Compiling For SIMD Within A Register

Randall J. Fisher and Henry G. Dietz

School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907-1285
rfisher@ecn.purdue.edu and hankd@ecn.purdue.edu

Abstract

Although SIMD (Single Instruction stream Multiple Data stream) parallel computers have existed for decades, it is only in the past few years that a new version of SIMD has evolved: SIMD Within A Register (SWAR). Unlike other styles of SIMD hardware, SWAR models are tuned to be integrated within conventional microprocessors, using their existing memory reference and instruction handling mechanisms, with the primary goal of improving the speed of specific multimedia operations.

Because the SWAR implementations for various microprocessors vary widely and each is missing instructions for some SWAR operations that are needed to support a more general, portable, high-level SIMD execution model, this paper focuses on how these missing operations can be implemented using either the existing SWAR hardware or even conventional 32-bit integer instructions. In addition, SWAR offers a few new challenges for compiler optimization, and these are briefly introduced.

1. Introduction

The SWAR model takes advantage of the fact that a wide data path within a processor also can be treated as multiple, thinner, SIMD-parallel data paths. Thus, each register is effectively partitioned into *fields* that can be operated on in parallel. Operations like partitioned field-by-field addition are easily implemented with minimal hardware enhancements. For example, to make a 64-bit adder function as eight 8-bit adders, one simply needs to modify the adder's carry logic so that carry from one 8-bit field to the next is suppressed. But what SWAR field sizes and operations should be supported?

Because the introduction of SWAR techniques was motivated by the need to improve multimedia application performance, it is not surprising that the field sizes supported in hardware tend to be those commonly associated with multimedia's "natural" data types. In particular, 8-bit pixel color channel values are used for most forms of video and image processing, so this data size becomes very important. Similarly, 16-bit audio samples are very commonly used, and this field size also can be used for intermediate results when computations on 8-bit values need more than 8-bit precision. Other data sizes are common in other applications yet not directly supported by any of the SWAR hardware extensions in current microprocessors, for example: 1-bit fields hold boolean values, 2-bit fields hold either 0/1/X values for a logic simulator or a base-pair value for operating on a gene database, and 10-bit or 12-bit values are used for a wide range of digitizing devices. A high-level SWAR language can and should allow users to specify the precise number of bits needed for each datum.

The operations implemented in SWAR hardware also are strongly biased in favor of specific multimedia applications and algorithms. Addition, subtraction, and various other integer operations are commonly supported. Because adding two bright pixel values should not result in a dark pixel, there are a variety of *saturation arithmetic* operations supported (e.g., overflow in saturation arithmetic yields the maximum representable value rather than the low bits of a larger value). 32-bit floating-point operations soon also will be supported (e.g., by the recently announced AMD/Cyrix MMFP), although all current SWAR hardware supports only integer field operations.

Unfortunately, at least from the point of view of building high-level language compilers, current SWAR hardware support is limited to just a few field sizes and operations. Even worse, the supported operations and partitioning reflect the designers' expectations of the needs of multimedia programmers and the internal structures within the particular processor architectures, and are therefore inconsistent across architecture families. The following table illustrates this point by showing the partitioning supported by each of five enhanced architecture families for typical arithmetic operations.

In table 1, the columns represent the multimedia extension set families. The extension families included are Digital Equipment Corporation extensions for the Alpha [DEC96]; Hewlett-Packard PA-RISC MAX v2.0 [Lee95][LeH96]; Intel [Int96], Advanced Micro Devices [AMD97], and Cyrix [Cyr96] versions of MMX; Silicon Graphics MIPS MDMX [SGI96]; and Sun SPARC V9 VIS [Sun97].

The first few rows of the table describe the basic characteristics of the multimedia (MM) registers used for SWAR: How many registers are there? How wide is each register? Are these registers the processor's integer registers or are they overlaid on the floating-point register file? The remaining rows sample various parallel arithmetic operations which one or more of the families support. "Part" is used to identify operations using partitioned register operands, with the notation *a*x*b*u specifying that the register holds *a* fields, each containing a *b*-bit value, with the u suffix indicating that the field values are unsigned. "Single" refers to a single, scalar, operand value; "Immed" refers to an immediate value.

The many differences between these SWAR extension implementations are significantly more complex to manage than the variations in the base instruction sets of these processors. In order to execute fairly general SIMD code, there are also a number of instructions that are conspicuously absent — for example, instructions to help implement SIMD-style conditional branches based on `ANY` or `ALL` condition tests. To support a high-level language, these problems must be efficiently managed by the compiler.

	DEC Alpha	HP PA-RISC MAX	SGI MIPS MDMX	Intel, AMD, & Cyrix MMX	Sun Sparc V9 VIS
# MM Registers	32	31	32 (1 Acc.)	8 (+ Mem.)	32
# Bits/ MM register	64	32 or 64	64 (192 Acc.)	64	64
Which registers?	Integer	Integer	Float	Float	Float
Modular Add & Sub					
Part/Part	-	4x16,4x16u	8x8u,4x16	8x8,8x8u,4x16, 4x16u,2x32,2x32u	2x16,4x16, 1x32,2x32
Single/Part	-	-	8x8u,4x16	-	-
Immed/Part	-	-	8x8u,4x16	-	-
Saturation Add & Sub					
Part/Part	-	4x16,4x16u	8x8u,4x16	8x8,8x8u,4x16,4x16u	-
Single/Part	-	-	8x8u,4x16	-	-
Immed/Part	-	-	8x8u,4x16	-	-
Sum of Abs. Diffs	8x8→4x16, 8x8u→4x16u	-	-	-	-
Modular Mul					
Part/Part	-	-	8x8u,4x16	(4x16)x(4x16) →(4x16 high), (4x16)x(4x16) →(4x16 low)	(4x16)x(4x8) →(4x16)
Single/Part	-	-	8x8u,4x16	-	-
Immed/Part	-	-	8x8u,4x16	-	-
Saturation Mul					
Part/Part	-	-	8x8u,4x16	-	-
Single/Part	-	-	8x8u,4x16	-	-
Immed/Part	-	-	8x8u,4x16	-	-
Mul by Sign (-,0,+)					
Part/Part	-	-	4x16	-	-
Single/Part	-	-	4x16	-	-
Immed/Part	-	-	4x16	-	-
Mul/Add	-	-	-	2-(4x16)→2-(2x32)→ (2x32)	-
Average	-	4x16,4x16u	-	-	-
Shift Left/Right					
Part/Part	-	-	8x8u,4x16	-	-
Single/Part	-	-	8x8u,4x16	4x16,2x32,1x64	-
Immed/Part	-	4x16,4x16u	8x8u,4x16	4x16,2x32,1x64	-
Signed Shift Right					
Part/Part	-	-	4x16	-	-
Single/Part	-	-	4x16	4x16,2x32,1x64	-
Immed/Part	-	4x16	4x16	4x16,2x32	-
Shift and Add					
(1,2,3, or 4-bit Immed)	-	4x16,4x16u	-	-	-
Scale (various forms)	-	-	8x8u,4x16	-	-
Maximum/Minimum					
Part/Part	8x8,8x8u, 4x16,4x16u	-	8x8u,4x16	-	-
Single/Part	-	-	8x8u,4x16	-	-
Immed/Part	-	-	8x8u,4x16	-	-

Table 1: Comparison Of Multimedia Instruction Set Extensions

One obvious shortcoming of the SWAR hardware support is that it generally provides operations for just a few field sizes. Fortunately, we are not limited to the set of partitionings directly supported by the hardware. Logically, partitioning is not a property of "partitioned registers," but of an operation applied to the data in one or more registers. In other words, the data path through the function units may be partitioned in various ways, but the registers really are not partitioned; for example, there is nothing to prevent the programmer from treating a register's contents as having 8-bit fields in one operation and 16-bit fields in the next. It is even possible, and sometimes desirable, for operations to treat a register as though it is partitioned into field sizes which are not supported directly by the enhanced hardware. In fact, the partitioning need not even keep all fields within a register the same size, although managing irregular field sizes is too complex to be discussed in detail in this paper.

In summary, the fact that multimedia registers do not store partitioning information offers great flexibility. In a traditional SIMD execution model, data storage is allocated to each processing element, thus fixing both the number of data that can be operated on in parallel and the boundaries between values in different processors. Using SWAR, we can explicitly trade precision for parallelism width, using only as many bits for each operation as is algorithmically required. Further, the penalty for data crossing these imaginary inter-processing-element boundaries is zero for a SWAR system, which gives SWAR interesting new abilities for operations like reductions.

Our approach to making use of SWAR is based on creating a SWAR module language and compilers, so that users can write **portable SIMD programs** that will be compiled into efficient SWAR modules and interface code that allows these modules to be used within ordinary C programs. It is not particularly difficult to build an optimizing SIMD module compiler supporting only those operations directly provided by a particular SWAR hardware implementation, but portability across all flavors of SWAR-capable hardware makes basic coding of constructs an interesting problem. Section 2 discusses the basic coding techniques needed to achieve efficient execution of general SIMD constructs on any type of SWAR hardware support — including 32-bit integer instruction sets that have no explicit support for SWAR. Beyond the basic coding of SWAR constructs, the SWAR execution model makes it appropriate to consider several new types of compiler optimizations, which are briefly discussed in section 3. The work is summarized, and pointers to the support code that we have developed are given, in section 4.

2. Basic SIMD-to-SWAR Compilation

In determining how to code high-level SIMD constructs using SWAR instructions, it is useful to distinguish several different classes of SIMD operations based on how they can be implemented using SWAR:

- A *polymorphic* operation is a SIMD operation for which the same instruction(s) can be used to implement the operation, independent of the field partitioning. For example, bitwise operations like `NOT`, `AND`, `OR`, and `XOR` are polymorphic, as is the classical SIMD conditional test `ANY` (which returns "true" if any bit in any field is a 1).
- A *partitioned* operation is a SIMD operation in which the value computed by each processing element is a function only of data that also resides in that processing element. In a SWAR implementation, a partitioned operation requires corresponding operand fields to be manipulated without interfering with adjacent fields. This is best accomplished using hardware partitioning, and SWAR hardware extensions generally support various partitioned operations, such as field addition and subtraction. However, no current SWAR hardware implements fully general partitioning for all operations, so software techniques to construct appropriate partitioned operations are critical.
- A *communication* operation logically transmits data across processing elements in an arbitrary pattern. For example, the MasPar MPL construct `router[a].b` accesses the value of the variable `b` on processing element `a`. Unfortunately, except for HP PA-RISC MAX, current SWAR extensions do not allow such general communication patterns to be used in re-arranging field values, and even MAX only allows this rearrangement within a singe register (thus not solving the problem when SIMD vector lengths exceed one register). In fact, most SWAR hardware does not explicitly provide any communication operations; instead, they must be constructed using operations that can cross field boundaries, such as unpartitioned shifts.
- A *type conversion* operation converts SIMD data of one type into another. This becomes complicated in SWAR systems because data types that have different sizes yield different parallelism widths (numbers of fields per register). However, current SWAR hardware supports a variety of type conversion operations, and others can be constructed easily using communication and partitioned operations.
- A *reduction* operation recursively combines field values into a single value. In a traditional SIMD architecture, this involves communication between a shrinking set of processing elements working on values slowly growing in precision; this is a much more natural procedure for SWAR, and is thus worth exploring.
- A *masking* operation allows some processing elements to disable themselves. Obviously, for SWAR, there is no way to disable part of a register; however, there are arithmetic techniques that can be used to achieve the same result.
- A *control flow* operation is essentially a branch instruction that is executed by the SIMD control unit. Although SWAR systems do not have a control unit per se, the ordinary processor instruction set can be viewed as providing this functionality. In order to simplify pipeline structure, most SWAR hardware does not directly allow branching based on the contents of a partitioned SWAR register; it is thus necessary to move the SWAR fields into an ordinary register and test it there. These sequences are often awkward, but are unsurprising.

The following sections describe the basic coding of each of the more interesting classes: partitioned, communication, reduction, and masking operations.

2.1. Partitioned Operations

Partitioned operations are the primary focus of most of the hardware support for SWAR. In particular, most of the speedup claims for SWAR are based on partitioned additions of 8-bit fields, etc. However, not all important field sizes are handled by all SWAR hardware, and many partitioned instructions are simply omitted. We describe three different methods for implementing partitioned operations.

2.1.1. Hardware-Partitioned Operations

In an ideal partitioned operation, the operation is applied both concurrently and independently to the entire set of register fields. Thus, where SWAR hardware supports this, the SWAR code looks just like a single pure SIMD instruction. For example, consider adding two 4x8 values, `A` and `B` (each stored in a 32-bit register), using SWAR partitioned unsigned modular addition:

	31	23	15	7	0
A:	128	46	178	255	
B:	49	43	135	7	
A+B:	177	89	57	6	

Numbering the fields from right to left, the addition performed in field 3 is 128+49 and yields the field result 177. Because this value is storable in an 8-bit field, it is not modified for storage. In contrast, the addition performed in field 1 is 178+135 and yields 313. This value requires nine bits for proper storage, thus it is modularized to fit in the eight bits which are available. The result stored for field 1 is then $313\%(2^8)$, or 57. Note that the overflow bit is lost and does not interfere with the operation as performed in field 2.

2.1.2. Partitioned Operations Using Spacer Bits

	31	23	15		7	0
A:	46	?	178	?	255	
B:	43	?	135	?	7	
A&~S	46	0	178	0	255	
B&~S:	43	0	135	0	7	
(A&~S)+(B&~S):	89	1	57	1	6	
		Spacer		*Spacer*		

When an ideal partitioned addition is not supported by the hardware, spacer bits may be used to allow the field operations to be applied concurrently and independently (as shown above). These spacer bits are used as buffer zones between data fields, so that overflow and/or borrow can occur without interfering with adjacent fields. The spacer bit technique may be used not only to implement additional field sizes for SWAR hardware, but also to implement SWAR partitioned operations on architectures providing only conventional 32-bit integer operations.

The example shown is essentially the same one used in section 2.1.1, but use of spacer bits allows only 3x8 values to fit in each register. At the start of the operation, the spacer values are unknown. This is indicated by a question mark (`?`) in each spacer field. To ensure that none of the field additions will overflow into the next field, the spacer bits of the addends are preset, or normalized, to zero before the addition is performed. This is done by `AND`ing the addends with a mask `S`, which has 1 bits only in the spacer positions.

The addition performed in field 2 occurs just as for an ideal partitioned addition, with no overflow and no modification of the stored result. The addition performed in field 1 is 178+135, yielding 313. This value requires nine bits for storage, and is stored with its lower eight bits in field 1, and its ninth bit carried into the spacer between fields 1 and 2. Only the part of the result stored in field 1 is considered to be valid. Thus, the valid result stored is $313\%(2^8)$ or 57. Note that the overflow bit does not interfere with the operation as performed in field 2. Similarly, the addition in field 0 results in a carry to the spacer between fields 0 and 1, and the storage of the result 6.

It is important to note that spacer bits can also be used to implement partitioned subtraction using unpartitioned instructions. In this case, however, it is necessary to ensure that a borrow will not span fields. Thus, to subtract `A-B`, it is necessary to ensure that the spacer bits in `A` each contain at least one 1 bit and each of the corresponding spacer bits in `B` should have a lower numeric value. For the case of a single spacer bit separating each pair of fields, the spacers in `A` should all be 1 and the spacers in `B` should all be 0.

Only one spacer bit is needed between fields for addition or subtraction, but use of multiple spacer bits between the fields may allow multiple partitioned operations to be performed before re-normalizing the spacer bits. This static optimization simply requires tracking the range of possible values of the spacer bits to determine when re-normalization would be required.

2.1.3. Software-Partitioned Operations

Although the spacer-based partitioned operation code is fast, it only allows 3x8 values in each 32-bit register — not the 4x8 that could be used with hardware partitioning. With a few more instructions, the full densely-packed partitioned operation can be implemented.

The trick is simply to consider each field as being one bit smaller than it truly is, replacing the most significant bit of each field by a “virtual” spacer bit. After performing the spacer-partitioned operation on the modified register value, the most significant bits of each field are computed and inserted in the result. Thus, adding two 4x8 values is done as shown on the following page.

At the start of the operation, the two addends are each split into two partitioned registers. `C` and `D` contain the data — with the virtual spacer bit positions cleared in preparation for the partitioned addition. The resulting value for `E` is the correct 4x8 value, except in that the most significant bits of fields in `A` and `B` have not been added in. Fortunately, the bitwise `XOR` operation implements a one-bit add with no carry; thus, we can compute the addition of the most significant bits of each field by `XOR`ing the most significant bits of `A`, `B`, and `E` (with appropriate masking of the other bits).

	31		23		15		7	0
A:		128		46		178		255
B:		49		43		135		7
C=A&~S:	0	0	0	46	0	50	0	127
D=B&~S:	0	49	0	43	0	7	0	7
E=C+D:	0	49	0	89	0	57	1	6
F=A^B:	1	49	0	5	0	53	1	120
G=F&S:	1	0	0	0	0	0	1	0
H=E^G:	1	49	0	89	0	57	0	6
H:		177		89		57		6

Note that the result field values in `H` are the same as those for ideal partitioned addition.

2.2. Inter-Processing-Element Communication

	31	23	15	7 0
A0:	3	2	1	0
A1:	7	6	5	4
B=A0>>8:	0	3	2	1
C=A1<<24:	4	0	0	0
A0'=B|C:	4	3	2	1
A1'=A1>>8:	0	7	6	5

Partitioned registers may allow mesh and toroidal communication to be performed by applying shift and rotate instructions to the register. In the example above, `A1` and `A0` are, respectively, the upper and lower halves of an 8x8 vector layed out across two 4x8 partitioned registers. Although the numbering of fields is somewhat arbitrary, consider a right-neighbor communication as each processing element sending its value one field to the right.

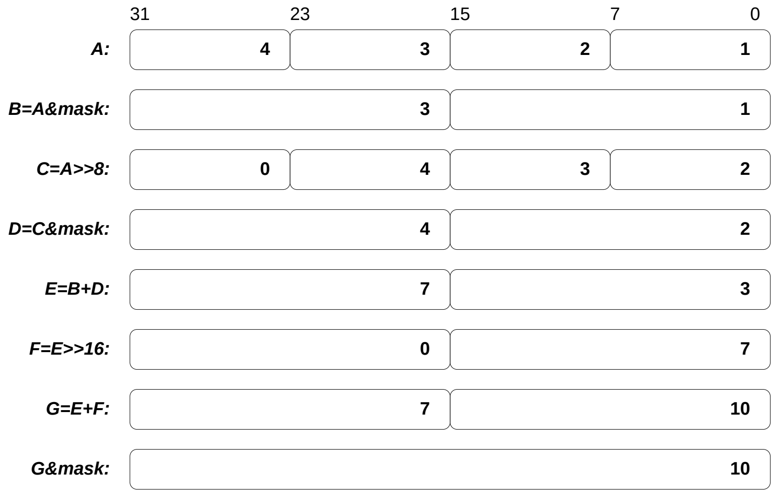
Notice that the new value `A0′` was harder to compute than `A1′` because the datum from processing element 0 logically fell off the right side of the machine. If the right-neighbor communication had included a wrap-around (toroidal) link, the computation of `A1′` would have been much more similar to that of `A0′`.

Unfortunately, more complex communication patterns are not directly supported by most of the SWAR implementations. Implementations of more general communication patterns can be built using the `PACK` and `UNPACK` operations of SWAR extensions like MMX to implement shuffle and inverse-shuffle communication patterns that can be composed to simulate a multistage interconnection network; however, even this is not particularly efficient. Thus, for portability reasons, SWAR programs should avoid the use of complex communication patterns.

2.3. Reductions

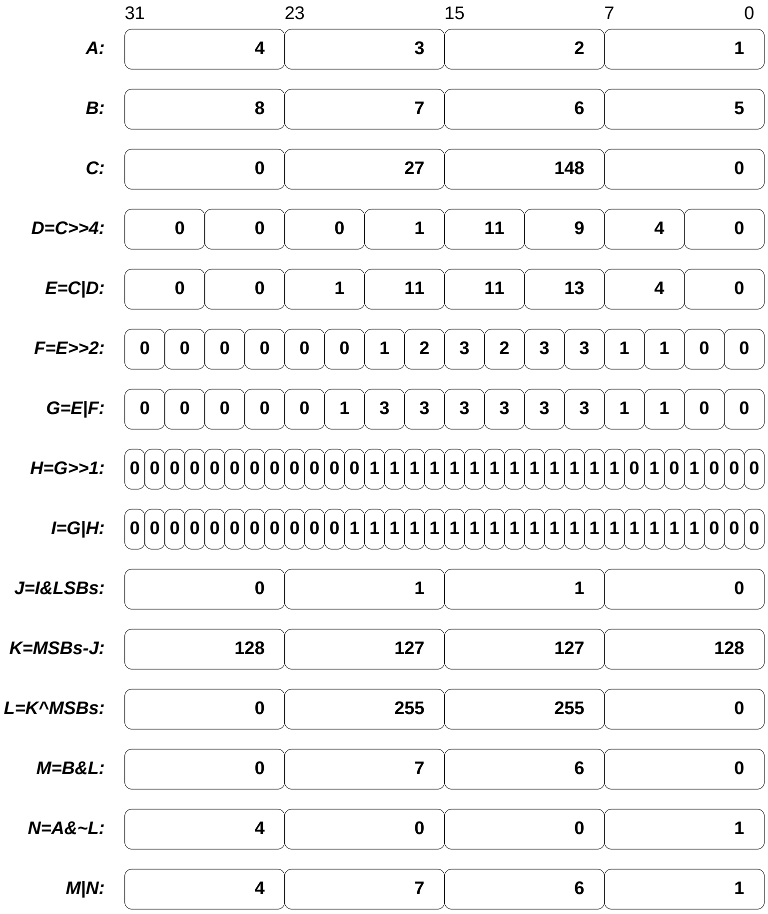
Reductions are operations in which the values stored in the fields of a register are combined into a single result value by successively applying an operation to an intermediate result and each of the field values. The final value may be stored in one or all of the fields of the result register.

For example, if 4x8 a partitioned register contains the values { 3, 4, 9, 18 }, we may store the result of a reduceAdd (3+4+9+18=34) in each of the fields to form the single result 34. In the following example, we perform an unsigned reduceAdd of the fields of the 4x8 partitioned register A containing the values { 4, 3, 2, 1 } to form the single result value 10:



This operation is performed recursively. First, the register is split into the even (multiples of two) and odd (non-multiples of two) fields using masking operations. Then, using an unpartitioned shift, the fields are aligned and added. The result is a register with half as many fields, but each is twice as large — conveniently ensuring that overflow will not occur, even if no spacers are used. This process is repeated until only one field remains.

2.4. Enable Masking



One of the distinguishing characteristics of SIMD computation is the ability to disable processing elements for portions of a computation. Unfortunately, SWAR hardware does not allow fields to be disabled per se; instead, a form of arithmetic nulling must be used. Consider the simple SIMD code fragment:

```
WHERE (C) A=B
```

In this code, `A`, `B`, and `C` are all vectors of the same length. Where the corresponding field of `C` is true (non-zero), that field within `A` should be replaced with the value from the corresponding field from `B`. Other elements of `A` should be unaltered. If it were possible to disable fields, the above statement could be executed very straightforwardly by disabling fields corresponding to 0 values within `C` and then simply having the enabled processing elements execute `A=B`.

Without the ability to disable fields, we will need to arithmetically nullify the undesired computations. The first step is to make the SIMD code fragment symmetric:

```
WHERE (C) A=B ELSEWHERE A=A
```

This looks strange, but accurately reflects the fact that the fields of `A` that are to be unaffected by the `WHERE` must literally be actively read and pasted-together with the fields taken from `B`.

There are a variety of techniques that can be used to arithmetically merge the appropriate fields taken from `A` and `B`, but by far the cheapest is to use bitwise `AND` to mask the undesired field and then use bitwise `OR` to merge the masked results. Indeed, this is the approach used in the example at the start of this section.

First `C` must be converted into an appropriate mask, which has a field full of 0s where `C` was 0 and a field full of 1s where `C` was non-zero. A log-length sequence of unpartitioned shifts and bitwise `OR` operations is used to convert `C` into a usable mask, `L`. This mask is then used to select the appropriate fields of `A` and `B`. In terms of scalar C code:

```
if (c) a=b; else a=a;
```

Essentially became:

```
l = -(c != 0);
a = ((b & l) | (a & ~l));
```

Although the SWAR code looks somewhat strange, this arithmetic masking is actually an old trick borrowed from various early SIMD machines. For example, the Thinking Machines CM-2 used this approach.

3. Compiler Optimizations For SWAR

Optimizations that have been devised for SIMD programming often apply in a natural way to SWAR programming. However, the unique features of SWAR execution also motivate a variety of new compiler technologies. Three such technologies are briefly discussed here: promotion of field sizes, SWAR value tracking, and enable masking optimizations.

3.1. Promotion Of Field Sizes

Just because a high-level language program stated that a value needed only k bits does not mean that *precisely* k bits worth of hardware must be used. For example, 16 bit values are handled very well by HP's PA-RISC MAX, but smaller sizes are not. Thus, a vector that was declared as containing 14-bit values will yield more efficient code sequences if the array's object type is promoted to 16-bit fields. This promotion is particularly favorable because the number of 16-bit fields that fit in a 64-bit register is precisely the same as the number of 14-bit fields that would fit — using 14-bit fields would not add any parallelism. Not all hardware-unsupported field sizes are inefficient; for example, 2-bit fields are not directly supported by any of the SWAR hardware, but for all the SWAR implementations described, the extra parallelism width makes operations on 2-bit fields far more effective than promoting these fields to 8-bits. In general, for each SWAR implementation, there are certain field sizes that are less efficient than a somewhat larger field size, and thus none of these inefficient field sizes should be directly used.

Notice that this promotion of field sizes, and even the use of spacer bits, can result in different data layouts for the same vector on different computers, and the user-specified field sizes do not imply any particular field layout. However, by using a separate SWAR module compiler that communicates only through C code interfaces that it generates, we can ensure that non-obvious machine-dependent layouts are not visible outside the SWAR modules.

3.2. SWAR Value Tracking

Nearly all traditional compiler optimizations are based on tracking which values are available where and when. For example, common subexpression elimination (CSE) depends on this analysis. The interesting question that SWAR technology raises is what constitutes the basic unit of SWAR data that we wish to track?

Fundamentally, one would like to track the values of all the fields within a SWAR register or vector. However, as we have discussed, the compiler's treatment of SWAR coding often results in code sequences that dynamically change the apparent partitioning, and this would have the effect of destroying the field-based value tracking information. Similarly, as discussed in sections 2.1.2 and 2.1.3, it is often desirable to use spacer bits between fields to allow ordinary instructions to function as partitioned operations, and these spacer bits are by definition not part of the fields they separate. To optimize spacer manipulations, we need to be able to track spacer values as well as field values. In fact, unpartitioned operations manipulate both field and spacer values alike, and operations such as shifts can transform one into the other.

To support aggressive SWAR optimizations, a new value tracking method is necessary. We suggest that symbolic tracking of values for arbitrary masked-bit-patterns within a register is appropriate. The following two subsections give brief examples of the benefits of such tracking.

3.2.1. Bitwise Value Tracking

It is not surprising that SWAR code often uses many bitwise masking (`AND` and `OR`) operations and unpartitioned shifts using constant-valued masks and shift distances. Consider a simple C code example:

```
x = (( (x & 0x00ff) << 4 ) & 0xff00);
```

By tracking how arbitrary masked-bit-patterns are shifted in this sequence, this code can be converted to the equivalent, but simpler, form:

```
x = (( x << 4 ) & 0x0f00);
```

In general, this type of tracking can merge multiple `AND` or `OR` operations through shifts, as well as merging shifts.

3.2.2. Simplification Of Spacer Manipulation

It is not unusual that the manipulations of spacer bits will become a significant fraction of all the instructions executed. Thus, any optimization technique that can reduce the frequency of spacer manipulations is highly desirable.

In preparation for a partitioned operation, the spacer bits may have to be set to particular values which depend on the operation. For example, if the operation is an addition, the spacer bits should be set to 0 in both operand registers; for subtraction, one should be set to a 1 and the other should be set to 0. After a partitioned operation, a carry or borrow may alter these spacer bit values. Thus, it may be necessary to zero all the spacer bits to correctly isolate the fields.

Actually, most operations can alter the values of spacer bits. The interesting fact is that even though most polymorphic instructions, such as bitwise operations, can alter spacer bit values, they produce field values without being affected by the values of spacer bits. Thus, these instructions offer the opportunity to set spacer bits to the next desired value at no cost. For example, consider computing `e=((a+b)-(c+d));` using a SWAR representation employing spacer bits identified by the mask `s`:

```
e = (((((a & ~s) + (b & ~s)) & ~s) | s) -
      ((((c & ~s) + (d & ~s)) & ~s) & ~s)) & ~s;
```

We would expect conventional compiler optimizations to eliminate the redundant `AND` of the subtrahend with the one's complement of the spacer mask required by the field isolation stage of the addition and the normalization stage of the subtraction. This would save one operation, and change the calculation of the subtrahend from

```
((((c & ~s) + (d & ~s)) & ~s) & ~s)
```

To:

```
(((c & ~s) + (d & ~s)) & ~s)
```

By using spacer value tracking, we may be able to make significantly larger reductions in the number of spacer manipulations needed. Suppose that the spacer bits for each of `a`, `b`, `c`, and `d` were already known by the compiler to be zeros. The normalizations for the additions would not be required, and the original code could be compacted to the following:

```
e = ((((a + b) & ~s) | s) -
      (((c + d) & ~s) & ~s)) & ~s;
```

Further inspection reveals that the isolation stages following the additions are unnecessary because they are immediately followed by the normalization stage of the subtraction, which overwrites the spacer values just written. This last observation reduces the original 12 operations to its final form, with just 6 operations (given that `s` and `~s` are constants):

```
e = (((a + b) | s) |
      ((c + d) & ~s)) & ~s;
```

It is also interesting to note that by the same analysis, computing `e=((a+b)-((c+d)&M));` where `M` is a constant-valued mask, could be accomplished in the same number of instructions:

```
e = (((a + b) | s) |
      ((c + d) & (M & ~s)) & ~s;
```

This is because `(M & ~s)` is also a constant. In general, the bitwise operations and unpartitioned shifts discussed in the previous section can be optimized at the same time as the spacer manipulation.

3.3. Enable Masking Optimizations

Because SWAR hardware does not allow fields to be disabled, there is a significant cost associated with the arithmetic nulling of undesired field computations. However, this cost need not be incurred if the compiler's static analysis can prove that *all* fields are active, or if the compiler can generate code that allows all fields to be active later correcting the should-have-been-inactive field values.

For "virtualized" processing elements, a single vector may span the fields of multiple words (registers). Thus, enable masking would have to be performed on each of the words. This allows the compiler to generate three different versions of the word-wide SWAR code, and to select among them by examining each word worth of enable mask:

- If the enable mask word is entirely enabled, then no masking is done; the corresponding word of the result is directly computed.
- If the enable mask word is entirely disabled, then the corresponding word of the result is copied from the original value — no computation is done.
- If the enable mask word is partially enabled, then the usual masking is used in computing the result.

4. Conclusion

The latest processors from DEC, HP, MIPS, Intel, AMD, Cyrix, and Sun (and also the soon-to-be-announced next generation of the PowerPC) have proven that hardware support for SWAR, SIMD Within A Register, is easily and efficiently added to a conventional microprocessor. However, more than year after most of these processors became available, there are still no general-purpose high-level languages that allow them to be programmed using a portable SIMD model. Essentially by design, these SWAR hardware implementations are tuned for hand-coding of specific algorithms, with sparse coverage of SWAR functionality driven by the need to minimize disturbance of the base processor instruction set and architecture.

In this paper, we have shown that these flaws can be largely overcome by the combination of clever coding sequences and a few new types of compiler analysis and optimization. Only the lack of random inter-field communication is an unsolvable problem, but these operations have been avoided in many SIMD algorithms because many traditional SIMD machines also lacked the necessary hardware support.

We currently are working on public-domain SWAR module compilers that will compile portable SIMD programs into SWAR modules with compiler-generated C code interfaces. These compilers will target at least MMX and "generic" 32-bit integer processors. At this writing, however, only the basic library for using MMX with Gnu C has been released. It, and our compilers when they become available, can be accessed at:

`http://shay.ecn.purdue.edu/~swar/Index.html`

References

[AMD97] Advanced Micro Devices, Inc., *AMD-K6 Processor Multimedia Extensions*, Advanced Micro Devices, Inc., Sunnyvale, California, March 1997.

[Cyr96] Cyrix Corporation, *Multimedia Instruction Set Extensions for a Sixth-Generation x86 Processor*, Cyrix Corporation, `ftp://ftp.cyrix.com/developr/hc-mmx4.pdf`, August 1996.

[DEC96] Digital Equipment Corporation, *Alpha Architecture Handbook, Version 3*, Digital Equipment Corporation, Maynard, Massachusetts, October 1996.

[Int96] Intel Corporation, *Intel Architecture MMX Technology: Programmer's Reference Manual*, Intel Corporation, `http://developer.intel.com/drg/mmx/Manuals/prm/prm_covr.htm`, March 1996.

[Lee95] Ruby B. Lee, Accelerating Multimedia with Enhanced Microprocessors, *IEEE Micro*, 15(2):22-32, April 1995

[LeH96] Ruby Lee and Jerry Huck, *HP Technical Computing — 64-bit and Multimedia in the PA-RISC 2.0 Architecture*, Hewlett-Packard Company, `http://hpcc997.external.hp.com:80/wsg/strategies/pa2go3/pa2go3.html`, June 1996.

[SGI96] Silicon Graphics, Inc., *MIPS Digital Media Extension*, Silicon Graphics, Inc., `http://www.sgi.com/MIPS/arch/ISA5/`, 1996.

[Sun97] Sun Microsystems, Inc., *The VIS Instruction Set, Sun Microelectronics*, Sun Microsystems Corporation, `http://www.sun.com/sparc/vis/index.html`, April 1997.