-	movd, movq	movss, movhps, movlps, movups	movd, movq,movsd, movhpd, movlpd, movdqu,movupd
Store Aligned w/Cache Flush	stdfa,e0	-	-	-
Masked Store				
by Bitmask	stdfa,c0, stdfa,c2, stdfa,c4	-	-	-
by msb of Part	-	-	maskmovq	maskmovdqu
Store Sync	-	-	sfence	-
Load Sync	-	-	-	lfence
Memory Sync	-	-	-	mfence
Spin-wait Hint	-	-	-	pause

Table C.17 cont'd.
Memory Access Operations

Operation Types	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Load Aligned	-	-	movaps	-
Load Unaligned	-	-	movss, movhps, movlps, movups	-
Load Field	-	pinsrw	-	-
Load Immediate	-	-	-	-
Load Zeros	-	-	-	-
Load All Ones	-	-	-	-
Load Alignment Vector	-	-	-	-
Store Aligned	-	movntq	movaps,movntps	-
Store Unaligned	-	-	movss, movhps, movlps, movups	-
Store Aligned w/Cache Flush	-	-	-	-
Masked Store				
by Bitmask	-	-	-	-
by msb of Part	-	maskmovq	-	-
Store Sync	-	sfence	-	-
Load Sync	-	-	-	-
Memory Sync	-	-	-	-
Spin-wait Hint	-	-	-	-

Table C.18
Cache Management Operations

Operation Types	DEC MVI	HP MAX-1	HP MAX-2	SGI MIPS-V	SGI MDMX	Motorola AltiVec	Sun VIS
Prefetch Data Line	-	-	ldd	-	-	-	-
Prefetch Data Line for Write	-	-	ldw	-	-	-	-
Prefetch Hint	fetch	-	-	-	-	dst	-
Prefetch Hint Transient	-	-	-	-	-	dstt	-
Store Hint	fetch_m	-	-	-	-	dstst	-
Store Hint Transient	wh64	-	-	-	-	dststt	-
Disassociate ID and Stream(s)	-	-	-	-	-	dss or dssall	-
Evict Hint	ecb	-	-	-	-	-	-
Flush Line	-	fdc,fic	-	-	-	-	-
Purge Line	-	pdc	-	-	-	-	-
Flush Cache	-	fdce,fice	-	-	-	-	-

Operation Types	Intel MMX	Intel SSE	Intel SSE2	AMD 3DNow!	AMD E3DNow!	AMD 3DNow!Pro	Cyrix EMMX
Prefetch Data Line	-	-	-	prefetch	-		-
Prefetch Data Line for Write	-	-	-	prefetchw	-		-
Prefetch Hint	-	prefetch*[1]	-	-	prefetch*[1]		-
Prefetch Hint Transient	-	-	-	-	-		-
Store Hint	-	-	-	-	-		-
Store Hint Transient	-	-	-	-	-		-
Disassociate ID and Stream	-	-	-	-	-		-
Evict Hint	-	-	-	-	-		-
Flush Line	-	-	Yes	-	-		-
Purge Line	-	-	-	-	-		-
Flush Cache	-	-	-	-	-		-

[1] prefetcht0, prefetcht1, prefetcht2, prefetchnta.

APPENDIX D

SCC INTERNAL PSEUDO-OPERATIONS

The following table lists the pseudo-ops used internally in the Scc compiler, along with the number of arguments each takes (i.e. the number of subtrees representing arguments which are attached to the pseudo-op's node. A "-" means that the construct has multiple children, but these are not considered to be arguments per se. "Null" means that the node is a leaf, or that it's arguments are provided in some other manner. "U" means that the operation is unary (one argument). "UR" means that it is a unary reduction (i.e. a unary that returns a single value. "Bi" means that the operation is binary (two arguments). "Tri" means that the operation is trinary (three arguments).

Table D.1
Scc Internal Pseudo-operations

Pseudo-op	Args.	Meaning
BLOCK	-	A block of code
BREAK	-	Break statement
CALL	-	Function call
CONTINUE	-	Continue statement
DO	-	Do statement
EVERYWHERE	-	Everywhere statement
EXPR	-	Expression
FOR	-	For statement
GOTO	-	Goto statement
IF	-	If statement
LABEL	-	Label
RETURN	-	Return statement
SEMI	-	An empty statement
WHERE	-	Where statement
WHILE	-	While statement
NUM	Null	A constant single number
VNUM	Null	A constant parallel number
SIZEOF	Null	sizeof operator
LVSL	Null	Load index vector for shift left (used for alignment in AltiVec)
LOAD	Null	Vector load
NEG	U	Parallel negate
RCP	U	Parallel reciprocal (or 1st step of 3 step operation)
NOT	U	Parallel bitwise-NOT (1's complement)
CAST	U	Type cast arg0
I2F	U	Parallel convert arg0 from integer to floating-point
F2I	U	Parallel convert arg0 from floating-point to integer
LNOT	U	Parallel logical NOT yielding -1 or 0
LEA	U	Load/calculate effective address and store in register
LOADR	U	Fragment load based on effective address in register
LOADRR	U	Fragment load based on effective address in a pair of registers
LOADX	U	Vector element load
STORE	U	Vector store
UNPACKL	U	Unpack and extend the lower fields of a source
UNPACKH	U	Unpack and extend the higher fields of a source
ALL	UR	Reduce logical-AND of arg0
ANY	UR	Reduce logical-OR of arg0
REDUCEADD	UR	Reduce add of arg0
REDUCEAND	UR	Reduce bitwise-AND of arg0
REDUCEAVG	UR	Reduce average of arg0
REDUCEMAX	UR	Reduce maximum of arg0
REDUCEMIN	UR	Reduce minimum of arg0
REDUCEMUL	UR	Reduce multiply of arg0
REDUCEOR	UR	Reduce bitwise-OR of arg0
REDUCEXOR	UR	Reduce bitwise-XOR of arg0

Table D.1 cont'd.
Scc Internal Pseudo-operations

Pseudo-op	Args.	Meaning
ADD	Bi	Parallel add
ADDH	Bi	Parallel add high (low bit is carry-out of add)
AVG	Bi	Parallel average
DIV	Bi	Parallel divide
MOD	Bi	Parallel modulus
MUL	Bi	Parallel multiply (low N bits of result of NxN)
MULEVEN	Bi	Parallel multiply (even N-bit fields yeilding 2N-bit result)
MULODD	Bi	Parallel multiply (odd N-bit fields yeilding 2N-bit result)
MULH	Bi	Parallel multiply high (high N bits of result)
MAX	Bi	Parallel maximum
MIN	Bi	Parallel minimum
RCP1	Bi	Parallel reciprocal (or 2nd step of 3 step operation)
RCP2	Bi	Parallel reciprocal (or 3rd step of 3 step operation)
SUB	Bi	Parallel subtract
AND	Bi	Parallel bitwise-AND
ANDN(x,y)	Bi	Parallel bitwise-AND with complement (Identical to AND(NOT x, y))
NOR	Bi	Parallel bitwise-NOR
OR	Bi	Parallel bitwise-OR
XOR	Bi	Parallel bitwise-XOR
EQ	Bi	Parallel == yielding -1 or 0
EQ_C	Bi	Parallel == yielding 1 or 0 (C-like result)
GE	Bi	Parallel >= yielding -1 or 0
GT	Bi	Parallel > yielding -1 or 0
GT_C	Bi	Parallel > yielding 1 or 0 (C-like result)
LE	Bi	Parallel <= yielding -1 or 0
LT	Bi	Parallel < yielding -1 or 0
NE	Bi	Parallel ! = yielding -1 or 0
LAND	Bi	Parallel logical AND yielding -1 or 0
LOR	Bi	Parallel logical OR yielding -1 or 0
STORER	Bi	Fragment store based on effective address in register
STORERR	Bi	Fragment store based on effective address in a pair of registers
STOREX	Bi	Vector element store
ROTATE	Bi	Vector rotate (inter-element rotate) (count>0 is left?)
SHIFT	Bi	Vector shift (inter-element shift) (count>0 is left?)
SHL	Bi	Parallel intra-element shift left
SHLBIT	Bi	Parallel fragment shift left by bits
SHLBYTE	Bi	Parallel fragment shift left by bytes
SHR	Bi	Parallel intra-element shift right
SHRBIT	Bi	Parallel fragment shift right by bits
SHRBYTE	Bi	Parallel fragment shift right by bytes
INTRLVLOW	Bi	Interleave lower fields of sources
INTRLVHIGH	Bi	Interleave higher fields of sources
INTRLVEVEN	Bi	Interleave even fields of sources
INTRLVODD	Bi	Interleave odd fields of sources

Table D.1 cont'd.
Scc Internal Pseudo-operations

Pseudo-op	Args.	Meaning
PACK	Bi	Catenate the even fields of sources (arg0 into low half)
PACKS2U	Bi	Catenate the signed, even? fields of sources (arg0 into low half?)
PERM	Bi	Permute arg0 indexed via arg1
REPL	Bi	Replicate field 'arg1' of 'arg0' in rest of fragment
PUTGET	Bi?	Unused
QUEST	Tri	Trinary construct (e.g. a? true:false)
TPERM	Tri	Permute arg0 and arg1 indexed via arg2

APPENDIX E

THE INTEGER EXPRESSION VALIDATION PROGRAM

The integer expression validation program is written using C preprocessor macros to minimize its size. In this form, it is about 500 lines long, so I only include some sections here with some empty lines removed. Below is the macro which is expanded to create the SWARC functions for testing an operation *op*, for vectors of *fields* elements of signed or unsigned (*sign*), *bits*-bit precision, using modular or saturation arithmetic (*ms*).

```
#define BINOP(name, op, bits, fields, sign, ms)                  \
void name(ms sign##signed int i, ms sign##signed int j,          \
        ms sign##signed int:bits[fields] c)                      \
{                                                                \
        ms sign##signed int:bits[fields] a;                      \
        ms sign##signed int:bits[fields] b;                      \
                                                                 \
        a = i;                                                   \
        b = j;                                                   \
        c = a op b;                                              \
}
```

The C versions of these operations are generated using a set of macros which are not included here. These must emulate the operations performed by the SWARC code, handling saturation and non-standard bits sizes correctly.

Debugging the Scc compiler using macro-generated code is particularly painful. Here is an example function generated by the macro shown above for adding 1-bit unsigned integer values using modular addition:

```
void add1um(modular unsigned int i, modular unsigned int j,
modular unsigned int :1[64] c) { modular unsigned int:1[64] a;
modular unsigned int:1[64] b; a = i; b = j; c = a + b; }
```

This is embedded in a longer line of code because the above macro is nested in another macro that generates all the functions for 1-bit unsigned modular data. This can be seen by typing "`make valid64-MMX`" in the Examples/Valid directory of the Scc compiler sources and looking at the file valid64.Si. I have never found a way to embed a newline or line-feed character in these macros to make the generated code more readable.

An example of the C code generated by the Scc compiler for an MMX-based target is given below. Here, we see the wrapper macros generated to execute the MMX instructions:

```
/* # 100 "valid64.Sc"*/
__inline__
void add1um(unsigned int *i,
unsigned int *j,
p64_t *c)
{
    extern p64_t cpool64[];
    register p64_t *_cpool = &(cpool64[0]);

    extern p64_t spool64[];
    register p64_t *_spool = &(spool64[0]);

    {
        volatile p64_t a[1];
        volatile p64_t b[1];
        movq_m2r(*(_cpool + 0), mm0);
        movq_m2r(*(((p64_t *) ((char *)i +0)) + 0), mm1);
        movq_m2r(*(_cpool + 2), mm2);
        pand_r2r(mm2, mm1);
        movq_r2r(mm0, mm3);
        psubd_r2r(mm1, mm0);
        movq_r2r(mm0, mm4);
        psllq_i2r(32, mm0);
        movq_m2r(*(((p64_t *) ((char *)j +0)) + 0), mm5);
        por_r2r(mm0, mm4);
        pand_r2r(mm5, mm2);
        psubd_r2r(mm2, mm3);
        movq_r2r(mm3, mm6);
        psllq_i2r(32, mm3);
        por_r2r(mm3, mm6);
        pxor_r2r(mm6, mm4);
        movq_r2m(mm4, *(((p64_t *) ((char *)c +0)) + 0));
    }
    emms();
}
```

In this code, a macro with a name that ends in “r2r” performs an operation on one or more registers and leaves its result in a register. A macro with an “m2r” suffix takes an operand from memory and leaves its result in a register. Also, a macro with an “r2m” suffix takes an operand from a register and leaves its result in memory. The objects “mm0”, “mm1”, etc. represent MMX registers. The type `p64_t` is a `union` type which represents a fragment of data. `emms()` must be called to put the system back into floating-point mode from MMX mode.

Each macro is replaced with one or more inline assembly statements. Here is the preprocessed code corresponding to the above function with some formatting changes:

```
__inline__
void add1um(unsigned int *i, unsigned int *j, p64_t *c)
{
    extern p64_t cpool64[]; register p64_t *_cpool = &(cpool64[0]);
    extern p64_t spool64[]; register p64_t *_spool = &(spool64[0]);
    {
        volatile p64_t a[1];
        volatile p64_t b[1];
        __asm__ __volatile__ ("movq" " %0, %%" "mm0" : : "m" (*(_cpool + 0)));
        __asm__ __volatile__ ("movq" " %0, %%" "mm1" : :
                              "m" (*(((p64_t *) ((char *)i +0)) + 0)));
        __asm__ __volatile__ ("movq" " %0, %%" "mm2" : : "m" (*(_cpool + 2)));
        __asm__ __volatile__ ("pand" " %" "mm2" ", %" "mm1");
        __asm__ __volatile__ ("movq" " %" "mm0" ", %" "mm3");
        __asm__ __volatile__ ("psubd" " %" "mm1" ", %" "mm0");
        __asm__ __volatile__ ("movq" " %" "mm0" ", %" "mm4");
        __asm__ __volatile__ ("psllq" " $" "32" ", %" "mm0");
        __asm__ __volatile__ ("movq" " %0, %%" "mm5" : :
                              "m" (*(((p64_t *) ((char *)j +0)) + 0)));
        __asm__ __volatile__ ("por" " %" "mm0" ", %" "mm4");
        __asm__ __volatile__ ("pand" " %" "mm5" ", %" "mm2");
        __asm__ __volatile__ ("psubd" " %" "mm2" ", %" "mm3");
        __asm__ __volatile__ ("movq" " %" "mm3" ", %" "mm6");
        __asm__ __volatile__ ("psllq" " $" "32" ", %" "mm3");
        __asm__ __volatile__ ("por" " %" "mm3" ", %" "mm6");
        __asm__ __volatile__ ("pxor" " %" "mm6" ", %" "mm4");
        __asm__ __volatile__ ("movq" " %%" "mm4" ", %0" :
                              "=m" (*(((p64_t *) ((char *)c +0)) + 0)) : );
    }
    __asm__ __volatile__ ("emms");
}
```

The above function is compiled by the C compiler to assembly. The assembly code generated is directly related to the C code generate by the Scc compiler. The final code is scheduled according to the Scc compiler's scheduling algorithm with the C compiler generating the code for handling the stack.

```
.globl add1um
        .type   add1um,@function
add1um:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %esi
        pushl   %ebx
        subl    $16, %esp
        movl    8(%ebp), %edx
        movl    12(%ebp), %ecx
        movl    16(%ebp), %ebx
        movl    $cpool64, %eax
        movl    $spool64, %esi
#APP
        movq (%eax), %mm0
        movq (%edx), %mm1
        movq 16(%eax), %mm2
        pand %mm2, %mm1
        movq %mm0, %mm3
        psubd %mm1, %mm0
        movq %mm0, %mm4
        psllq $32, %mm0
        movq (%ecx), %mm5
        por %mm0, %mm4
        pand %mm5, %mm2
        psubd %mm2, %mm3
        movq %mm3, %mm6
        psllq $32, %mm3
        por %mm3, %mm6
        pxor %mm6, %mm4
        movq %mm4, (%ebx)
        emms
#NO_APP
        addl    $16, %esp
        popl    %ebx
        popl    %esi
        popl    %ebp
        ret
```

APPENDIX F
THE DNA EXAMPLE BENCHMARK

This is the SWARC source with inlined C code for the DNA example benchmark. Note the lack of target-specific code in the SWARC sections of the source, and the exposure of the target architecture in the inlined C code.

```
${
$include <stdio.h>
$include <stdlib.h>
$include <time.h>
$include <sys/times.h>
$include <limits.h>
$include <time.h>
typedef unsigned long long int ullong;
$}

#include "common.h"

#if BPF == 128
        #define FRAGTYPE p128_t
        #define FRAGCTYPE ullong
        #define FRAGEXT uq
#elif BPF == 64
        #define FRAGTYPE p64_t
        #define FRAGCTYPE ullong
        #define FRAGEXT uq
        #define FRAGCONST ULL
#else
        #define FRAGTYPE p32_t
        #define FRAGCTYPE unsigned
        #define FRAGEXT ud
        #define FRAGCONST U
#endif

${
struct tms junk;
clock_t start, end, comptime;
$}
```

```
void f(unsigned int:2[LENGTH] DNA, unsigned int total)
{
        unsigned int:2[3] substring;
        unsigned int:2[LENGTH] count;
        unsigned int i;

#ifdef DEBUG_PEEK
        ${
                static p128_t output;
                {
        $}
#endif
#ifdef DEBUG_TOTAL
        ${
                printf("total=%u\n", *total);
        $}
#endif
#ifdef DEBUG_SUBSTRING
        ${
                substring[0].uq[0] = 0x0123456789abcdefULL;
                substring[0].uq[1] = 0xfedcba9876543210ULL;
                printf("substring[0]={0x%016llx,0x%016llx}\n",
                        substring[0].uq[0], substring[0].uq[1]);
        $}
#endif
        substring[0]=A; substring[1]=G; substring[2]=T;
#ifdef DEBUG_SUBSTRING
        ${
                printf("substring[0]={0x%016llx,0x%016llx}\n",
                        substring[0].uq[0], substring[0].uq[1]);
        $}
#endif
        count = 0;
#ifdef DEBUG_COUNT
        ${
                int frag, x;
                for (frag=0; frag<((LENGTH/(BPF/2))+1); ++frag) {
                        printf("count[%d]={0x%016llx,0x%016llx}\n",
                                frag, count[frag].uq[0], count[frag].uq[1]);
                }
                for (x=0; x<((LENGTH/(BPF/2))+1); ++x)
                        printf("  DNA[%d]={0x%016llx,0x%016llx}\n",
                                x, DNA[x].uq[0], DNA[x].uq[1]);
        $}
#endif
```

```
        for (i=0; i<3; ++i) {
#ifdef DEBUG_COUNT
                ${
                        int x;

                        printf("At top of loop: in memory order\n");
                        for (x=0; x<((LENGTH/(BPF/2))+1); ++x)
                                printf("count[%d]={0x%016llx,0x%016llx}\n",
                                        x, count[x].uq[0], count[x].uq[1]);
#ifdef DEBUG_SETCOUNTBYHAND
                        printf("After setting by hand: in memory order\n");
                        count[0].uq[0] = 0xffffffffffffffff;
                        count[0].uq[1] = 0x0000000000000000;
                        count[1].uq[0] = 0x0123456789abcdef;
                        count[1].uq[1] = 0xfedcba9876543210;
                        for (x=0; x<((LENGTH/(BPF/2))+1); ++x)
                                printf("count[%d]={0x%016llx,0x%016llx}\n",
                                        x, count[x].uq[0], count[x].uq[1]);
#endif
                $}
#endif
                count = count[<< 1];
#ifdef DEBUG_COUNT
                ${
                        int x;

                        printf("After shift: in memory order\n");
                        for (x=0; x<((LENGTH/(BPF/2))+1); ++x)
                                printf("count[%d]={0x%016llx,0x%016llx}\n",
                                        x, count[x].uq[0], count[x].uq[1]);
                $}
#endif
                count += (DNA == substring[i])? 1:0;
#ifdef DEBUG_COUNT
                ${
                        int x;

                        printf("At bottom of loop: in memory order\n");
                        for (x=0; x<((LENGTH/(BPF/2))+1); ++x)
                                printf("count[%d]={0x%016llx,0x%016llx}\n",
                                        x, count[x].uq[0], count[x].uq[1]);
                        printf("\n");
                $}
#endif
        }
```

```
#ifdef DEBUG_COUNT
        ${
                int x;

                printf("Just outside loop: in memory order\n");
                for (x=0; x<((LENGTH/(BPF/2))+1); ++x)
                        printf("count[%d]={0x%016llx,0x%016llx}\n",
                                x, count[x].uq[0], count[x].uq[1]);
        $}
#endif

        count = (count == 3)? 1:0;
#ifdef DEBUG_COUNT
        ${
                int x;

                printf("After marking full counts: in memory order\n");
                for (x=0; x<((LENGTH/(BPF/2))+1); ++x)
                        printf("count[%d]={0x%016llx,0x%016llx}\n",
                                x, count[x].uq[0], count[x].uq[1]);
        $}
#endif

        total += count;
#ifdef DEBUG_TOTAL
        ${
                printf("total=%u\n", *total);
        $}
#endif
#ifdef DEBUG_PEEK
        ${
                printf("output = {0x%016llx, 0x%016llx}\n",
                        output.uq[0], output.uq[1]);
                }
        $}
#endif
}

${
int main(void)
{
        int iters, i, j, k;
        unsigned int total = 0;
        FRAGTYPE DNA[((2*LENGTH-1)/BPF)+1];

        #ifdef TIME_OVERALL
```

```
        start = times(&junk);
#endif

#ifdef TIME_COMPUTE
        comptime = 0ULL;
#endif
srand(SEED);

for (iters=0; iters<ITERS; ++iters) {
        /* Full fragments - 32,31,...,0 */
        for (i=0; i<LENGTH/(BPF/2); ++i) {
                for (j=0; j<(BPF/2); ++j) {
                    #if BPF==128
                      DNA[i].FRAGEXT[0] = (DNA[i].FRAGEXT[0]>>2) |
                        (DNA[i].FRAGEXT[1] << ((BPF/2)-2));
                      DNA[i].FRAGEXT[1] = (DNA[i].FRAGEXT[1]>>2) |
                        ( ((FRAGCTYPE)(4.0*rand()/(RAND_MAX+1.0))&0x3)
                          << 62 );
                    #else
                      DNA[i].FRAGEXT = (DNA[i].FRAGEXT>>2) |
                        ( ((FRAGCTYPE)(4.0*rand()/(RAND_MAX+1.0))&0x3)
                          << BPF-2 );
                    #endif
                }
        }

        /* Final, possibly partially-filled, fragment */
        if (i == (2*LENGTH-1)/BPF) {
                #if BPF==128
                        DNA[i].FRAGEXT[1] = DNA[i].FRAGEXT[0] = 0ULL;
                #else
                        DNA[i].FRAGEXT = 0ULL;
                #endif
        }
        for (j=0; j<LENGTH%(BPF/2); ++j) {
                #if BPF==128
                    if (LENGTH%(BPF/2) > 32) {
                      /* Store in upper half */
                      DNA[i].FRAGEXT[0] = (DNA[i].FRAGEXT[0]>>2) |
                        (DNA[i].FRAGEXT[1] << ((BPF/2)-2));
                      DNA[i].FRAGEXT[1] = (DNA[i].FRAGEXT[1]>>2) |
                        ( ((FRAGCTYPE)(4.0*rand()/(RAND_MAX+1.0))&0x3)
                          << ((LENGTH%((BPF/2)/2))*2)-2);
                    } else {
                      /* Store in lower half */
                      DNA[i].FRAGEXT[1] = 0ULL;
```

```
                DNA[i].FRAGEXT[0] = (DNA[i].FRAGEXT[0]>>2) |
                  ( ((FRAGCTYPE)(4.0*rand()/(RAND_MAX+1.0))&0x3)
                    << ((LENGTH%((BPF/2)/2))*2)-2);
              }
        #else
            DNA[i].FRAGEXT = (DNA[i].FRAGEXT>>2) |
                ( ((FRAGCTYPE)(4.0*rand()/(RAND_MAX+1.0))&0x3)
                  << ((LENGTH%(BPF/2))*2)-2);
        #endif
}

#ifdef DEBUG
        printf("(Descending order) DNA[i]=");
        for (i=0; i<LENGTH; ++i) {
            #if BPF==128
                j = (LENGTH-1-i)/(BPF/2);
                k = (LENGTH-1-i)%(BPF/2);
                if (k >= (BPF/4)) {
                  /* Field is in upper half */
                  k -= (BPF/4);
                  printf("%llu ",
                         (DNA[j].FRAGEXT[1]>>2*k) & 0x3ULL);
                } else {
                  /* Field is in lower half */
                  printf("%llu ",
                         (DNA[j].FRAGEXT[0]>>2*k) & 0x3ULL);
                }
            #else
                j = (LENGTH-1-i)/(BPF/2);
                k = (LENGTH-1-i)%(BPF/2);
                printf("%d ",
                       (int)((DNA[j].FRAGEXT >> 2*k) &
                             0x3FRAGCONST));
            #endif
        }
        printf("\n");
#endif

#ifdef TIME_COMPUTE
        start = times(&junk);
#endif
f(DNA, &total);
#ifdef TIME_COMPUTE
        end = times(&junk);
        comptime += (end-start);
#endif
```

```
        }
        printf ("Total was %u.\n", total);

        #ifdef TIME_OVERALL
                end = times(&junk);
                printf("Time elapsed for %d element check: %ld (%ld, %ld)\n",
                        LENGTH, end-start, end, start);
        #endif
        #ifdef TIME_COMPUTE
                printf("Time elapsed for %d element check: %ld\n",
                        LENGTH, comptime);
        #endif

        return 0;
}
$}
```

The C versions of this program are similar to one another. The C character version is:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/times.h>
#include <limits.h>
#include <time.h>

#include "common.h"

struct tms junk;
clock_t start, end, comptime;

int f (char DNA[])
{
        char substring[3] = {A, G, T};
        char count[LENGTH];
        int total;
        int i, j;

        /* start = times(&junk); */

        for (i=0; i<LENGTH-2; ++i) count[i] = 0;

        total = 0;
```

```
	for (i=0; i<3; ++i)
		for (j=0; j<LENGTH-2; ++j)
			count[j] += (DNA[j+i] == substring[i]);

	for (i=0; i<LENGTH-2; ++i)
		total += (count[i] == 3);

	/* end = times(&junk); */
	/* printf("Time elapsed for %d element check: %ld (%ld, %ld)\n",
		LENGTH, end-start, end, start);
	*/

	return total;
}

int main(void)
{
	int iters;
	int i;
	int total = 0;
	char DNA[LENGTH];

	#ifdef TIME_OVERALL
		start = times(&junk);
	#endif

	#ifdef TIME_COMPUTE
		comptime = 0ULL;
	#endif

	srand(SEED);
	for (iters=0; iters<ITERS; ++iters) {
		for (i=0; i<LENGTH; ++i) {
			DNA[i] = (char)(4.0*rand()/(RAND_MAX+1.0));
		}
		#ifdef DEBUG
			printf("DNA[i]=");
			for (i=LENGTH-1; i>=0; --i) {
				printf("%x ", DNA[i]);
			}
			printf("\n");
		#endif

		#ifdef TIME_COMPUTE
			start = times(&junk);
		#endif
```

```
                total += f(DNA);
                #ifdef TIME_COMPUTE
                        end = times(&junk);
                        comptime += (end-start);
                #endif
        }
        printf ("Total was %d.\n", total);

        #ifdef TIME_OVERALL
                end = times(&junk);
                printf("Time elapsed for %d element check: %ld (%ld, %ld)\n",
                        LENGTH, end-start, end, start);
        #endif
        #ifdef TIME_COMPUTE
                printf("Time elapsed for %d element check: %ld\n",
                        LENGTH, comptime);
        #endif

        return 0;
}
```

The C integer version is:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/times.h>
#include <limits.h>
#include <time.h>

#include "common.h"

struct tms junk;
clock_t start, end, comptime;

int f (int DNA[])
{
        int substring[3] = {A, G, T};
        int count[LENGTH];
        int total;
        int i, j;

        /* start = times(&junk); */

        for (i=0; i<LENGTH-2; ++i) count[i] = 0;
```

```
        total = 0;

        for (i=0; i<3; ++i)
                for (j=0; j<LENGTH-2; ++j)
                        count[j] += (DNA[j+i] == substring[i]);

        for (i=0; i<LENGTH-2; ++i)
                total += (count[i] == 3);

        /* end = times(&junk); */
        /* printf("Time elapsed for %d element check: %ld (%ld, %ld)\n",
                LENGTH, end-start, end, start);
        */

        return total;
}

int main(void)
{
        int iters;
        int i;
        int total = 0;
        int DNA[LENGTH];

        #ifdef TIME_OVERALL
                start = times(&junk);
        #endif

        #ifdef TIME_COMPUTE
                comptime = 0ULL;
        #endif

        srand(SEED);
        for (iters=0; iters<ITERS; ++iters) {
                for (i=0; i<LENGTH; ++i) {
                        DNA[i] = (int)(4.0*rand()/(RAND_MAX+1.0));
                }
                #ifdef DEBUG
                        printf("DNA[i]=");
                        for (i=LENGTH-1; i>=0; --i) {
                                printf("%x ", DNA[i]);
                        }
                        printf("\n");
                #endif
```

```
                #ifdef TIME_COMPUTE
                        start = times(&junk);
                #endif
                total += f(DNA);
                #ifdef TIME_COMPUTE
                        end = times(&junk);
                        comptime += (end-start);
                #endif
        }
        printf ("Total was %d.\n", total);

        #ifdef TIME_OVERALL
                end = times(&junk);
                printf("Time elapsed for %d element check: %ld (%ld, %ld)\n",
                        LENGTH, end-start, end, start);
        #endif
        #ifdef TIME_COMPUTE
                printf("Time elapsed for %d element check: %ld\n",
                        LENGTH, comptime);
        #endif

        return 0;
}
```

The following file defines the parameters of the experimental run to ensure commonality between each version:

```
/* Parameters of run ****************************************************
   LENGTH is the length of the DNA vector to be searched,
   ITERS is the number of iterations that the main loop will be run,
   SEED is for random() to ensure all versions generate the same data.
*/
        #define LENGTH  350
        #define ITERS   1000000
        #define SEED    11

/* Choose one of the following for timimg information. ********************
   TIME_OVERALL includes time to initialize the data,
   TIME_COMPUTE does not.
*/
        #undef TIME_OVERALL
        #define TIME_COMPUTE

/* Define this to generate some debugging information ********************/
#undef DEBUG_PEEK
```

```
#undef DEBUG_SUBSTRING
#undef DEBUG_COUNT
#undef DEBUG_SETCOUNTBYHAND
#undef DEBUG_TOTAL

/* Values for the genes.  Do not change these. ***************************/
#define A       0
#define G       1
#define T       2
#define C       3
```

APPENDIX G
NUMERICAL RESULTS FOR DNA BENCHMARK

This appendix contains the numeric results of experiments in porting the dna.Sc benchmark program.

Speedup was calculated as the average time for the fastest C version divided by the average time for the version under test. In all cases, 10 trials were run and the running times of the counting function averaged. This separated the time to generate the random data and pack it into the correct layout from the actual processing time. This is reasonable under the assumption that measured data can be presented to the computer in an optimal layout by the measuring device.

To ensure that the timing averages are reasonably precise despite the relatively coarse-grained timing mechanism used, one million iterations of the loop were performed in each trial and the resultant timings averaged.

G.1 Results on AltiVec Target

The Scc-generated AltiVec code achieved speedup, though significantly less than one would hope given AltiVec's 128-bit registers and the 2-bit data. The optimal speedup would have been approximately 128/2 or 64x over serial 32-bit integer or 8-bit character code. The average speedup calculated from the measured trials ranged from about 3.8x to about 4.6x.

The results are presented in table G.1 below for Scc-generated code using 2-bit integers and employing various fragment sizes, compiler optimization levels, and optimization types; GCC-generated C code using 32-bit integers, and GCC-generated C code using 8-bit characters.

The numbers in this table were obtained by compiling and running each of the four versions on a 1GHz PowerBook G4 computer running LinuxPPC. No other applications (excluding normal services) were running, and no other users had access to the machine.

The best speedup was achieved by Scc-generated 32-bit integer C code. While this code was obviously incorrect (the calculated total is slightly off), it is remarkable because it does *not* use the AltiVec instruction set. The best speedup using the AltiVec instructions was 4.567, which is nearly as good. Given that the AltiVec registers are four times as large as the PowerPC's general registers, we would expect the 128-bit AltiVec SWAR code to be about four times as fast as the 32-bit SWAR code on the same platform.

G.2 Results for MMX Target

Scc-generated MMX code did not achieved speedup in any of the tests. The speedup calculated from the measured trials was between approximately 0.4x and 0.8x. These results are summarized in table G.2 below for 2-bit Scc-generated MMX code, 2-bit Scc-generated C-only code using the target's 32-bit general-purpose integer registers, GCC-generated C code using 32-bit integers, and GCC-generated C code using 8-bit characters.

The worst-case Scc code was generated without using any of the optimizations built into the compiler. The best case Scc code was generated without using the MMX registers, with Scc running at optimization level 0, and with Scc only performing back-end peephole optimizations. Thus, we might assume that the overhead of using the MMX-enhanced hardware was greater than the gains made. However, an inspection of the generated C code reveals that the MMX-based C code is hindered by the relatively small number of enhanced registers available. Scc's spill code is admittedly horrendous, so there is a high penalty for spills. This is probably the primary reason for the relatively poor performance of the MMX code.

Table G.1
AltiVec Trial Runs

Compiler	Data Type: Bits:Fragsize	Optimization Level(s)	Compiler Switches	Avg. Time	Speedup (x factor)	Calculated Total
Scc 010530	int:2:128	Scc 0 / GCC 3	-	439.5	4.458	5441660
Scc 010530	int:2:128	Scc 1 / GCC 3	-	437.7	4.476	5441660
Scc 010530	int:2:128	Scc 2 / GCC 3	-	429.0	4.567[1]	5441660
Scc 010530	int:2:128	Scc 3 / GCC 3	-	441.5	4.438	5441660
Scc 010530	int:2:32	Scc 0 / GCC 3	-	431.8	4.537	5440685
Scc 010530	int:2:32	Scc 1 / GCC 3	-	435.4	4.500	5440685
Scc 010530	int:2:32	Scc 2 / GCC 3	-	437.4	4.479	5440685
Scc 010530	int:2:32	Scc 3 / GCC 3	-	422.6	4.636[2]	5440685
Scc 010530	int:2:128	Scc 0 / GCC 3	–fe-bvt	437.7	4.476	5441660
Scc 010530	int:2:128	Scc 1 / GCC 3	–fe-bvt	446.0	4.393	5441660
Scc 010530	int:2:128	Scc 2 / GCC 3	–fe-bvt	436.0	4.494	5441660
Scc 010530	int:2:128	Scc 3 / GCC 3	–fe-bvt	441.9	4.434	5441660
Scc 010530	int:2:32	Scc 0 / GCC 3	–fe-bvt	438.9	4.464	5440685
Scc 010530	int:2:32	Scc 1 / GCC 3	–fe-bvt	438.6	4.476	5440685
Scc 010530	int:2:32	Scc 2 / GCC 3	–fe-bvt	436.2	4.492	5440685
Scc 010530	int:2:32	Scc 3 / GCC 3	–fe-bvt	435.6	4.498	5440685
Scc 010530	int:2:128	Scc 0 / GCC 3	–no-be-cofold	440.8	4.445	5441660
Scc 010530	int:2:128	Scc 1 / GCC 3	–no-be-cofold	434.3	4.511	5441660
Scc 010530	int:2:128	Scc 2 / GCC 3	–no-be-cofold	432.9	4.526	5441660
Scc 010530	int:2:128	Scc 3 / GCC 3	–no-be-cofold	437.6	4.477	5441660
Scc 010530	int:2:32	Scc 0 / GCC 3	–no-be-cofold	439.1	4.462	5440685
Scc 010530	int:2:32	Scc 1 / GCC 3	–no-be-cofold	441.3	4.440	5440685
Scc 010530	int:2:32	Scc 2 / GCC 3	–no-be-cofold	430.2	4.554	5440685
Scc 010530	int:2:32	Scc 3 / GCC 3	–no-be-cofold	442.7	4.426	5440685
Scc 010530	int:2:128	Scc 0 / GCC 3	–no-be-peep	456.7	4.290	5441660
Scc 010530	int:2:128	Scc 1 / GCC 3	–no-be-peep	443.2	4.421	5441660
Scc 010530	int:2:128	Scc 2 / GCC 3	–no-be-peep	455.1	4.305	5441660
Scc 010530	int:2:128	Scc 3 / GCC 3	–no-be-peep	452.9	4.326	5441660
Scc 010530	int:2:32	Scc 0 / GCC 3	–no-be-peep	453.6	4.319	5440685
Scc 010530	int:2:32	Scc 1 / GCC 3	–no-be-peep	457.1	4.286	5440685
Scc 010530	int:2:32	Scc 2 / GCC 3	–no-be-peep	452.4	4.331	5440685
Scc 010530	int:2:32	Scc 3 / GCC 3	–no-be-peep	452.0	4.335	5440685
Scc 010530	int:2:128	Scc 0 / GCC 3	–no-be-cofold –no-be-peep	440.0	4.435	5441660
Scc 010530	int:2:128	Scc 1 / GCC 3	–no-be-cofold –no-be-peep	449.3	4.361	5441660
Scc 010530	int:2:128	Scc 2 / GCC 3	–no-be-cofold –no-be-peep	451.0	4.344	5441660
Scc 010530	int:2:128	Scc 3 / GCC 3	–no-be-cofold –no-be-peep	459.3	4.266	5441660
Scc 010530	int:2:32	Scc 0 / GCC 3	–no-be-cofold –no-be-peep	457.4	4.283	5440685
Scc 010530	int:2:32	Scc 1 / GCC 3	–no-be-cofold –no-be-peep	456.6	4.291	5440685
Scc 010530	int:2:32	Scc 2 / GCC 3	–no-be-cofold –no-be-peep	460.7	4.253	5440685
Scc 010530	int:2:32	Scc 3 / GCC 3	–no-be-cofold –no-be-peep	446.9	4.384	5440685

[1] Best dna128
[2] Best dna32, best overall

Table G.1 cont'd.
AltiVec Trial Runs

Compiler	Data Type: Bits:Fragsize	Optimization Level(s)	Compiler Switches	Avg. Time	Speedup (x factor)	Calculated Total
Scc 010530	int:2:128	Scc 0 / GCC 3	–no-fe-cofold –no-be-cofold	445.1	4.402	5441660
Scc 010530	int:2:128	Scc 1 / GCC 3	–no-fe-cofold –no-be-cofold	438.8	4.465	5441660
Scc 010530	int:2:128	Scc 2 / GCC 3	–no-fe-cofold –no-be-cofold	445.6	4.397	5441660
Scc 010530	int:2:128	Scc 3 / GCC 3	–no-fe-cofold –no-be-cofold	443.5	4.418	5441660
Scc 010530	int:2:32	Scc 0 / GCC 3	–no-fe-cofold –no-be-cofold	440.7	4.446	5440685
Scc 010530	int:2:32	Scc 1 / GCC 3	–no-fe-cofold –no-be-cofold	444.9	4.404	5440685
Scc 010530	int:2:32	Scc 2 / GCC 3	–no-fe-cofold –no-be-cofold	451.3	4.341	5440685
Scc 010530	int:2:32	Scc 3 / GCC 3	–no-fe-cofold –no-be-cofold	451.7	4.337	5440685
Scc 010530	int:2:128	Scc 0 / GCC 3	–no-fe-cofold	445.5	4.398	5441660
Scc 010530	int:2:128	Scc 1 / GCC 3	–no-fe-cofold	446.3	4.390	5441660
Scc 010530	int:2:128	Scc 2 / GCC 3	–no-fe-cofold	454.7	4.309	5441660
Scc 010530	int:2:128	Scc 3 / GCC 3	–no-fe-cofold	446.7	4.386	5441660
Scc 010530	int:2:32	Scc 0 / GCC 3	–no-fe-cofold	446.5	4.388	5440685
Scc 010530	int:2:32	Scc 1 / GCC 3	–no-fe-cofold	436.6	4.487	5440685
Scc 010530	int:2:32	Scc 2 / GCC 3	–no-fe-cofold	442.5	4.428	5440685
Scc 010530	int:2:32	Scc 3 / GCC 3	–no-fe-cofold	453.1	4.324	5440685
Scc 010530	int:2:128	Scc 0 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	465.6	4.208	5441660
Scc 010530	int:2:128	Scc 1 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	471.4	4.156	5441660
Scc 010530	int:2:128	Scc 2 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	460.2	4.257	5441660
Scc 010530	int:2:128	Scc 3 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	470.5	4.164	5441660
Scc 010530	int:2:32	Scc 0 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	491.4	3.987	5440685
Scc 010530	int:2:32	Scc 1 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	509.1	3.848[1]	5440685
Scc 010530	int:2:32	Scc 2 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	489.6	4.002	5440685
Scc 010530	int:2:32	Scc 3 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	475.3	4.122	5440685
GCC 2.95.3	char:8:32	GCC 0	-	7148.9	0.274	5441660
GCC 2.95.3	char:8:32	GCC 1	-	2222.2	0.882	5441660
GCC 2.95.3	char:8:32	GCC 2	-	1959.2	1.000[2]	5441660
GCC 2.95.3	char:8:32	GCC 3	-	1960.0	1.000	5441660
GCC 2.95.3	int:32:32	GCC 0	-	8130.2	0.241[3]	5441660
GCC 2.95.3	int:32:32	GCC 1	-	2835.1	0.691	5441660
GCC 2.95.3	int:32:32	GCC 2	-	1961.1	0.999[4]	5441660
GCC 2.95.3	int:32:32	GCC 3	-	1967.9	0.996	5441660

[1]Worst Scc-compiled
[2]Best C char, best GCC-compiled
[3]Worst GCC-compiled
[4]Best C int

The numbers in this table were obtained by compiling and running each of the four versions on a Pentium4 computer running Redhat Linux 7.0 with kernel version 2.2.16-22. No other applications (excluding normal services) were running, and no other users had access to the machine.

Correct operation of the Scc-generated MMX code was assumed to be verified by comparing the results with the GCC-generated C versions and finding no difference in the calculated totals. Note that there is no difference in the results of the Scc-generated non-MMX code and the GCC-generated code.

G.3 Results for 3DNow! Target

The Scc-generated 3DNow! code also achieved speedup; again significantly less than the theoretical maximum of 64/2 or 32x over serial 32-bit integer or 8-bit character code, but more than the AltiVec code and significantly more than the MMX code.

The speedup calculated for Scc-generated code ranged from approximately 3.9x to 5.1x. The results are summarized in table G.3 for 2-bit Scc-generated 3DNow! code, 2-bit Scc-generated C-only code using the target's 32-bit general-purpose registers, GCC-generated C code using 32-bit integers, and GCC-generated C code using 8-bit characters.

3DNow! suffers from the same problems as MMX in relation to register spills. Interestingly though, the 3DNow! trials all obtained speedup over the best GCC-generated C code. This is a significant difference in two relatively similar architectures. The reason for this needs to be studied.

G.4 Results for IA32 Target

Scc-generated IA32 code achieved speedup in only one case, but not by a significant amount over the best GCC-generated C code. In the majority of cases, the Scc-generated code was actually slower. This is to be expected because the architecture

Table G.2
MMX Trial Runs

Compiler	Data Type: Bits:Fragsize	Optimization Level(s)	Compiler Switches	Avg. Time	Speedup (x factor)	Calculated Total
Scc 010530	int:2:64	Scc 0 / GCC 3	-	908.2	0.810	5441660
Scc 010530	int:2:64	Scc 1 / GCC 3	-	904.2	0.813	5441660
Scc 010530	int:2:64	Scc 2 / GCC 3	-	891.2	0.825[1]	5441660
Scc 010530	int:2:64	Scc 3 / GCC 3	-	902.7	0.815	5441660
Scc 010530	int:2:32	Scc 0 / GCC 3	-	981.2	0.749	5441660
Scc 010530	int:2:32	Scc 1 / GCC 3	-	979.3	0.751	5441660
Scc 010530	int:2:32	Scc 2 / GCC 3	-	983.0	0.748	5441660
Scc 010530	int:2:32	Scc 3 / GCC 3	-	989.3	0.743	5441660
Scc 010530	int:2:64	Scc 0 / GCC 3	–fe-bvt	930.9	0.790	5441660
Scc 010530	int:2:64	Scc 1 / GCC 3	–fe-bvt	924.5	0.795	5441660
Scc 010530	int:2:64	Scc 2 / GCC 3	–fe-bvt	919.3	0.800	5441660
Scc 010530	int:2:64	Scc 3 / GCC 3	–fe-bvt	938.8	0.783	5441660
Scc 010530	int:2:32	Scc 0 / GCC 3	–fe-bvt	933.8	0.787	5441660
Scc 010530	int:2:32	Scc 1 / GCC 3	–fe-bvt	939.0	0.783	5441660
Scc 010530	int:2:32	Scc 2 / GCC 3	–fe-bvt	934.7	0.787	5441660
Scc 010530	int:2:32	Scc 3 / GCC 3	–fe-bvt	952.9	0.772	5441660
Scc 010530	int:2:64	Scc 0 / GCC 3	–no-be-cofold –no-be-peep	1154.2	0.637	5441660
Scc 010530	int:2:64	Scc 1 / GCC 3	–no-be-cofold –no-be-peep	1147.0	0.641	5441660
Scc 010530	int:2:64	Scc 2 / GCC 3	–no-be-cofold –no-be-peep	1160.0	0.634	5441660
Scc 010530	int:2:64	Scc 3 / GCC 3	–no-be-cofold –no-be-peep	1180.9	0.623	5441660
Scc 010530	int:2:32	Scc 0 / GCC 3	–no-be-cofold –no-be-peep	1096.4	0.671	5441660
Scc 010530	int:2:32	Scc 1 / GCC 3	–no-be-cofold –no-be-peep	1079.8	0.681	5441660
Scc 010530	int:2:32	Scc 2 / GCC 3	–no-be-cofold –no-be-peep	1085.6	0.677	5441660
Scc 010530	int:2:32	Scc 3 / GCC 3	–no-be-cofold –no-be-peep	1090.5	0.674	5441660
Scc 010530	int:2:64	Scc 0 / GCC 3	–no-be-cofold	959.4	0.766	5441660
Scc 010530	int:2:64	Scc 1 / GCC 3	–no-be-cofold	968.1	0.760	5441660
Scc 010530	int:2:64	Scc 2 / GCC 3	–no-be-cofold	959.8	0.766	5441660
Scc 010530	int:2:64	Scc 3 / GCC 3	–no-be-cofold	974.4	0.755	5441660
Scc 010530	int:2:32	Scc 0 / GCC 3	–no-be-cofold	968.5	0.759	5441660
Scc 010530	int:2:32	Scc 1 / GCC 3	–no-be-cofold	968.7	0.759	5441660
Scc 010530	int:2:32	Scc 2 / GCC 3	–no-be-cofold	991.3	0.742	5441660
Scc 010530	int:2:32	Scc 3 / GCC 3	–no-be-cofold	998.7	0.736	5441660
Scc 010530	int:2:64	Scc 0 / GCC 3	–no-be-peep	1130.9	0.650	5441660
Scc 010530	int:2:64	Scc 1 / GCC 3	–no-be-peep	1116.6	0.659	5441660
Scc 010530	int:2:64	Scc 2 / GCC 3	–no-be-peep	1107.6	0.664	5441660
Scc 010530	int:2:64	Scc 3 / GCC 3	–no-be-peep	1117.3	0.658	5441660
Scc 010530	int:2:32	Scc 0 / GCC 3	–no-be-peep	1069.9	0.687	5441660
Scc 010530	int:2:32	Scc 1 / GCC 3	–no-be-peep	1089.0	0.675	5441660
Scc 010530	int:2:32	Scc 2 / GCC 3	–no-be-peep	1074.4	0.684	5441660
Scc 010530	int:2:32	Scc 3 / GCC 3	–no-be-peep	1071.5	0.686	5441660

[1] Best dna64

Table G.2 cont'd.
MMX Trial Runs

Compiler	Data Type: Bits:Fragsize	Optimization Level(s)	Compiler Switches	Avg. Time	Speedup (x factor)	Calculated Total
Scc 010530	int:2:64	Scc 0 / GCC 3	–no-fe-cofold –no-be-cofold	931.9	0.789	5441660
Scc 010530	int:2:64	Scc 1 / GCC 3	–no-fe-cofold –no-be-cofold	942.8	0.780	5441660
Scc 010530	int:2:64	Scc 2 / GCC 3	–no-fe-cofold –no-be-cofold	948.1	0.776	5441660
Scc 010530	int:2:64	Scc 3 / GCC 3	–no-fe-cofold –no-be-cofold	930.1	0.791	5441660
Scc 010530	int:2:32	Scc 0 / GCC 3	–no-fe-cofold –no-be-cofold	885.4	0.830[1]	5441660
Scc 010530	int:2:32	Scc 1 / GCC 3	–no-fe-cofold –no-be-cofold	897.9	0.819	5441660
Scc 010530	int:2:32	Scc 2 / GCC 3	–no-fe-cofold –no-be-cofold	900.6	0.816	5441660
Scc 010530	int:2:32	Scc 3 / GCC 3	–no-fe-cofold –no-be-cofold	897.9	0.819	5441660
Scc 010530	int:2:64	Scc 0 / GCC 3	–no-fe-cofold	963.7	0.763	5441660
Scc 010530	int:2:64	Scc 1 / GCC 3	–no-fe-cofold	959.9	0.766	5441660
Scc 010530	int:2:64	Scc 2 / GCC 3	–no-fe-cofold	950.6	0.774	5441660
Scc 010530	int:2:64	Scc 3 / GCC 3	–no-fe-cofold	963.5	0.763	5441660
Scc 010530	int:2:32	Scc 0 / GCC 3	–no-fe-cofold	984.4	0.747	5441660
Scc 010530	int:2:32	Scc 1 / GCC 3	–no-fe-cofold	971.0	0.757	5441660
Scc 010530	int:2:32	Scc 2 / GCC 3	–no-fe-cofold	967.0	0.760	5441660
Scc 010530	int:2:32	Scc 3 / GCC 3	–no-fe-cofold	964.9	0.762	5441660
Scc 010530	int:2:64	Scc 0 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	1182.1	0.622	5441660
Scc 010530	int:2:64	Scc 1 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	1977.0	0.372[2]	5441660
Scc 010530	int:2:64	Scc 2 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	1191.7	0.617	5441660
Scc 010530	int:2:64	Scc 3 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	1194.5	0.616	5441660
Scc 010530	int:2:32	Scc 0 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	1453.9	0.506	5441660
Scc 010530	int:2:32	Scc 1 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	1455.1	0.505	5441660
Scc 010530	int:2:32	Scc 2 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	1435.8	0.512	5441660
Scc 010530	int:2:32	Scc 3 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	1406.8	0.523	5441660
GCC 2.96	char:8:32	GCC 0	-	1675.8	0.439	5441660
GCC 2.96	char:8:32	GCC 1	-	968.4	0.759	5441660
GCC 2.96	char:8:32	GCC 2	-	735.3	1.000[3]	5441660
GCC 2.96	char:8:32	GCC 3	-	785.7	0.936	5441660
GCC 2.96	int:32:32	GCC 0	-	2477.0	0.297[4]	5441660
GCC 2.96	int:32:32	GCC 1	-	1046.6	0.703	5441660
GCC 2.96	int:32:32	GCC 2	-	904.7	0.813[5]	5441660
GCC 2.96	int:32:32	GCC 3	-	912.1	0.806	5441660

[1] Best dna32, best Scc-compiled
[2] Worst Scc-compiled
[3] Best C char, best overall
[4] Worst overall
[5] Best C int

Table G.3
3DNow! Trial Runs

Compiler	Data Type: Bits:Fragsize	Optimization Level(s)	Compiler Switches	Avg. Time	Speedup (x factor)	Calculated Total
Scc 010530	int:2:64	Scc 0 / GCC 3	-	1122.7	4.708	5441660
Scc 010530	int:2:64	Scc 1 / GCC 3	-	1095.0	4.827	5441660
Scc 010530	int:2:64	Scc 2 / GCC 3	-	1114.0	4.744	5441660
Scc 010530	int:2:64	Scc 3 / GCC 3	-	1115.7	4.737	5441660
Scc 010530	int:2:32	Scc 0 / GCC 3	-	1065.7	4.959	5441660
Scc 010530	int:2:32	Scc 1 / GCC 3	-	1057.1	5.000	5441660
Scc 010530	int:2:32	Scc 2 / GCC 3	-	1059.0	4.991	5441660
Scc 010530	int:2:32	Scc 3 / GCC 3	-	1065.3	4.961	5441660
Scc 010530	int:2:64	Scc 0 / GCC 3	–fe-bvt	1080.3	4.892[1]	5441660
Scc 010530	int:2:64	Scc 1 / GCC 3	–fe-bvt	1082.9	4.881	5441660
Scc 010530	int:2:64	Scc 2 / GCC 3	–fe-bvt	1090.4	4.847	5441660
Scc 010530	int:2:64	Scc 3 / GCC 3	–fe-bvt	1099.7	4.806	5441660
Scc 010530	int:2:32	Scc 0 / GCC 3	–fe-bvt	1054.9	5.010	5441660
Scc 010530	int:2:32	Scc 1 / GCC 3	–fe-bvt	1061.6	4.979	5441660
Scc 010530	int:2:32	Scc 2 / GCC 3	–fe-bvt	1040.1	5.082[2]	5441660
Scc 010530	int:2:32	Scc 3 / GCC 3	–fe-bvt	1060.1	4.986	5441660
Scc 010530	int:2:64	Scc 0 / GCC 3	–no-be-cofold –no-be-peep	1308.9	4.038	5441660
Scc 010530	int:2:64	Scc 1 / GCC 3	–no-be-cofold –no-be-peep	1295.7	4.079	5441660
Scc 010530	int:2:64	Scc 2 / GCC 3	–no-be-cofold –no-be-peep	1304.1	4.053	5441660
Scc 010530	int:2:64	Scc 3 / GCC 3	–no-be-cofold –no-be-peep	1299.5	4.067	5441660
Scc 010530	int:2:32	Scc 0 / GCC 3	–no-be-cofold –no-be-peep	1306.9	4.044	5441660
Scc 010530	int:2:32	Scc 1 / GCC 3	–no-be-cofold –no-be-peep	1279.7	4.130	5441660
Scc 010530	int:2:32	Scc 2 / GCC 3	–no-be-cofold –no-be-peep	1297.1	4.075	5441660
Scc 010530	int:2:32	Scc 3 / GCC 3	–no-be-cofold –no-be-peep	1308.2	4.040	5441660
Scc 010530	int:2:64	Scc 0 / GCC 3	–no-be-cofold	1095.4	4.825	5441660
Scc 010530	int:2:64	Scc 1 / GCC 3	–no-be-cofold	1096.6	4.820	5441660
Scc 010530	int:2:64	Scc 2 / GCC 3	–no-be-cofold	1091.7	4.841	5441660
Scc 010530	int:2:64	Scc 3 / GCC 3	–no-be-cofold	1095.7	4.824	5441660
Scc 010530	int:2:32	Scc 0 / GCC 3	–no-be-cofold	1057.9	4.996	5441660
Scc 010530	int:2:32	Scc 1 / GCC 3	–no-be-cofold	1055.4	5.008	5441660
Scc 010530	int:2:32	Scc 2 / GCC 3	–no-be-cofold	1040.9	5.078	5441660
Scc 010530	int:2:32	Scc 3 / GCC 3	–no-be-cofold	1051.5	5.026	5441660
Scc 010530	int:2:64	Scc 0 / GCC 3	–no-be-peep	1295.2	4.081	5441660
Scc 010530	int:2:64	Scc 1 / GCC 3	–no-be-peep	1277.4	4.138	5441660
Scc 010530	int:2:64	Scc 2 / GCC 3	–no-be-peep	1289.1	4.100	5441660
Scc 010530	int:2:64	Scc 3 / GCC 3	–no-be-peep	1282.0	4.123	5441660
Scc 010530	int:2:32	Scc 0 / GCC 3	–no-be-peep	1281.9	4.123	5441660
Scc 010530	int:2:32	Scc 1 / GCC 3	–no-be-peep	1297.6	4.073	5441660
Scc 010530	int:2:32	Scc 2 / GCC 3	–no-be-peep	1256.6	4.206	5441660
Scc 010530	int:2:32	Scc 3 / GCC 3	–no-be-peep	1251.0	4.225	5441660

[1] Best dna64
[2] Best dna32, best overall

Table G.3 cont'd.
3DNow! Trial Runs

Compiler	Data Type: Bits:Fragsize	Optimization Level(s)	Compiler Switches	Avg. Time	Speedup (x factor)	Calculated Total
Scc 010530	int:2:64	Scc 0 / GCC 3	–no-fe-cofold –no-be-cofold	1126.9	4.690	5441660
Scc 010530	int:2:64	Scc 1 / GCC 3	–no-fe-cofold –no-be-cofold	1146.7	4.609	5441660
Scc 010530	int:2:64	Scc 2 / GCC 3	–no-fe-cofold –no-be-cofold	1128.9	4.682	5441660
Scc 010530	int:2:64	Scc 3 / GCC 3	–no-fe-cofold –no-be-cofold	1130.0	4.677	5441660
Scc 010530	int:2:32	Scc 0 / GCC 3	–no-fe-cofold –no-be-cofold	1084.6	4.873	5441660
Scc 010530	int:2:32	Scc 1 / GCC 3	–no-fe-cofold –no-be-cofold	1105.1	4.783	5441660
Scc 010530	int:2:32	Scc 2 / GCC 3	–no-fe-cofold –no-be-cofold	1104.7	4.784	5441660
Scc 010530	int:2:32	Scc 3 / GCC 3	–no-fe-cofold –no-be-cofold	1100.7	4.802	5441660
Scc 010530	int:2:64	Scc 0 / GCC 3	–no-fe-cofold	1138.8	4.641	5441660
Scc 010530	int:2:64	Scc 1 / GCC 3	–no-fe-cofold	1137.5	4.646	5441660
Scc 010530	int:2:64	Scc 2 / GCC 3	–no-fe-cofold	1124.6	4.700	5441660
Scc 010530	int:2:64	Scc 3 / GCC 3	–no-fe-cofold	1114.7	4.741	5441660
Scc 010530	int:2:32	Scc 0 / GCC 3	–no-fe-cofold	1097.2	4.817	5441660
Scc 010530	int:2:32	Scc 1 / GCC 3	–no-fe-cofold	1121.8	4.711	5441660
Scc 010530	int:2:32	Scc 2 / GCC 3	–no-fe-cofold	1120.0	4.719	5441660
Scc 010530	int:2:32	Scc 3 / GCC 3	–no-fe-cofold	1111.3	4.756	5441660
Scc 010530	int:2:64	Scc 0 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	1332.0	3.968	5441660
Scc 010530	int:2:64	Scc 1 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	1346.3	3.926[1]	5441660
Scc 010530	int:2:64	Scc 2 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	1327.5	3.981	5441660
Scc 010530	int:2:64	Scc 3 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	1342.9	3.936	5441660
Scc 010530	int:2:32	Scc 0 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	1289.7	4.098	5441660
Scc 010530	int:2:32	Scc 1 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	1309.0	4.038	5441660
Scc 010530	int:2:32	Scc 2 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	1292.0	4.091	5441660
Scc 010530	int:2:32	Scc 3 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	1319.1	4.007	5441660
egcs 2.91.66	char:8:32	GCC 0	-	14800.7	0.357	5441660
egcs 2.91.66	char:8:32	GCC 1	-	5285.3	1.000[2]	5441660
egcs 2.91.66	char:8:32	GCC 2	-	5385.9	0.981	5441660
egcs 2.91.66	char:8:32	GCC 3	-	6064.3	0.872	5441660
egcs 2.91.66	int:32:32	GCC 0	-	15580.7	0.339[3]	5441660
egcs 2.91.66	int:32:32	GCC 1	-	6697.4	0.789[4]	5441660
egcs 2.91.66	int:32:32	GCC 2	-	7037.3	0.751	5441660
egcs 2.91.66	int:32:32	GCC 3	-	7311.3	0.723	5441660

[1] Worst Scc-compiled
[2] Best C char, best GCC
[3] Worst GCC-compiled
[4] Best C int

does not provide any form of SWAR instructions other than the basic polymorphics (bitwise logical operations). However, this isn't the point of porting this code to an unenhanced 32-bit architecture. The point proven here is that the SWARC code can be ported to an unenhanced architecture without modification.

The speedup for Scc-generated code ranged from approximately 0.42x to 1.03x. It is worth noting that the GCC-generated code achieved speedups ranging from 0.28x to 1.00x. Thus, the choice of compiler switches appears to affect the performance more than the choice between Scc and GCC. The results are summarized in table G.4 for 2-bit Scc-generated C-only code using 32-bit integer fragments in the general registers, GCC-generated C code using 32-bit integers, and GCC-generated C code using 8-bit characters.

Table G.4
IA32 Trial Runs

Compiler	Data Type: Bits:Fragsize	Optimization Level(s)	Compiler Switches	Avg. Time	Speedup (x factor)	Calculated Total
Scc 010530	int:2:32	Scc 0 / GCC 3	-	9029.7	0.801	5435001
Scc 010530	int:2:32	Scc 1 / GCC 3	-	9029.9	0.801	5435001
Scc 010530	int:2:32	Scc 2 / GCC 3	-	8869.6	0.816	5435001
Scc 010530	int:2:32	Scc 3 / GCC 3	-	8736.8	0.828	5435001
Scc 010530	int:2:32	Scc 0 / GCC 3	–fe-bvt	8993.0	0.804	5435001
Scc 010530	int:2:32	Scc 1 / GCC 3	–fe-bvt	13259.8	0.546	5435001
Scc 010530	int:2:32	Scc 2 / GCC 3	–fe-bvt	8977.9	0.806	5435001
Scc 010530	int:2:32	Scc 3 / GCC 3	–fe-bvt	12811.6	0.565	5435001
Scc 010530	int:2:32	Scc 0 / GCC 3	–no-be-cofold –no-be-peep	11067.3	0.654	5435001
Scc 010530	int:2:32	Scc 1 / GCC 3	–no-be-cofold –no-be-peep	17094.1	0.423[1]	5435001
Scc 010530	int:2:32	Scc 2 / GCC 3	–no-be-cofold –no-be-peep	11543.7	0.627	5435001
Scc 010530	int:2:32	Scc 3 / GCC 3	–no-be-cofold –no-be-peep	11063.0	0.654	5435001
Scc 010530	int:2:32	Scc 0 / GCC 3	–no-be-cofold	13329.4	0.543	5435001
Scc 010530	int:2:32	Scc 1 / GCC 3	–no-be-cofold	8949.7	0.808	5435001
Scc 010530	int:2:32	Scc 2 / GCC 3	–no-be-cofold	9105.7	0.794	5435001
Scc 010530	int:2:32	Scc 3 / GCC 3	–no-be-cofold	9129.2	0.792	5435001
Scc 010530	int:2:32	Scc 0 / GCC 3	–no-be-peep	11414.6	0.634	5435001
Scc 010530	int:2:32	Scc 1 / GCC 3	–no-be-peep	15728.9	0.460	5435001
Scc 010530	int:2:32	Scc 2 / GCC 3	–no-be-peep	11213.7	0.645	5435001
Scc 010530	int:2:32	Scc 3 / GCC 3	–no-be-peep	11477.7	0.630	5435001
Scc 010530	int:2:32	Scc 0 / GCC 3	–no-fe-cofold –no-be-cofold	11583.6	0.625	5435001
Scc 010530	int:2:32	Scc 1 / GCC 3	–no-fe-cofold –no-be-cofold	8859.9	0.817	5435001
Scc 010530	int:2:32	Scc 2 / GCC 3	–no-fe-cofold –no-be-cofold	8615.7	0.840	5435001
Scc 010530	int:2:32	Scc 3 / GCC 3	–no-fe-cofold –no-be-cofold	8841.0	0.818	5435001
Scc 010530	int:2:32	Scc 0 / GCC 3	–no-fe-cofold	7037.6	1.028[2]	5435001
Scc 010530	int:2:32	Scc 1 / GCC 3	–no-fe-cofold	9833.9	0.736	5435001
Scc 010530	int:2:32	Scc 2 / GCC 3	–no-fe-cofold	9175.9	0.788	5435001
Scc 010530	int:2:32	Scc 3 / GCC 3	–no-fe-cofold	9125.7	0.793	5435001
Scc 010530	int:2:32	Scc 0 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	16283.1	0.444	5435001
Scc 010530	int:2:32	Scc 1 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	15685.0	0.461	5435001
Scc 010530	int:2:32	Scc 2 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	11604.4	0.623	5435001
Scc 010530	int:2:32	Scc 3 / GCC 3	–no-fe-cofold –no-be-cofold –no-be-peep	11597.6	0.624	5435001
GCC 2.7.2.1	char:8:32	GCC 0	-	26276.9	0.275[3]	5435001
GCC 2.7.2.1	char:8:32	GCC 1	-	10349.3	0.699[4]	5435001
GCC 2.7.2.1	char:8:32	GCC 2	-	10773.6	0.674	5435001
GCC 2.7.2.1	char:8:32	GCC 3	-	10887.6	0.664	5435001
GCC 2.7.2.1	int:32:32	GCC 0	-	19702.7	0.367	5435001
GCC 2.7.2.1	int:32:32	GCC 1	-	7234.4	1.000[5]	5435001
GCC 2.7.2.1	int:32:32	GCC 2	-	7264.5	0.996	5435001
GCC 2.7.2.1	int:32:32	GCC 3	-	7757.5	0.933	5435001

[1] Worst Scc-compiled
[2] Best dna32, best overall
[3] Worst GCC-compiled
[4] Best C char
[5] Best C int, best GCC-compiled

APPENDIX H
LINPACK PERFORMANCE

The SWARC code used to replace core loops in the C/C++ Linpack 100x100 Benchmark included the following source:

```
/* lp.Sc - Compile with Scc -c -k -mK6-2 -O5 */

void swar_saxpy(float:32[VECTSIZE] x, float:32[VECTSIZE] y, float s)
{
        y += (s * x);
}

void swar_sdot(float:32[VECTSIZE] x, float:32[VECTSIZE] y, float s)
{
        s += (x * y);
}

void swar_sscal(float:32[VECTSIZE] x, float s)
{
        x = x * s;
}
```

Currently, floating-point operations are only supported for 3DNow! and AltiVec. The code should be compiled with the correct target switch to allow the compiler to take advantage of SWAR floating-point instructions.

H.1 Results for 3DNow!

The following two tables report performance results on a 1GHz AMD Athlon-based HP Pavilion N5470 laptop computer. The first of these reports average MFLOPS using rolled standard C code. The second reports average MFLOPS using Scc-generated SWARC code.

Table H.1
Results for rolled C code

VECTSIZE	OPTIME	Average 201	Average 200
2	1	267.99	249.22
2	2	268.10	249.23
2	4	268.01	249.22
2	100	267.90 (Worst)	249.89 (Best)
4	1	268.17	248.78 (Worst)
4	2	268.17	248.79
4	4	268.28 (Best)	248.99
4	100	268.10	249.39
8	1	268.19	249.29
8	2	268.08	249.10
8	4	268.08	249.15
8	100	267.98	249.87
16	1	268.01	249.56
16	2	267.93	249.63
16	4	268.01	249.69
16	100	267.93	249.63
32	1	268.10	249.33
32	2	268.01	249.33
32	4	268.01	249.48
32	100	268.10	249.33

In both cases, the source was sent through the Scc compiler, which generated C code from the SWARC code. This was passed by Scc to the native C compiler (GCC 2.96), which generated the executable. The Scc-generated code was called by the executable conditionally depending on the definition of a macro.

In the first table, table H.1, VECTSIZE and OPTIME are irrelevant because VECTSIZE is only used within the SWARC code and OPTIME was the time that Scc was allowed to spend generating a schedule for this SWARC code. The SWARC code was not called by the executables in this set of runs.

In the second table, table H.2, VECTSIZE represents the fixed vector length used for generating code. Currently, the Scc compiler does not allow for variable vector lengths. The length of each vector must be declared or the compiler will assign it a length of one element. OPTIME was the time allowed for the Scc compiler to attempt to find the best schedule for each basic block.

For the rolled C code, the best run with a dimension of 201 achieved 268.28 MFLOPS, while the best run for a dimension of 200 achieved 249.89 MFLOPS. In each case, the variance was negligible.

Table H.2
Results for SWARC code

VECTSIZE	OPTIME	Average 201	Average 200
2	1	408.28	402.27
2	2	408.09	402.50
2	4	407.52 (Worst)	401.57 (Worst)
2	100	407.90	402.82
4	1	464.86	487.24
4	2	463.76	487.00
4	4	464.53	486.68
4	100	464.53	487.14
8	1	540.28	586.58
8	2	540.91	586.96
8	4	540.91	587.27
8	100	540.75	587.04
16	1	550.75	616.65 (Best)
16	2	551.28 (Best)	616.30
16	4	551.01	616.64
16	100	551.17	616.34
32	1	521.63	559.43
32	2	521.41	557.53
32	4	521.91	558.95
32	100	520.97	558.54

In comparison, the best run for SWARC code with a dimension of 201 achieved 551.28 MFLOPS with a VECTSIZE of 16 and a 2 second maximum optimization time. This is an improvement of $\frac{551.28-268.28}{268.28} = 105\%$ over the best rolled C code.

The best run for SWARC code with a dimension of 200 achieved 616.65 MFLOPS with a VECTSIZE of 16 and a 1 maximum second optimization time. This is an improvement of $\frac{616.65-249.89}{249.89} = 147\%$ over the best rolled C code.

The worst run for SWARC code with a dimension of 201 achieved 407.52 MFLOPS with a VECTSIZE of 2 and a 4 second maximum optimization time. This is an improvement of $\frac{407.52-268.28}{268.28} = 51.9\%$ over the best rolled C code.

The worst run for SWARC code with a dimension of 200 achieved 401.57 MFLOPS with a VECTSIZE of 2 and a 4 second maximum optimization time. This is an improvement of $\frac{401.57-249.89}{249.89} = 60.7\%$ over the best rolled C code.

VECTSIZE was limited to 32 because longer VECTSIZEs led to basic blocks which required more tuples than the current compiler could handle. Notice that the best VECTSIZE for the SWARC version was an intermediate value (8 or 16 elements per subvector).

Table H.3
Results for rolled C code

VECTSIZE	OPTIME	Average 201	Average 200
2	1	175.37	177.45 (Worst)
2	2	175.67	177.45 (Worst)
2	4	175.37	177.54
2	100	175.37	177.54
4	1	175.53	181.33 (Best)
4	2	175.53	180.75
4	4	174.75 (Worst)	180.62
4	100	175.53	180.75
8	1	175.53	180.75
8	2	175.99 (Best)	180.71
8	4	175.45	180.60
8	100	175.45	180.60
16	1	175.45	180.89
16	2	175.45	180.60
16	4	175.39	180.62
16	100	175.45	180.60
32	1	175.45	180.60
32	2	175.99	180.71
32	4	175.53	180.66
32	100	175.53	180.75
64	1	175.31	177.45
64	2	175.45	177.45
64	4	175.37	177.45
64	100	175.61	177.40

H.2 Results for AltiVec

The following two tables report performance results on a 500MHz PowerPC G4-based Apple PowerBook laptop computer. Again, the first of these, table H.3, reports average MFLOPS using rolled standard C code. The second, table H.4, reports average MFLOPS using Scc-generated SWARC code.

The same compilation process was used as for the 3DNow! trials, with the same version of the Scc compiler being used. Version 2.95.3 of GCC was used to compile the final C code for the PowerPC target.

Again, VECTSIZE and OPTIME are irrelevant in the first table and have the same meaning in the second as in the previous section.

For the rolled C code, the best run with a dimension of 201 achieved 175.99 MFLOPS, while the best run for a dimension of 200 achieved 181.33 MFLOPS. In each case, the variance was relative small.

Table H.4
Results for SWARC code

VECTSIZE	OPTIME	Average 201	Average 200
2	1	49.48	49.73
2	2	49.46 (Worst)	49.69 (Worst)
2	4	49.60	49.73
2	100	49.48	49.73
4	1	93.34	94.02
4	2	93.34	94.14
4	4	93.36	94.02
4	100	93.50	94.02
8	1	126.34	127.41
8	2	126.34	127.41
8	4	126.68	127.41
8	100	126.29	127.41
16	1	150.69	152.29
16	2	150.75	152.27
16	4	150.69	152.29
16	100	150.69	152.29
32	1	160.03	167.59 (Best)
32	2	160.40 (Best)	167.17
32	4	160.36	167.15
32	100	160.32	167.20
64	1	96.27	97.10
64	2	96.31	96.93
64	4	96.52	96.91
64	100	96.31	96.91

In comparison, the best run for SWARC code with a dimension of 201 achieved 160.40 MFLOPS with a VECTSIZE of 32 and a 2 second maximum optimization time. This is a degradation of $\frac{175.99-160.40}{175.99} = 8.9\%$ versus the best rolled C code. This means that the best Scc-generated code had significantly slower performance than the corresponding GCC-generated code.

The best run for SWARC code with a dimension of 200 achieved 167.59 MFLOPS with a VECTSIZE of 32 and a 1 second maximum optimization time. This is a degradation of $\frac{181.33-167.59}{181.33} = 7.6\%$ versus the best rolled C code. Again, this means that the Scc-generated code was significantly slower than the corresponding C code.

VITA

Randall James Fisher was born in Niles, Michigan on February of 1966. He received a B.S. in Electrical Engineering in 1989 from Michigan State University, where he studied computer engineering under the Computer Engineering Option.

He entered the Master's program there in the same year to study computers and control systems, graduating with an M.S. in Electrical Engineering in 1991 during the peak of the "downsizing" trend. Thus, he was able to attend several job interviews with companies that weren't planning to hire anyone anyway.

After tiring of this, he took a year off to be with his family before applying to several Doctoral programs. Purdue being the only school wise (foolish?) enough to accept him, he moved to West Lafayette in August of 1993 to study compilers and parallel processing.

As a member of the PAPERS research group at Purdue University, he has helped to develop the aggregate function model of parallel processing and techniques for using and maintaining clusters of workstations including computational clusters and video walls. His own research has focused on developing languages and compilers for parallel processing, especially for the general-purpose SWAR (SIMD Within A Register) processing model which he co-developed with Hank Dietz.

While at Purdue, he was a teaching assistant in the undergraduate compilers course and operating systems laboratory. For these, he wrote supplemental materials and developed online examples, several of which are still in use.

He is currently an instructor of Electrical and Computer Engineering at Pennsylvania State University, the Behrend College in Erie, Pennsylvania, where he has played a significant role in the development of its Computer Engineering program.