General-Purpose SIMD Within A Register: Parallel Processing On Consumer Microprocessors

Ph.D. Thesis Proposal

10:00AM, Wednesday, November 12, 1997
Room MSEE239

Randall Fisher
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907-1285

General-Purpose SIMD Within A Register: Parallel Processing On Consumer Microprocessors

Randall Fisher

rfisher@ecn.purdue.edu

School of Electrical and Computer Engineering

Purdue University

West Lafayette, IN 47907-1285

November 12, 1997

ABSTRACT

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

Multimedia algorithms often require identical operations to be performed on a set of graphical or audio data items as a single logical step. Multimedia extensions to microprocessor instruction sets have been developed to allow some basic operations to be performed simultaneously on multiple items in such a set. This is done by supporting SIMD (Single Instruction stream, Multiple Data stream) parallel processing across multiple data *fields* (sections) within specially enhanced processor registers. By providing this support, these extensions attempt to capture some of the potential speed-up due to the parallel nature of these multimedia algorithms.

These new extensions are sufficient for hand-coding parallel operations within algorithms at the assembly level, but do not directly support a single, consistent, high-level parallel programming model. Each of the extension families has been targeted to support the algorithms most often used on its host platform, and is limited in its functionality and support for various data sizes. The instructions and sizes supported often differ substantially between families, making portability difficult. Even within a single family, the data sizes supported differ between instructions, often hindering high-level programming by forcing data conversions between successive instructions.

While these limitations make it difficult to develop a general-purpose model for *SWAR* (SIMD Within A Register) parallel programming, they are not fatal. In fact, a generalized SWAR programming model can be developed which can target standard processor families with no multimedia extensions whatsoever. By designing this model as a consistent core programming interface, extension sets can be viewed as hardware optimizations to the basic target which it provides.

1.1 Related Work

Though we know of no other group pursuing a high-level approach to general-purpose SWAR processing, some microprocessor manufacturers do provide support for using their multimedia extensions via assembly-level programming tools and high-level multimedia libraries.

A few programming tools have been developed to use Intel's MMX instructions [10]. Microsoft's Visual C++ version 5.0 compiler [16] provides inline assembly support for, and the ability to disassemble, MMX instructions. NuMega Technologies's SoftICE for Windows 95 and SoftICE for Windows NT [17] are debuggers which allow the disassembly of MMX instructions. Also, Powersoft

(Sybase) Corporation's Watcom C/C++ [18] provides support for inline assembly, disassembly, and debugging of MMX instructions. These tools will give programmers assembly-level access to the MMX instructions, but do not claim to provide a higher-level programming model.

Intel has also announced the "C/C++ Compiler Plug-In" [11] for the Microsoft Visual C++ Integrated Development Environment. One of the listed features of the plug-in is that it "[s]upports MMX technology through the use of C intrinsics." Thus, programmers can use the available MMX instructions via a C function call syntax. This provides a higher-level interface than assembly, but does not alleviate the limitations of the MMX instruction set.

Several application-specific libraries for MMX have also been developed, including signal processing, graphics, imaging, speech, and video libraries, primarily from Intel [12] [13] [14] and IBM [19]. However, these libraries are not intended to provide a general-purpose programming model, and support only specific data sizes.

The VTune optimization package from Intel [15] provides programmers with performance tuning tools and advice for using MMX instructions. However, it also does not appear to provide a high-level SWAR programming model.

Silicon Graphics MIPS Group claims that several vendors provide "...software development tools that support MIPS architecture with a comprehensive set of tools and compilers...." [26] However, it is not clear that any of these groups are developing tools to target MIPS MDMX [25] extensions or develop a SWAR programming model.

Programmers of Sun's VIS extensions are expected to use assembly programming, macros in the "VIS Software Developer's Kit (VSDK)" [27], or a C library called "mediaLib" [30]. Both VSDK and mediaLib can be freely downloaded in binary form for certain platforms after a required licensing and non-disclosure agreements [32] [29] are electronically agreed to. The documentation for VSDK is part of the licensed package, but documentation for mediaLib is freely downloadable [30], 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.

MicroUnity's mediaprocessor [24] incorporates parallel processing techniques to enhance its ability to communicate with other devices to deliver information to the user. The mediaprocessor makes use of "sub-instruction level parallelism" over "all multiprecision data types", and provides support for VLIW and superscalar programming models, as well as for "thread-level parallelism". However, the mediaprocessor's primary function is communication, rather than computational processing, and the company is not pursuing general-purpose parallel processing.

1.2 The SIMD Processing Model

During the late 1960s, the ILLIAC IV [2] [21] was introduced to run applications requiring highly regular computation such as physical system modeling where the value of some variable at a given point is a function of its value at the neighboring points. To attack these problems, ILLIAC IV had a single "control unit" (CU) and 64 identical "processing units", each of which had a "processing element" (PE) and a local "processor memory" (PM). Each PE would typically represent a single point in the physical space, upon which identical computations were performed.

During operation, the CU would access an "instruction memory" and generate control signals for the entire array of processing units. Driven by these signals, each PE would execute the instruction on its own local data. Thus, a single stream of instructions was read by the CU and controlled multiple streams of data from the PMs. This processing model was later termed SIMD by Flynn [7].

The SIMD model produces speed-up by taking advantage of "data parallelism" in the executing code. *Data parallelism* refers to the ability of multiple data items to be processed simultaneously without changing the results of the code execution from those which would be generated by processing the same data sequentially. By applying the instruction simultaneously to as many data-parallel items as possible, SIMD execution can theoretically produce speed-up equivalent to the number of such items.

The actual speed-up that a SIMD machine can achieve is limited by the number of PEs which it contains. This is known as the parallelism "width" of the machine. Each PE can only operate on one data item during any given time slot, thus, the "wider" the machine, the more data parallelism it can take advantage of, and the more speed-up it can produce.

1.3 The SWAR Processing Model

The SIMD processing model can be applied to a processor register by logically partitioning it into multiple data fields. Each field represents a register on one PE of a standard SIMD machine, whose width equals the number of fields in the register. When a SIMD instruction is fetched, it is applied to each of the fields simultaneously but independently, and interactions which might normally occur between the fields are blocked. We refer to this form of the SIMD model as the SWAR (SIMD Within A Register) processing model.

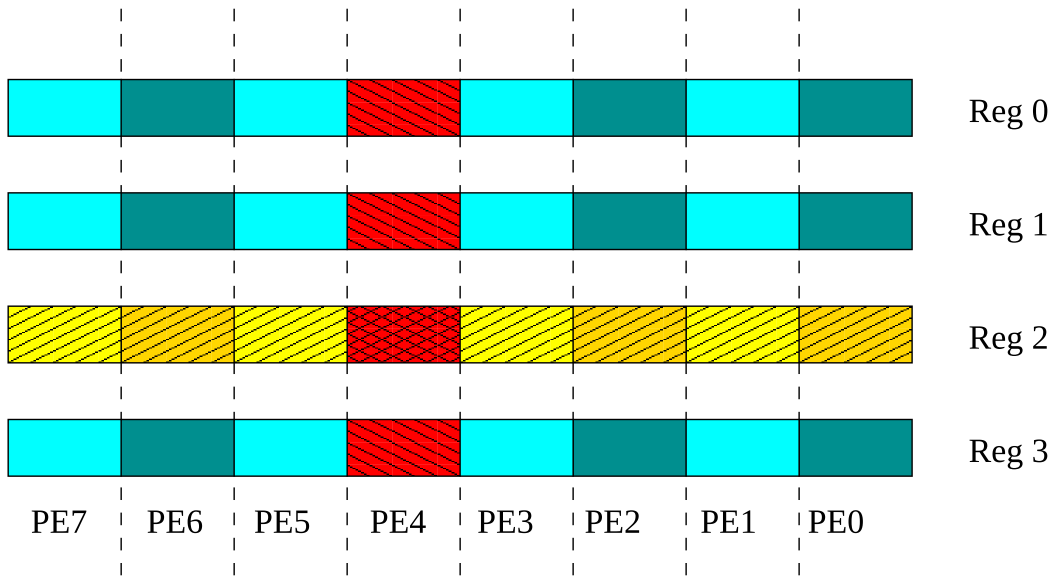

Figure 1: Four Registers Partitioned Into Eight 8-bit Fields (PEs)

Figure 1 shows how the SWAR model might be applied to a set of four 64-bit processor registers. The hardware registers, which are assumed to be named Reg 0 through Reg 3 on the host architecture, run horizontally across the figure. Reg 2 is shown in yellow with hatching falling from right to left. These 64-bit registers are partitioned into eight data fields of 8 bits each, which are assigned PE numbers. By convention, the register bits and fields are numbered from right to left using a 0-based scheme. Thus, the PEs are assigned numbers 0 through 7 from right to left.

Each “PE” can then be considered to have a set of four 8-bit registers named Reg 0 through Reg 3 upon which SWAR operations can be performed. The set of fields belonging to a given PE run vertically. The registers of PE4 are shown in orange and are hatched falling from left to right. Thus, if we were to refer to register Reg 2 on PE4, we would be referring to the orange cross-hatched field.

This same figure could also represent a set of 32-bit registers with 4-bit fields, 16-bit registers with 2-bit fields, or any size register that can be partitioned into eight fields. We could also think of Reg 1 and Reg 3 as continuations of Reg 0 and Reg 2, respectively. In this case, we would consider the machine to have sixteen PEs with two 8-bit registers each.

Note that in partitioning the hardware register into multiple fields, the SWAR model has limited the number of bits which can be stored per datum. This per-register magnitude/width trade-off is characteristic of the SWAR model when applied to an architecture in which the number of bits per register is fixed.

1.4 SWAR Operations

Certain operations must exist in order for the SWAR model to be useful. Data must be moved in and out of the parallel field representation, and between the partitioned register and either the memory system or other processor registers. It may also need to be restructured within the register so that the register can be repartitioned to change field sizes. Operations which actually perform parallel computation such as addition must be available. PEs must be able to share data and pass intermediate results to one another. Also, partial result values must be collected and combined to form final results or intermediate results which may need to be distributed back to the PEs.

A summary of SWAR operations was presented by Hank Dietz early in 1997 [5] [4]. The remainder of this section is a recapitulation of this information for the purposes of this proposal. In his summary, Dietz classifies SWAR operations into four primary areas by their inter-field effects. These areas are the *polymorphic*, *partitioned*, *communication & type conversion*, and *recurrence* operations.

1.4.1 Polymorphic Operations

In general, SWAR operations should be *non-interfering* in the sense that the application of the operation to any given field never produces an effect which would interfere with the application of the operation in another field.

A *polymorphic* operation is one which can be correctly implemented by applying its "full register" (i.e. width=1) form across the entire register, regardless of the actual partitioning, without causing inter-field interference. Examples of polymorphic operations are the bitwise-AND and one's-complement operations.

These operations are inherently non-interfering because each result bit is a function only of the bits of the operands which are in the same position as the result bit. Thus, no matter how the register is partitioned, no field result is a function of the operation in other fields.

1.4.2 Partitioned Operations

A *partitioned* operation is one in which an operation in one field can affect the operation in another field unless measures are taken to prevent this interference. The classic example is addition, in which a carry may occur from the most significant bit position of one field into the least significant bit position of an adjacent field.

There are several approaches to handling this form of interference. It can be prevented using new instructions coupled with hardware support which can modularize the operation by turning off the carry/borrow logic between fields. Operations can be performed as though they were polymorphic, then followed by correction code to undo the effects of any interference. Data values can be limited in magnitude to leave "buffer" bits in each field upon which the otherwise interfering effects of an operation will fall. Also, operations can be designed to saturate overflowing results at the largest possible value which will fit into the field, thus avoiding the interference by defining a new operation.

1.4.3 Communication & Type Conversion Operations

Communication operations are those whose purpose is to move data from one or more fields to one or more other fields. "Broadcasts" and nearest-neighbor data transfers are examples of such operations.

These operations can sometimes be carried out as though they were a standard operation on the entire hardware register. For example, having each PE pass information to its *successor* (the PE with the next number in sequence), is equivalent to performing a left shift on the full register by the number of bits in a field. "Permutations" and broadcasts, however, require special handling in the form of new hardware support or software emulation to deal with more complex communication patterns and data replication.

Type conversion operations are used to rearrange and restructure data in the hardware register during, or as a precursor to, a register repartitioning. These operations expand or contract the data fields of a register, thus affecting the position of these fields and the size of the data that can be stored in each. These operations also affect the parallelism width by changing the number of fields which can fit in the register. Support must be provided for this repartitioning to move fields, to restructure the data stored in each field, and to handle the loss or addition of fields.

1.4.4 Recurrence Operations

Recurrence operations are those which are traditionally solved by sequentially applying the operation to an initial pair of values, then repeatedly to the previous result and a new data value until all data values have been used. These operations generate either a single result, copied to all the fields, which is a function of all the operands' fields, or a set of results, each of which is a function of a subset of the operands' fields.

"Associative reductions" are a form of recurrence which produce a single result value from a set of field values. Because these operations are associative, they can theoretically occur in any order, and can therefore be parallelized. One example of such an operation is the global-Or, in which the bit patterns stored in the fields of the hardware register are bitwise-Ored together to form a single bit pattern. This pattern is copied into each of the fields as the single result of the operation.

"Parallel prefix" operations (a.k.a. "scans") are recurrences which form a set of field results by storing each of their intermediate results into one of the fields. Thus, each field result is a function of the data stored in some subset of the operands' fields.

1.5 Multimedia Extensions to Microprocessor Instruction Sets

Several commercial developers of microprocessors have created extensions to their standard instruction sets to improve performance for the multimedia applications they believe to be most important to their clientele. The philosophy behind the design of these extensions seems to be well described by the Hewlett-Packard Company's own design goal which was "to introduce instructions that provide significant performance improvement with insignificant impact on the area, cycle-time, and design time of the PA-RISC processor." [22]

These extensions are grouped into several *families*, each of which provides a particular set of instructions to perform some of the more frequently used multimedia operations. These extensions are usually carried out in modified processor registers, but one architecture includes a new "accumulator" expressly for use by the resident extension family.

None of these extension families appears to be an attempt to develop a high-level SWAR programming model. Support is usually limited to 8-, 16-, and 32-bit field sizes, and is usually not consistently provided 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 providing a large number of SWAR operations, while others provide only a few. Some support is usually provided 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.

Some of the more visible extension families and the support they provide for arithmetic SWAR operations are described in table 1. The level and variance of the support provided for arithmetic operations is typical of that provided for other types of operations as well. The families listed in these tables are: Digital Equipment Corporation's MAX [6]; Hewlett-Packard Company's PA-RISC MAX v2.0 [22] [23]; Intel Corporation's [20] [9], Advanced Micro Devices, Incorporated's [1], and Cyrix Corporation's [3] versions of MMX; Silicon Graphics MIPS MDMX [25]; and Sun Microsystems, Incorporated's VIS [28] [31].

The left column of table 1 shows different groups of SWAR operations and each of the remaining columns show the support provided by each of the extensions families for that operation. In the left column, "Part" indicates a partitioned operand, "Single" indicates a partitioned operand with identical field values, and "Immed" indicates an immediate operand provided by the instruction itself. "Acc" denotes the use of the SG-MIPS MDMX 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, while "Acc Subt" indicates that the operation will subtract its result from the accumulator.

In the right-hand columns, "8" is shorthand for "8x8", "16" is short for "4x16", with "8u" and "16u" short for "8x8u" and "4x16u", respectively. "4x16" indicates an operand or result partitioned into 4 fields of 16-bit signed integers. A trailing "u" indicates that the field data is unsigned. Where a value is listed by itself or in a comma-separated list of values, it indicates that a

Operation Types	DEC MAX	HP PA-RISC MAX	SG-MIPS MDMX	Intel, AMD, and Cyrix MMX	Sun VIS
Modular Addition					
Part/Part		16,16u		8x8,8x8u,4x16, 4x16u,2x32,2x32u	2x16,4x16,1x32,2x32
Part/Part w/Acc Init			8x8u,4x16u		
Part/Part w/Acc			8x8u,4x16u		
Single/Part w/Acc Init			8x8u,4x16u		
Single/Part w/Acc			8x8u,4x16u		
Immd/Part w/Acc Init			8x8u,4x16u		
Immd/Part w/Acc			8x8u,4x16u		
Saturation Addition					
Part/Part		16,16u	8x8u,4x16u	8x8,8x8u,4x16,4x16u	
Single/Part			8x8u,4x16u		
Immd/Part			8x8u,4x16u		
Modular Subtraction					
Part/Part		16,16u		8x8,8x8u,4x16, 4x16u,2x32,2x32u	2x16,4x16,1x32,2x32
Part/Part w/Acc Init			8x8u,4x16u		
Part/Part w/Acc Diff			8x8u,4x16u		
Single/Part w/Acc Init			8x8u,4x16u		
Single/Part w/Acc Diff			8x8u,4x16u		
Immd/Part w/Acc Init			8x8u,4x16u		
Immd/Part w/Acc Diff			8x8u,4x16u		
Saturation Subtraction					
Part/Part		16,16u	8x8u,4x16u	8x8,8x8u,4x16,4x16u	
Single/Part			8x8u,4x16u		
Immd/Part			8x8u,4x16u		
Specialized Forms					
Sum of Abs. Diffs	8,8u				
Maximum					
Part/Part	8,8u,16,16u		8u,16u		
Single/Part			8u,16u		
Immd/Part			8u,16u		

Table 1: Arithmetic SWAR Operations in Popular Multimedia Extension Families

form of the operation where both the operands and result have the listed partitioning is supported by the extension family. Where an entry contains an arrow, the notation shows the form of the operands in parenthesis separated by an operator and followed by the arrow and the form of the result. "high" indicates that the stored value is the high half of the actual result, with "low" being symmetrically defined. A "?" indicates that the form of the result is not described, and that the form given in the table is my guess. "[X]" indicates that the fields of the partitioned operand are multiplied internally: PE0xPE1 and PE2xPE3.

Operation Types	DEC MAX	HP PA-RISC MAX	SG-MIPS MDMX	Intel, AMD, and Cyrix MMX	Sun VIS
Minimum					
Part/Part	8,8u,16,16u		8u,16u		
Single/Part			8u,16u		
Immd/Part			8u,16u		
Modular Multiplication					
Part/Part				(4x16) X (4x16)	(4x16) X (4x8)→(4x16)?
				→(4x16 high)	(4x16) X (4x8)→(4x16)?
				(4x16) X (4x16)	(2x16) X (2x16)→(2x32)?
				→(4x16 low)	(2x16) X (2x16)→(2x32)?
Part/Part w/Acc Init			8x8u,4x16u		
Part/Part w/Acc			8x8u,4x16u		
Single/Part w/Acc Init			8x8u,4x16u		
Single/Part w/Acc			8x8u,4x16u		
Immd/Part w/Acc Init			8x8u,4x16u		
Immd/Part w/Acc			8x8u,4x16u		
Part/Part w/Acc Init			8x8u,4x16u		
Part/Part w/Acc Subt			8x8u,4x16u		
Single/Part w/Acc Init			8x8u,4x16u		
Single/Part w/Acc Subt			8x8u,4x16u		
Immd/Part w/Acc Init			8x8u,4x16u		
Immd/Part w/Acc Subt			8x8u,4x16u		
Sat. Multiplication					
Part/Part			8x8u,4x16u		
Single/Part			8x8u,4x16u		
Immd/Part			8x8u,4x16u		
Mult. by Sign (-,0,+)					
Part/Part			16u		
Single/Part			16u		
Immd/Part			16u		
Specialized Forms					
Multiply/Add				2-(4x16)→[X]→	
				2-(2x32)→(2x32)	
Saturation Division					
Part/Part					
Specialized Forms					
Average (round to odd)		16,16u			
Shift Left Logical					
Part/Part			8x8u,4x16u		
Single/Part			8x8u,4x16u		
Immd/Part		16,16u	8x8u,4x16u	4x16,2x32,1x64	
Shift Right Logical					
Part/Part			8x8u,4x16u		
Single/Part			8x8u,4x16u		
Immd/Part		16,16u	8x8u,4x16u	4x16,2x32,1x64	
Shift Right Arithmetic					
Part/Part			4x16u		
Single/Part			4x16u		
Immd/Part		16,16u	4x16u	4x16,2x32,1x64	
Shift Left and Add					
(1,2,3, or 4-bit Immd)		16,16u			
Shift Right and Add					
(1,2,3, or 4-bit Immd)		16,16u			
Scale (various forms)			8u,16u		

Table 1 cont'd. Arithmetic SWAR Operations in Popular Multimedia Extension Families

2 Proposed Work

In this chapter, we propose to define a general-purpose SWAR programming model, libraries, and compiler technology to widen the range of applications able to achieve speed-up through SIMD execution using COTS (Consumer Off-The-Shelf) microprocessors.

2.1 Performing SIMD Parallel Processing Within a Register

SIMD processing is best suited to highly regular applications where the same operation or algorithm is applied repeatedly or to numerous items of data. Consumer applications have not always possessed high degrees of regularity, but with the advent of multimedia applications, this situation is changing. The recently developed extensions to consumer microprocessor instruction sets attempt to capitalize on the data parallel nature of audio and video algorithms by performing SIMD processing within processor registers.

SWAR processing is an application of the basic SIMD model to the registers of a modern commercial COTS microprocessor. In this paradigm, a processor register is either treated as, or physically partitioned into, a set of data fields which each hold a single data item. These fields, and their associated processing logic, act as the processing units of a basic SIMD machine as described in section 1.2. The microprocessor reads an instruction as in its normal sequential processing mode, and controls the processing logic associated with the register and its constituent fields. Thus, each field is treated as a processing unit in a standard SIMD machine, with the microprocessor's execution logic acting as the control unit.

The major property that is required for a processor to be *SWAR-capable* is that a single SWAR instruction must operate on the data in each field of a partitioned register, as a single logical operation, possibly consisting of multiple processor instructions, without disrupting the operation in other fields. These data fields must be large enough to hold an item of interest, but are generally not required to be any larger. It is also possible that some bits will not be used once the register is partitioned into fields of equal sizes.

As an example, in figure 2a, we see a 64-bit register partitioned into four 16-bit fields. In this particular partitioning, all bits are used. In figure 2b, we see one way in which the same register can be partitioned as a set of six 10-bit fields with four bits left unused and shown in black.

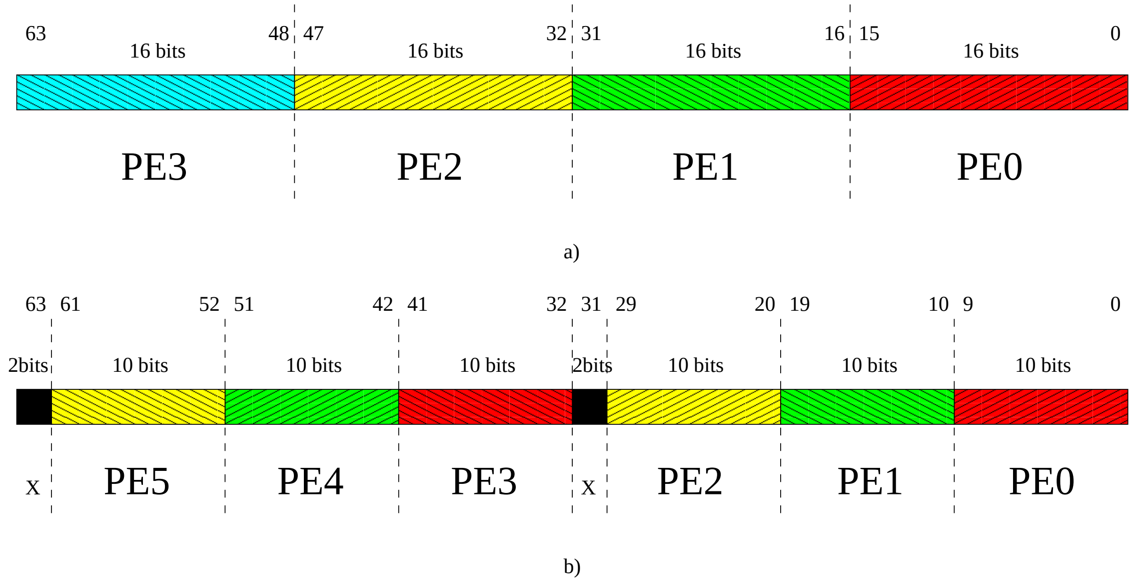

Figure 2: (a) A 64-bit register divided into four 16-bit fields, and (b) a 64-bit register divided into six 10-bit fields with unused bits in black.

To illustrate the benefit of SWAR execution, we can compare the basic code to implement a set of twelve 10-bit additions using each of these partitionings.

In figure 3a, we see the pseudocode for adding the value 3 to each of twelve data items under standard sequential execution. In figure 3b, we see the pseudocode for executing this same task using four 16-bit partitions, and in figures 3c and d, we see pseudocode for execution using six 10-bit partitions.

In examples b and c, "Load" and "Store" are 32-bit memory/integer register operations. In example d, they are 64-bit memory/enhanced register operations. In each of the examples, "Rx" refers to integer register "x" and "ERy" refers to enhanced (partitionable) register "y". "V(a)" is the value stored at the memory location of variable "a" or the memory location itself in a "Store" operation. Brackets indicate the set of values or locations they enclose. "Pack" and "Unpack" move sets of integers from the integer registers into partitioned form within enhanced registers, and vice versa, respectively. "Padd" is the partitioned add operation for the current register partitioning. The first operand is the destination of the operation, with one or two source operands following.

Ideally, all the fields of a partitioned register could be loaded or stored in a single operation. In example d, we see the pseudocode for execution using six 10-bit partitions with the assumption that the data is prearranged in memory and direct 64-bit memory/enhanced register "Load" and

```
Load    R0  0x0003
Load    R1, V(a)
Add     R1, R1, R0
Store   V(a), R1
Load    R1, V(b)
Add     R1, R1, R0
Store   V(b), R1
Load    R1, V(c)
Add     R1, R1, R0
Store   V(c), R1
Load    R1, V(d)
Add     R1, R1, R0
Store   V(d), R1
Load    R1, V(e)
Add     R1, R1, R0
Store   V(e), R1
Load    R1, V(f)
Add     R1, R1, R0
Store   V(f), R1
Load    R1, V(g)
Add     R1, R1, R0
Store   V(g), R1
Load    R1, V(h)
Add     R1, R1, R0
Store   V(h), R1
Load    R1, V(i)
Add     R1, R1, R0
Store   V(i), R1
Load    R1, V(j)
Add     R1, R1, R0
Store   V(j), R1
Load    R1, V(k)
Add     R1, R1, R0
Store   V(k), R1
Load    R1, V(l)
Add     R1, R1, R0
Store   V(l), R1
```

a)

```
Load    R0, 0x00030003
Load    R1, 0x00030003
Pack    ER0, R0, R1
Load    R0, V(a)
Load    R1, V(b)
Load    R2, V(c)
Load    R3, V(d)
Pack    ER1, [R1,R0], [R3,R2]
Load    R0, V(e)
Load    R1, V(f)
Load    R2, V(g)
Load    R3, V(h)
Pack    ER2, [R1,R0], [R3,R2]
Load    R0, V(i)
Load    R1, V(j)
Load    R2, V(k)
Load    R3, V(l)
Pack    ER3, [R1,R0], [R3,R2]
Padd    ER1, ER1, ER0
Padd    ER2, ER2, ER0
Padd    ER3, ER3, ER0
Unpack [R1, R0], [R3,R2], ER3
Store   V(i), R0
Store   V(j), R1
Store   V(k), R2
Store   V(l), R3
Unpack [R1, R0], [R3,R2], ER2
Store   V(e), R0
Store   V(f), R1
Store   V(g), R2
Store   V(h), R3
Unpack [R1, R0], [R3,R2], ER1
Store   V(a), R0
Store   V(b), R1
Store   V(c), R2
Store   V(d), R3
```

b)

```
Load    R0, 0x00300c03
Load    R1, 0x00300c03
Pack    ER0, R0, R1
Load    R0, V(a)
Load    R1, V(b)
Load    R2, V(c)
Load    R3, V(d)
Load    R4, V(e)
Load    R5, V(f)
Pack    ER1, [R2,R1,R0],[R5,R4,R3]
Load    R0, V(g)
Load    R1, V(h)
Load    R2, V(i)
Load    R3, V(j)
Load    R4, V(k)
Load    R5, V(l)
Pack    ER2, [R2,R1,R0],[R5,R4,R3]
Padd    ER1, ER1, ER0
Padd    ER2, ER2, ER0
Unpack [R2,R1,R0],[R5,R4,R3], ER2
Store   V(g), R0
Store   V(h), R1
Store   V(i), R2
Store   V(j), R3
Store   V(k), R4
Store   V(l), R5
Unpack [R2,R1,R0],[R5,R4,R3], ER1
Store   V(a), R0
Store   V(b), R1
Store   V(c), R2
Store   V(d), R3
Store   V(e), R4
Store   V(f), R5
```

c)

```
Load    ER0, 0x000c0
Load    ER1, V(a)
Load    ER2, V(g)
Padd    ER1, ER1, ER
Padd    ER2, ER2, ER
Store   V(g), ER2
Store   V(a), ER1
```

d)

Figure 3: (a) Pseudo-code for twelve sequential 10-bit adds, (b) for twelve 10-bit SWAR adds in 16 bit fields, (c,d) for twelve 10-bit SWAR adds in 10 bit fields.

“Store” operations are available.

The number of actual arithmetic operations performed drops from 12 for sequential execution, to 3 using 16-bit fields, and to 2 using 10-bit fields. If there are no more cycles required to fetch and execute a SWAR operation than for the equivalent integer operation, both the number of execution and fetch cycles required for the addition instructions should be reduced by using SWAR-based execution.

However, the number of memory accesses stays roughly the same unless the memory system of the target allows some or all of the data to be accessed simultaneously. In this case, the number of clock cycles spent loading and storing the data could also be reduced, thus possibly raising the speedup above linear relative to the number of additions. The enhanced architectures provide functionality to directly transfer a partitioned word between the enhanced registers and memory via standard word-sized load and store instructions assuming that the data is stored in a partitioned format in contiguous memory. The impact of this functionality is evidenced by comparing the code in examples c and d.

Thus, applications which posses regularity and have required data sizes small enough to allow multiple fields in a single register can theoretically achieve speedups equal to the parallelism width of the partitioned register. It is for this reason that SIMD has become interesting to multimedia developers, and has spurred the development of SWAR-capable architectures. The level of support for SWAR execution should continue to grow as software developers take advantage of extended architectures and push for higher performance. Correspondingly, the gap between real and theoretical gains due to SWAR processing should continue to close in the future.

2.2 General-Purpose SWAR Programming

While multimedia applications have spurred the current interest in SIMD, there is no reason that SWAR processing should be limited to the multimedia field. Other applications, such as those targeted by the ILLIAC IV, still exist, and have kept the SIMD model alive for three decades. These, and new applications from fields such as genetic research, should not only be able to benefit from SWAR processing, but be able to use it to exploit as much data-parallelism as possible. To this end, a general-purpose programming model for SWAR processing should be developed.

The set of applications and algorithms which can benefit from SWAR processing is not entirely known at this time. However, a programming model designed in the present may survive into an unknown future in which some of these applications will arise or become dominant. Thus, it is only prudent that a general-purpose programming model avoid creating artificial barriers to these applications. Such an approach should allow current and future applications to benefit from SWAR processing, and also permit the continued broadening of the applications base.

Multimedia extensions can be viewed either as the complete set of permitted SWAR operations or as a limit on the functionality directly supported by the host architecture. We will take the second view, and realize that these extensions place artificial barriers on the structure of the data that can be used and the functionality provided. Thus, the design of a generalized model for SWAR programming should avoid codifying these unique architectural features as an integral part of the model in order to avoid the limitations they impose.

By making the SWAR model independent of these unique features, it can continue to exist as a basis for SIMD applications programming long after the extinction of any particular extension set. Applications written to such a model would have the ability to ride the processor technology curve as new extensions are developed without requiring changes in the application code. Furthermore, the possibility would exist for implementing the model on processors with only standard integer registers.

Because the high-level algorithms needed by future applications are not entirely known, the general-purpose model should be designed as a set of basic building blocks upon which higher-level algorithms can be built. To achieve this, the model should implement basic arithmetic, logical, conditional, and pattern-manipulation operations, operations for the management of data layout and collection, and processing control operations.

2.3 Multimedia Extensions as Opportunities for Optimization

In the previous section, the view was taken that multimedia extensions would overly confine any programming model which was designed to them. Taking the opposite perspective, we see that they can be viewed as opportunities to optimize an implementation of a broader, general-purpose SWAR programming model for their host architectures.

For example, in figure 4 we see that a four-field, k-bit partitioned addition can be implemented on an unenhanced processor in standard integer registers using one (4 x k)-bit ADD, four (4 x k)-bit bitwise-ANDs, two (4 x k)-bit bitwise-exclusive-ORs, and a one's-compliment of the bit mask, for a total of eight instructions. An enhanced processor with a k-bit field partitioned ADD instruction can optimize this set of sequential instructions to a single partitioned SWAR instruction. However, it is not necessary to limit the general-purpose model to a special value of k because it happens to be directly supported by some family of extensions.

It should be noted that mixing hardware and software implementations of SWAR operations will often imply overhead due to processor mode changes, data conversion, and the movement of data between standard and enhanced registers. This overhead may tend to nullify the gains made

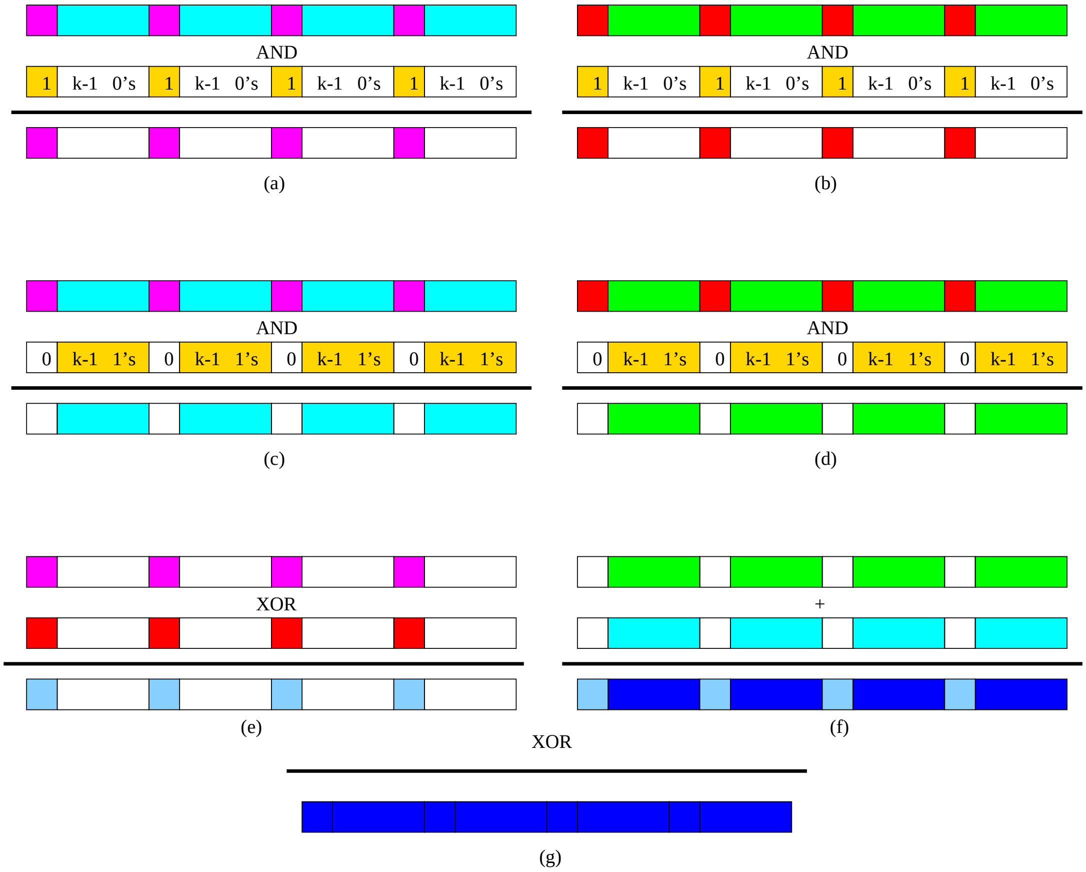

Figure 4: A four-field, k-bit partitioned addition. (a) Isolating the MSbs of operand A and (b) of operand B. (c) Isolating the lower bits of operand A and (d) of operand B. (e) Forming the partial sum of the MSbs, and (f) of the lower bits. (g) The final sum formed by XORing the partial sums together.

through parallelism if these switches are made too frequently. However, this situation may not occur in many applications because the data set from which parallelism is to be derived will often be of one precision.

Data which can fit in an unsupported field size may actually be stored in a larger, but supported, field size in order to take advantage of the target architecture's enhanced hardware features. Thus, some amount of potential parallelism may be traded off by an optimizing compiler to lessen the overhead of software emulation.

As an example, consider an application where data which fits into 2-bit fields are to be added. In figure 4 it was shown that the software implementation of a parallel add requires approximately eight operations per k adds. Thus, the operation provides a theoretical speedup of 32 when executed on a 64-bit register machine, but may only achieve a speedup of 4 due to the overhead required to emulate the operation in software. If the hardware supports an eight field by 8-bit add instruction, storing and operating on the data using 8-bit fields would cut the potential speedup to 8 (the number of 8-bit fields available), but would remove the software emulation overhead, thus allowing this potential speedup to be achieved.

Hardware supported implementations of SWAR operations do not always have a large advantage over software implementations. In particular, polymorphic operations which are supported as standard operations may require only one operation, and may therefore match the speedup gained by using an extended instruction.

It is clear that while software emulation of SWAR operations achieves speedup once the overhead of field separation is met, hardware extensions which provide this separation implicitly will be able to achieve a higher level of speedup. Thus, when available, hardware extensions can be used to optimize SWAR operations; however, the amount of speedup may be diminished due to switches between hardware and software implementations forced by an incomplete set of hardware-enhanced instructions.

2.4 High Level SWAR Languages

A well-defined general-purpose SWAR programming model should provide sufficient functionality, structure, and portability to make the development of new high-level languages (HLLs) which target it be straight-forward. The model should avoid presenting barriers to its integration into existing languages; however, the languages themselves may present barriers which must be overcome by the language implementer.

High-level languages have various purposes. General-purpose HLLs are likely to attempt to make the entire SWAR model available to the programmer, while application-specific languages may limit the scope of the model to those functions and operations which are most likely to be useful in implementing the target application.

A language may incorporate basic SWAR operations by adding reserved symbols to the language, or redefining a symbol's meaning to treat SWARed values differently than other values. For example, APL [8] uses the character combination "+/" to denote a binary reduce-add operation. This combination could be used by a new language, or added to an existing language which does not define this combination, to denote the SWAR reduce-add (or some other) operation.

Data types may also be added or generalized to allow the description of SWARed objects. Thus, SWAR data types can be incorporated into the language definition and applied directly with or without special operators.

Language designers may also choose to present the SWAR model as a set of built-in user-level functions. These may be made available to the programmer in forms which differ from the generalized SWAR model to provide an application-specific interface to the model.

Because the needs of applications programmers vary considerably, the creation of a well-defined generalized SWAR programming model should allow language designers to develop both general-purpose and application specific languages which provide some level of SWAR functionality suitable to the scope of applications targeted by the language.

3 Research Plan

The focus of this research is to develop a general-purpose SWAR programming model and related compiler technology to widen the range of applications able to exploit SIMD parallelism using COTS microprocessors. The following sections outline the research to be performed toward the development of this model, prototype compiler implementations, and optimization techniques.

3.1 Analysis of Current Multimedia Extension Families

The goal of the first phase of research is to determine the capabilities of, and requirements for using, the multimedia extension families which are available on current COTS processors.

Each of these families provides a unique set of enhanced instructions for SWAR computation, data layout, and execution management. An understanding of the range of SWAR features provided by these families is necessary to the development of a good generalized SWAR programming model.

Each extension family is limited in the SWAR functionality that it directly supports. This limits the possibility of code optimization based on its enhanced features. These limits must be considered when deciding how the enhanced features of the target will be used within an implementation of the generalized model.

This analysis will provide a basis for the design and implementations of a general-purpose SWAR programming model to be undertaken in later phases of the research.

3.2 Definition of a General-Purpose SWAR Programming Model

The goal of the second phase of research is to design and develop a new high-level programming model which will allow programmers of general-purpose applications to more readily exploit SIMD-style data-parallelism when targeting SWAR-capable COTS processors.

Such a model must be sufficient for performing basic SIMD parallel programming, and provide a consistent programming interface to the applications programmer. Given these requirements, it should attempt to make use of the most widely available features of the various extension families, and should allow optimization through the use of available architectural enhancements. It should not, however, force marginally useful enhancements to be implemented on targets which do not support them directly.

The model should not artificially restrain the allowed field sizes. Many applications use data which can be stored in a very few bits per datum. By permitting programmers to specify the maximum field size required for their data, the model allows them to trade unnecessary data precision or magnitude to increase the number of fields per register, thereby increasing the amount of parallelism that the register can support.

These issues will be explored more fully to produce a general-purpose SWAR programming model from which a wide range of applications can benefit.

3.3 Proof-Of-Concept Implementations of the Model

In the third research phase, the goal will be to develop prototype implementations of the newly-defined general-purpose SWAR model to verify that it possesses the properties outlined in section 3.2.

A set of libraries will be developed to show that the model can be applied in this manner. This set will provide a portable interface whose implementations can be optimized to the target architectures.

A small parallel programming language will be defined which will be used as the input language for a set of proof-of-concept compilers. These compilers will target the C programming language and the assembly languages of at least two architectures which have been enhanced with multimedia extensions.

Optimizations based on the targets' enhanced capabilities will be developed during this phase to show that the design of the model permits exploitation of these features. Higher level optimizations based on the model itself will also be implemented.

3.4 Properties of a Well-Designed High Level Language for SWAR

The goal of the fourth research phase will be to determine and enumerate a set of properties that a well-designed high level SWAR language should exhibit.

The knowledge gained during the previous research phases will be analyzed to establish guidelines for implementing the general-purpose SWAR model as a full-scale high-level programming language.

3.5 Performance Analysis

The goal of the fifth, and final, research phase will be to develop and employ performance metrics to examine the measurable effects of SWAR-based technology.

Methods to compare the execution of sequential and SWAR-based target code will be developed to determine realistic expectations for speedup based on the use of SWAR technology. These comparisons should help determine the limiting factors which may prevent the theoretical benefits of SWAR execution from being achieved.

4 Significance

If successful, the proposed work will provide a new general-purpose SWAR programming model, compiler technology, and high-level language guidelines to facilitate SIMD parallel processing on commodity microprocessors. The primary contributions of this thesis are expected to be:

- A summary of the support provided by current multimedia extension families for general-purpose parallel processing and possible optimization techniques (section 3.1).
- A new public-domain, high-level programming model to allow general-purpose applications programmers to exploit SIMD parallelism when targeting SWAR-capable COTS processors (section 3.2).
- Prototype libraries and compilers for SWAR-based programming (section 3.3).
- New compiler techniques for optimizing SWAR-based target code (section 3.3).
- A list of properties that a well-designed full-scale high-level language for SWAR should exhibit (section 3.4).
- A small prototype SWAR programming language which exhibits these properties (section 3.4).
- New performance metrics to examine the measurable effects of SWAR-based technology (section 3.5).

References

[1] Advanced Micro Devices, Inc. *AMD-K6 Processor Multimedia Extensions.* Advanced Micro Devices, Inc., Sunnyvale, California, March 1997.

[2] 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*, 17(8):746–757, August 1968.

[3] Cyrix Corporation. Multimedia Instruction Set Extensions for a Sixth-Generation x86 Processor. Technical report, Cyrix Corporation, ftp://ftp.cyrix.com/developr/hc-mmx4.pdf, August 1996.

[4] Hank Dietz. Multimedia Extensions For Microprocessors: SIMD Within A Register. Technical report, School of Electrical and Computer Engineering, Purdue University, http://dynamo.ecn.purdue.edu/˜ hankd/SWAR/970213/slide1.html, February 1997.

[5] Hank Dietz. Technical Summary: SWAR Technology. Technical report, School of Electrical and Computer Engineering, Purdue University, http://dynamo.ecn.purdue.edu/˜ hankd/SWAR/over.html, February 1997.

[6] Digital Equipment Corporation. *Alpha Architecture Handbook, Version 3.* Digital Equipment Corporation, Maynard, Massachusetts, October 1996.

[7] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(8):948–960, 1972.

[8] Leonard Gilman and Allen J. Rose. *APL: An Interactive Approach, Second Edition Revised Reprinting.* John Wiley & Sons, Inc., New York, New York, 1976.

[9] 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.

[10] Intel Corporation. Development & Creation Tools. Technical report, Intel Corporation, http://developer.intel.com/drg/tools/ad.htm, May 1997.

[11] Intel Corporation. Intel C/C++ Compiler version 2.4. Technical report, Intel Corporation, http://developer.intel.com/design/perftool/icl24/, May 1997.

[12] Intel Corporation. Intel Image Processing Library. Technical report, Intel Corporation, http://developer.intel.com/design/perftool/perflibst/ipl/, May 1997.

[13] Intel Corporation. MMX Technology Tools Providers: Intel Corporation. Technical report, Intel Corporation, http://developer.intel.com/design/perftool/perflibst/rpl/, April 1997.

[14] Intel Corporation. MMX Technology Tools Providers: Intel Corporation. Technical report, Intel Corporation, http://developer.intel.com/design/perftool/perflibst/spl/, April 1997.

[15] Intel Corporation. MMX Technology Tools Providers: Intel Corporation. Technical report, Intel Corporation, http://developer.intel.com/design/perftool/vtune/, March 1997.

[16] Intel Corporation. MMX Technology Tools Providers: Microsoft Corporation. Technical report, Intel Corporation, http://developer.intel.com/drg/tools/ms_vc.htm, April 1997.

[17] Intel Corporation. MMX Technology Tools Providers: NuMega Technologies. Technical report, Intel Corporation, http://developer.intel.com/drg/tools/numega.ht, March 1997.

[18] Intel Corporation. MMX Technology Tools Providers: Powersoft (Sybase) Corporation. Technical report, Intel Corporation, http://developer.intel.com/drg/tools/sybase.htm, March 1997.

[19] Intel Corporation. MMX(TM) Technology Tools Providers: IBM Corporation. Technical report, Intel Corporation, http://developer.intel.com/drg/tools/ibm.htm, March 1997.

[20] Intel Corporation. MMX(TM) Techonology Overview. Technical report, Intel Corporation, http://developer.intel.com/drg/mmx/, February 1997.

[21] David J. Kuck. ILLIAC IV Software and Application Programming. *IEEE Transactions on Computers*, 17(8):758–770, August 1968.

[22] Ruby Lee and Jerry Huck. HPTechnical Computing - 64-bit and Multimedia in the PA-RISC 2.0 Architecture. Technical report, Hewlett-Packard Company, http://hpcc997.external.hp.com:80/wsg/strategies/pa2go3/pa2go3.html, June 1996.

[23] Ruby B. Lee. Accelerating Multimedia with Enhanced Microprocessors. *IEEE Micro*, 15(2):22–32, April 1995.

[24] MicroUnity Systems Engineering, Inc. Mediaprc. Technical report, MicroUnity Systems Engineering, Inc., http://www.microunity.com/www/mediaprc.htm, 1996.

[25] Silicon Graphics, Inc. MIPS Digital Media Extension. Technical report, Silicon Graphics, Inc., http://www.sgi.com/MIPS/arch/ISA5/, 1996.

[26] Silicon Graphics, Inc. Software Development Tools for MIPS Processors. Technical report, Silicon Graphics, Inc., http://www.sgi.com/MIPS/products/SfDev_Tools_Tb.html, 1996.

[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. 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/, June 1996.

[29] Sun Microsystems, Inc. mediaLib Object Code License. Technical report, Sun Microsystems Corporation, http://www.sun.com/sparc/vis/download/mlib/, March 1997.

[30] Sun Microsystems, Inc. mediaLib, Sun Microelectronics. Technical report, Sun Microsystems Corporation, http://www.sun.com/sparc/vis/mediaLib.html, March 1997.

[31] Sun Microsystems, Inc. The VIS Instruction Set, Sun Microelectronics. Technical report, Sun Microsystems Corporation, http://www.sun.com/sparc/vis/index.html, April 1997.

[32] Sun Microsystems, Inc. VSDK License. Technical report, Sun Microsystems Corporation, http://www.sun.com/sparc/vis/download/vsdk/, April 1997.

Contents

1 Introduction **2**

1.1 Related Work 2

1.2 The SIMD Processing Model 4

1.3 The SWAR Processing Model 4

1.4 SWAR Operations 6

1.4.1 Polymorphic Operations 6

1.4.2 Partitioned Operations 7

1.4.3 Communication & Type Conversion Operations 7

1.4.4 Recurrence Operations 8

1.5 Multimedia Extensions to Microprocessor Instruction Sets 8

2 Proposed Work **12**

2.1 Performing SIMD Parallel Processing Within a Register 12

2.2 General-Purpose SWAR Programming 15

2.3 Multimedia Extensions as Opportunities for Optimization 16

2.4 High Level SWAR Languages 18

3 Research Plan **20**

3.1 Analysis of Current Multimedia Extension Families 20

3.2 Definition of a General-Purpose SWAR Programming Model 20

3.3 Proof-Of-Concept Implementations of the Model 21

3.4 Properties of a Well-Designed High Level Language for SWAR 21

3.5 Performance Analysis 22

4 Significance **23**