Vector Instruction Set Support for Conditional Operations

J. E. Smith
Dept. of Elect. and Comp. Engr.
University of Wisconsin-Madison
Madison, WI 53706
jes@ece.wisc.edu

Greg Faanes
Silicon Graphics Inc.
1168 Industrial Blvd.
Chippewa Falls, WI 54729
gjf@sgi.com

Rabin Sugumar
Sun Microsystems Inc.
901 San Antonio Road
Palo Alto, CA 94303
Rabin.Sugumar@Eng.sun.com

Abstract

Vector instruction sets are receiving renewed interest because of their applicability to multimedia. Current multimedia instruction sets use short vectors with SIMD implementations, but long vector, pipelined implementations have a number of advantages and are a logical next step in multimedia ISA development.

Support for conditional operations (as occur in loops containing IF statements) is an important aspect of a vector ISA. Seven ISA alternatives for implementing conditional operations are systematically explored. Performance considerations are discussed through evaluation of a typical IF loop over a range of vector lengths and *true* conditional values. An approach using masked operations is shown to be one of the better methods, especially if its implementation is able to skip over blocks of *false* mask bits. Additional analyses of complex IF loops and parallel pipeline implementations support the masked operation approach. The paper concludes with a practical implementation of masked operations that skips over power-of-2-length blocks of false values. This implementation is simpler than skipping arbitrary-length blocks and provides similar performance.

1. Introduction

Vector instruction set architectures (ISAs) can express large amounts of data parallelism in a very compact manner. Furthermore, vector implementations are hardware efficient with simple, counter-based control structures. Historically, vector ISAs have been used in large scale numeric processors [1-9], and, as is the case with many supercomputer concepts, they have migrated to microprocessors where they show promise for multimedia applications. Many microprocessor companies have developed short vector multimedia extensions which are implemented in a parallel SIMD fashion. For example, one of the more advanced is the Motorola PowerPC AltiVec extension [10]. In contrast, a long vector ISA is one that specifies many operands (typically several 10s to 100s) in a single instruction. These instructions are executed in a pipelined fashion and require many cycles to execute. Long vector instructions sets with pipelined implementations have been proposed for multimedia applications [11-14] and have been shown to have significant advantages in terms of die area, power requirements, and performance.

Ultimately, the effectiveness of a vector ISA depends on its ability to vectorize large quantities of code. The basic memory, arithmetic, and logical operations are sufficient for vectorizing many straight-line loops. For higher levels of vectorization, however, it is important to vectorize loops containing conditional operations. In terms of high level language programs, these are loops containing IF statements.

Fig. 1 is a simple code example and a compilation for a Cray-1-like vector instruction set. Throughout this paper we assume pipelined vector implementations using vector registers. Because we are focusing our attention on control constructs, and not data types, the results should be equally applicable to floating point, fixed point, long or short precision data.

```
for (i=1, i<=looplen, i++) {
  if (a(i) == b(i)) {
    c(i) = a(i) + d(i);
  }
}
```

```
V1    <- a              .load a(i)
V2    <- b              .load b(i)
VM1   <- V1 == V2       .compare a and b; result to VM1
V3    <- d              .load d(i)
V4    <- V1 + V3        .add a and d
V5    <- c              .load c(i)
V5    <- V4; VM1        .merge sum into c
c     <- V5             .store new c(i)
```

Fig. 1. Compilation of an IF-loop using a Cray-1-like vector instruction set.

In the above example, the elements of *a* and *b* are compared pair-wise with a vector mask (VM1) containing the results of the comparisons. Then, the individual mask bit values are used to merge the values of *c* and *a+d*.

The "merge under mask" implementation is but one of several ways that conditional operations can be implemented with vectors. The primary focus of this paper is vector ISA design. This is accomplished by examining a number vector instruction set alternatives for implementing conditional loops, discussing the important tradeoffs, and arriving at conclusions regarding the better alternatives.

An important characteristic of several alternatives is that they provide performance proportional to the numbers of "true" results from the IF statement. To achieve this performance

characteristic when using vector masks requires implementations that are able to skip over zeros in the vector mask. Skipping arbitrary numbers of zeros each cycle is relatively complex, however, so a second objective of this paper is to propose and evaluate a simpler method that provides good performance.

Following are important considerations and definitions we will use when studying vector instructions for implementing conditional operations.

Performance *VL-time vs. Density-time performance* In some implementations, performance is directly proportional to the vector length (VL) -- as is the case with unconditional vector operations. For other ISA implementations, performance may be sensitive to the number of true (or false) cases. If the performance is directly proportional to (VL) and independent of the true/false values, then the implementation has *VL-time* performance. If the performance is related to the fractions of true/false cases, the implementation has *density-time* performance (referring to the density of true (or false) values.)

A good way of estimating vector performance is to measure time as *chain times* or *chimes*. A group of vector instructions that execute in parallel, either because they are independent or because they can be chained together, are considered to consume a single chime. The number of chimes required is related to the hardware resources available. Assuming a single load/store vector pipeline, the example in Fig. 1 consumes 5 chimes. The first instruction is one chime, the next two instructions chain together and consume a second chime. The fourth and fifth instructions consume a third chime. The sixth and seventh consume a fourth chime, and the final store instruction consumes the fifth chime. If there were two load/store vector pipelines, three chimes would be required.

Generality A good vector instruction set must be able to vectorize conditional loops of any structure and complexity. We will perform an initial performance analysis using the simple vector loop in Fig. 1, and then discuss the vectorization of more complex, nested-IF structures.

Implementation Efficiency The vector ISA can clearly affect the underlying implementation. For example, some methods may require more vector registers or multiple vector masks for efficient implementations. Other methods affect data path connectivity; we will see that this is the case with parallel-pipeline implementations.

Safety When a high level language (HLL) is used to specify a vectorized program, the vectorized version should be true to the semantics of the HLL. A vector ISA implementation is *unsafe* if it can cause exception conditions that do not occur when high level language (HLL) semantics are strictly followed. The vector code in Fig. 1 is unsafe because *a(i)* and *d(i)* are added for all values of *i*, regardless of whether the IF condition is true. It is possible that one of these additions could trigger an overflow even though the IF condition is false; this overflow would not occur if the HLL semantics were followed. Similarly, the load of *d* and store to *c* are unsafe because they may generate an out-of-range address that would not occur in the HLL version.

2. Simple IF Loops

We first consider a number of vector ISA alternatives for vectorizing simple IF loops. The loop in Fig. 1 is a good example that is adequate to clearly illustrate the important tradeoffs and issues that are involved. The IF loop in Fig. 1 has one operand (*a(i)*) that is first accessed in its entirety (to perform the IF test), and is then accessed again within the scope of the IF statement. A second operand (*d(i)*) is loaded from memory only within the scope of the IF statement.

Below we vectorize the Fig. 1 loop using a number of vector ISA alternatives. Each is shown in a pseudo-assembly language, and for clarity, loop setup and surrounding strip-mine code is not shown. Following the discussion of the various compilations, we evaluate the performance of each.

2.1. Compilations

Method 1: merge only

```
V1    <- a              .load a(i)
V2    <- b              .load b(i)
VM1   <- V1 == V2       .compare a and b; result to VM1
V3    <- d              .load d(i); assume safe
VM2   <- !VM1           .complement VM1
V3    <- 0; VM2         .merge zeros into d(i)
V4    <- V1 + V3        .safe add a and d
V5    <- c              .load c(i); assume safe
V5    <- V4; VM1        .merge sum into c
c     <- V5             .store new c(i)
```

The first "style" of vectorizing is similar to Fig. 1, but eliminates the unsafe addition of *a(i) + d(i)*. This code uses the vector merge instruction as the primary masked operation; ordinary vector instructions are not masked. This code is simplified by using an ISA that supports multiple VM registers. A single VM could be used with some means for backing it up; e.g. spills to memory. Masked versions of the individual vector instructions are not required. This method of vectorizing conditionals was supported in the original Cray-1 [5].

The addition of arrays *a* and *d* is made safe by merging safe operands into all the vector positions where the IF predicate is false prior to doing the add. The above code does this by forcing a zero into the false positions of one of the vectors; adding 0 can not result in overflow. In general, methods of this type can be used to make any arithmetic operation safe.

Memory operations may still be unsafe, however. In some situations, the compiler can guarantee safety by discerning the relative sizes of the arrays and the number of loop iterations. Unfortunately, this cannot always be done. The compiler or programmer can also insert runtime checks. Even if something is done to avoid out-of-range address, spurious page faults can still occur in a virtual memory implementation.

Method 2: masked operations

```
V1    <- a              .load a(i)
V2    <- b              .load b(i)
VM    <- V1 == V2       .compare a and b; result to VM
V3    <- d; VM          .load d(i) under mask
V4    <- V1 + V3; VM    .add under mask
c     <- V4; VM         .store to c(i) under mask
```

The second vectorization method uses an ISA where all the basic vector operations can be performed under a mask specified as part of the instruction. Only operations for "true" mask values are performed; "false" masked operations are effectively

no-ops. Additional instruction bits are needed to specify the vector mask register (if multiple VMs are used). This method is supported by the Fujitsu, Hitachi, and NEC processors [1-4]. A similar form was also supported in the memory-to-memory ISAs used in the CDC Cyber 200 series [8] and the BSP [9].

The code is straightforward. The *a* and *b* operands are loaded and compared to form a mask. Then, all instructions corresponding to operations within the scope of the IF statement are performed under the mask. All operations are "safe"; only real exceptions occurring in the HLL program are reported.

Method 3: memory gather/scatters

```
V1      <- a            .load a(i)
V2      <- b            .load b(i)
VM      <- V1 == V2     .compare a and b; result to VM
V3      <- IOTA (VM)    .form index set
VL      <- pop (VM)     .find new VL
V4      <- a, V3        .gather a(i) values
V5      <- d, V3        .gather d(i) values
V6      <- V4 + V5      .add a and d
c(i),V3 <- V6           .scatter sum into c(i)
```

Method 3 uses memory gather/scatters as the means for implementing the IF. The IOTA instruction [5] forms a vector of index values that indicate the true bit positions in the vector mask. The "pop" instruction is a population count that counts the number of ones in the VM register. The gather instruction uses the address of *a* as a base address and the indices in a vector register to gather (load) non-contiguous values from memory. The scatter (store) to memory is the complementary operation. All operations are safe. Some members of the *a* array are loaded from memory twice: all members of array *a* are loaded for the compare, and a subset is gathered for the add. This is one of the fundamental vectorization methods and is supported by the Cray-2 [5], the Cray X-MP-4 [7], the VP100/VP200 [2-3], and NEC SX processors [4].

Method 4: memory compress/expand

```
V1      <- a            .load a(i)
V2      <- b            .load b(i)
VM      <- V1 == V2     .compare a and b; result to VM
V3      <- a, VM        .compress-load a(i) values
V4      <- d, VM        .compress-load d(i) values
VL      <- pop(VM)      .get no. elts.
V5      <- V3 + V4      .add a and e
c,VM    <- V5           .expand-store sum into c(i)
```

Method 4 uses memory compress/expand operations as the means for implementing the IF. These instructions are like gather/scatters except they use the VM directly to determine which operands to load or store; the index vector does not have to be generated via an IOTA instruction. All operations are safe. As with gather/scatter, some members of the *a* array are loaded twice. Note that the memory compress-load as shown above assumes unit stride; for non-unit strides, the scalar stride value has to be provided to the memory compress/expand instructions. This method of vectorization is supported by VP100/200 [2-3], and a similar form was supported by the memory-to-memory vectors in the Cyber 200 series [8].

Method 5: register compress/expand

```
V1      <- a            .load a(i)
V2      <- b            .load b(i)
VM      <- V1 == V2     .compare a and b; result to VM
V3      <- V1 c VM      .compress a values
V4      <- d            .load d(i); assume safe
V5      <- V4 c VM      .compress d values
V6      <- c            .load c(i); assume safe
A1      <- VL           .save VL
VL      <- pop (VM)     .find new VL
V7      <- V3 + V5      .add
V6      <- V7 e VM      .expand into c
VL      <- A1           .restore VL
c       <- V6           .store modified c(i)
```

Method 5 uses register compress/expand operations as the means for selecting operands. In this form, the equivalent of gathers and scatters are implemented by combining ordinary load/stores with register-to-register compress/expands. With this method, some memory operations may be unsafe because all elements of the arrays are first loaded, then elements are selected via compress/expands. The arithmetic operations are safe because they are done after the compresses. Register compress/expand operations are supported by the NEC SX vector processors [4].

Method 6: gather/scatter plus register compress/expand

```
V1      <- a            .load a(i)
V2      <- b            .load b(i)
VM      <- V1 == V2     .compare a and b; result to VM
V3      <- IOTA (VM)    .form index set
V4      <- V1 c VM      .register compress a(i) values
VL      <- pop (VM)     .find new VL
V5      <- d, V3        .gather d(i) values
V6      <- V4 + V5      .add a and d
c,V3    <- V6           .scatter sum into c(i)
```

Method 6 is a "hybrid" solution that assumes both memory gather/scatters and register compress/expand. Mostly, the gather/scatter is used, but the register compress is used to avoid a re-load of data that is already in a register (the *a(i)* values in this example.) All operations are safe.

Method 7: memory plus register compress/expand

```
V1      <- a            .load a(i)
V2      <- b            .load b(i)
VM      <- V1 == V2     .compare a and b; result to VM
V4      <- d, VM        .compress-load d(i) values
V3      <- V1 c VM      .register compress a(i) values
VL      <- pop(VM)      .get no. elts.
V5      <- V3 + V4      .add a and d
c,VM    <- V5           .expand-store sum into c(i)
```

Method 7 is another hybrid solution, very similar to method 6. Here, the second load of the *a* array values is avoided. As in method 6, all operations are safe.

2.2. Performance Analysis

To study performance tradeoffs, simulation was used to time vector loops. Following are the simulator characteristics.

--single-width pipelined functional units
--one memory port (supports both loads and stores)
--memory and function unit chaining enabled
--load vector startup time: 18 clock cycles
--function unit startup time: 6 clock cycles

An important performance issue is whether sequences of zeros in the VM can be skipped simultaneously. In particular, there are two basic ways of implementing masked operations. One is to perform the false operations as no-ops where the control hardware initiates the operations, and they consume functional unit pipeline slots. Results (and errors) are simply cancelled and don't modify the machine state. This type of masked operation consumes time that is independent of the values of the VM bits and is proportional to the vector length, hence it is a "VL-time" implementation of the masked operations. The other implementation selects only the true VM operations and skips over blocks of false operations in a single clock cycle. Pipeline slots (and time) are only used by the true operations. Consequently, time is proportional to the number of true values in VM (or the "density" of true values), so this is a "density-time" implementation of masked operations.

Fig. 2 shows performance for all seven of the vectorization methods. Each graph shows performance in clock periods as a function of the "true" density ranging from 0 to 100 percent. The graphs are organized in pairs, with one pair for each loop length simulated (10, 100, 1000). The left member of the pair shows performance with a VL-time implementation of masked operations, and the right member shows performance with a density-time implementation of masked operations. The density-time assumption extends to IOTA instructions where a mask is converted into a set of index values. This particular loop was chosen because it highlights fundamental performance differences and the results correlate well with simulations for other IF loop variations.

2.2.1. VL-time mask operations

First, consider performance of the VL-time implementations of masked operations. These graphs are in the left column of Fig. 2. Note that even though these results are for VL-time *mask* implementations, the methods that use data in a compressed form (via gathers or compresses) still exhibit some density-time behavior.

Overall, the best method appears to be Method 7. Method 3 is sometimes better for longer vectors and sparse VMs because method 3 has more chimes that can run in density-time. Method 2 is insensitive to the true density. Consequently, it is as good as Method 7 for very high densities, but not as good for sparse densities; the performance difference is larger for longer vectors as the effects of vector startup overheads diminish.

All the other methods appear inferior to methods 2, 3, and 7. Method 1 has one more chime than many of the other methods because it has to merge zeros into the *d(i)* vector to assure safety. Method 4 also has one chime more than the better-performing methods because it must re-load array *a* in a compressed form. Consequently it gives mid-to-low performance. Method 5 has only register compress/expand at its disposal, and it consumes the most chimes of any of the methods. This loop is dominated by memory operations, and with only register compress/expand, all the memory operations must be performed for the full VL.

Method 6 gives worse performance than method 3, which seems counter-intuitive because it can use both gather/scatter and register compress/expand while method 3 can only use gather/scatter. Method 3 performs better because the *a* array values are selected in density-time via a memory gather. Method 6 does the compression of the *a* array values from a register, but this is based on VM and takes VL-time. Although Method 6 has one less memory operation, it takes more time. Method 6 is slower than method 7 because it was assumed that the register compress and IOTA instructions use the same functional unit and can not be executed in the same chime. With separate units, the performance of Methods 6 and 7 would be similar.

2.2.2. Density-time Mask Operations

The performance with density-time mask operations are in the right column of Fig. 2. Comparing the two columns, we see that across the board the density-time implementations perform better than the VL-time implementations, as would be expected.

Method 2 is the best performer with a density-time implementation. There is no additional overhead for IOTA, reloading data etc.; it only does operations that are actually needed, and does them in the minimum (density) time. Method 7 improves with a density-time implementation, and gives performance equal to method 2 except for short vectors where method 2 is better. This happens because method 7 has a register compress, adding pipeline latency to a chime.

Methods 1 and 5 show no performance difference when going to a density-time implementation because density-time instructions (if any) are in the same chime as a VL-time instruction. Method 3 shows no performance difference because its VM-sensitive instructions are all gathers based on an index vector; they are not dependent on masked operations that use VM. Methods 4 and 6 improve with density-time masked operations, but remain in the middle of the pack, performance-wise.

2.2.3. Preliminary Conclusions

Based on this initial evaluation, a preliminary conclusion is that Method 2, which relies solely on masked operations, is the best method for vectorizing IF statements. Method 2 generates safe code, and is one of the better performers for both VL- and density-time implementations. It provides the best performance overall when coupled with a density-time VM implementation. Practical density-time implementations are discussed in Section 5. Method 7 produces safe code and also produces very good performance for both VL and density-time implementations. However, the next section will show that Method 7 is inferior for more complex IF loops.

3. More Complex IF loops

The evaluation in the previous section was for simple IF loops. Simple IF loops are the most common, and performance for these common cases is very important. However, more complex IF structures often occur in practice, and a vectorization method must be able to handle them in an efficient manner. We now consider more complex IF structures to evaluate the flexibility of the alternative vector ISAs. In the process, we will discuss other issues of vector ISA design.

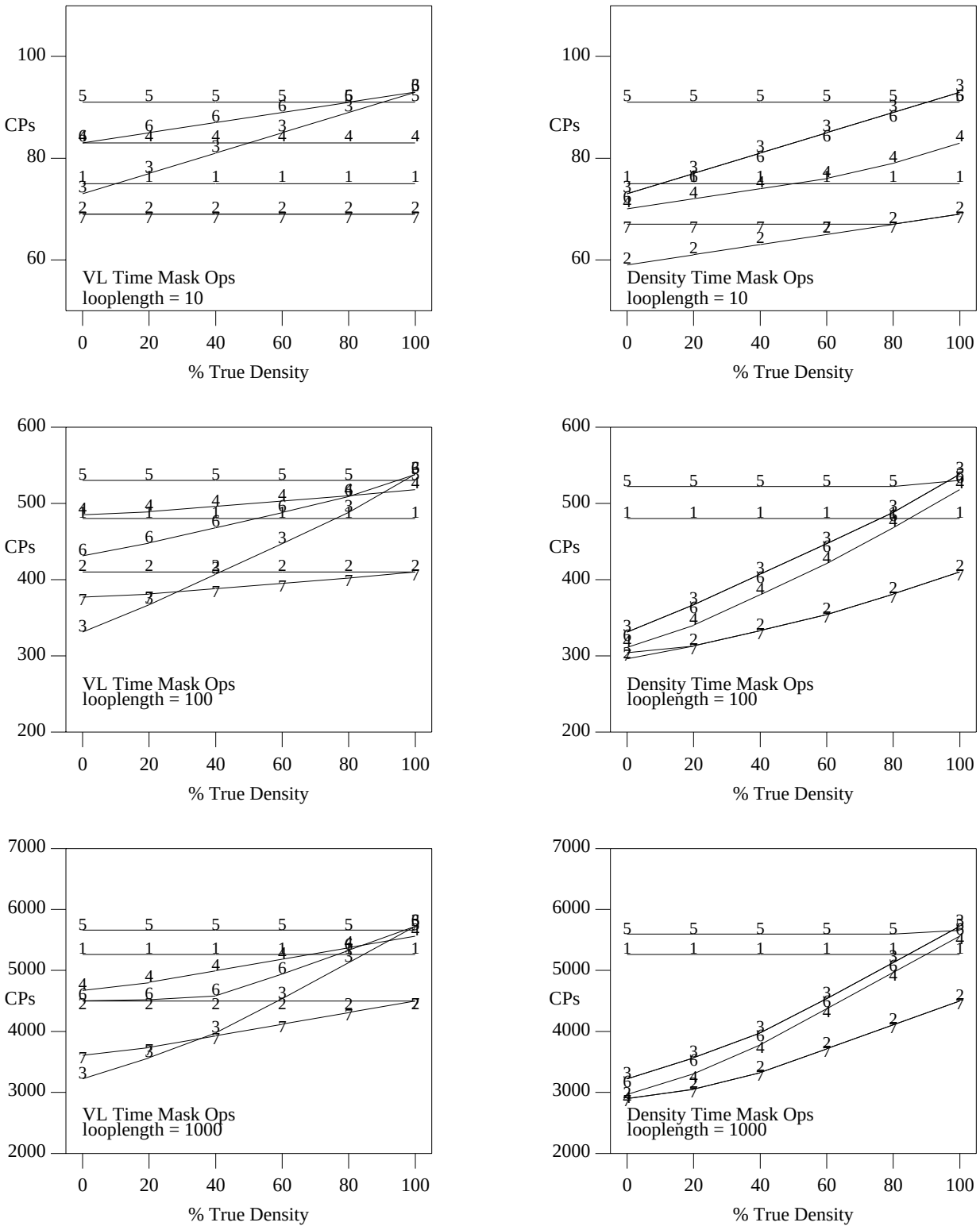

Fig. 2. Performance in clock periods for a variety of vectorization methods as a function of the density of "trues" in VM.

IF statements select subsets of arrays to be operated on, and the different methods of vectorizing IF loops use different ways of representing these subsets. The masked operation method holds array elements in a vector register with their original relative positions retained. A vector mask register is used to indicate those elements that belong to the selected set. This is illustrated in Fig. 3a. Here, a vector register holds a portion of an array from memory (in this example, loaded at the beginning of the array, stride one.) The elements of the array are in the same relative positions in the vector register as in memory. The VM selects a subset of the vector elements, and consequently, a subset of the memory array. Other subsets of the same vector can be represented by filling in more elements of the vector register and using additional vector masks. Fig. 3b uses two VM registers to represent two subsets of the vector; the data for both subsets is held in the same vector register, and the relative positions of the elements are the same as in memory.

This points out a valuable ISA characteristic: there are multiple VM registers, and VMs are not associated with specific vector registers (VR). This allows a VM to identify subsets of different vectors, and a vector can hold data from multiple subsets if it has multiple VMs associated with it.

The pure gather/scatter method, as exemplified by Method 3, represents subsets of data differently (see Fig. 4a). Here, a vector register (densely) holds only the members of the selected subset. A second vector register holds their indices. The compress/expand methods (e.g. Method 7) are similar (Fig. 4b). In this case data is still held densely, but a VM is used to indicate the elements in the set. If one wants to hold multiple subsets, then a separate vector register is required for each data subset and a different vector register or VM is needed for each subset.

Other efficiencies become apparent when one does set operations on the data, as is done when complex IF loops are vectorized. To illustrate this point, consider Fig. 5, where array subsets for a number of IF structures are represented in Venn diagram notation. Fig. 5a shows a simple IF loop where a subset of array elements are selected. In the IF ELSE, a subset is

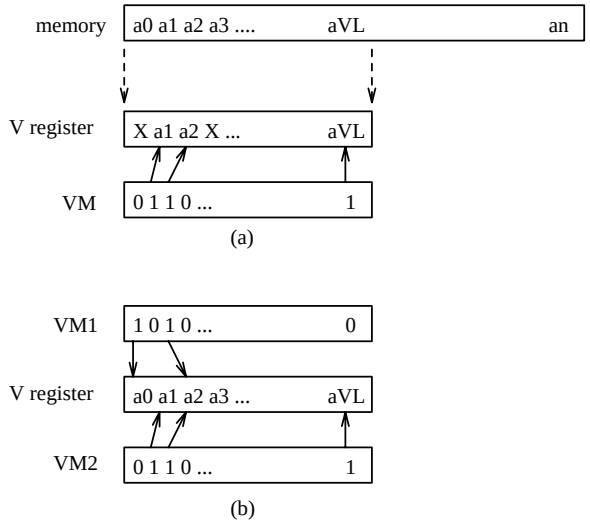

Fig. 3. Using masked operations to represent vector subsets.

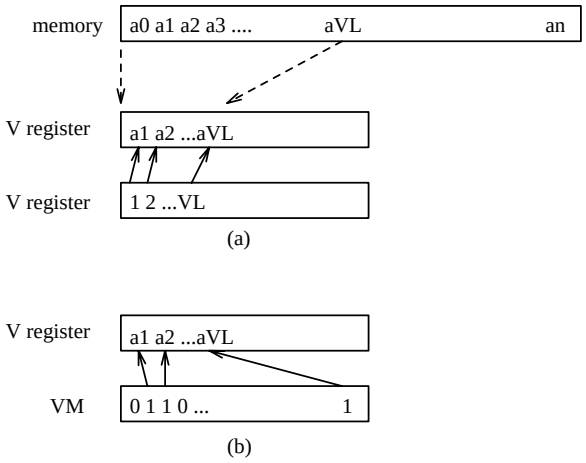

Fig. 4. Methods of representing vector subsets; a) Gather/scatter b) Compress/expand

selected by the IF, and all the other elements are selected by the ELSE. The other two cases show subsets selected by IF ELSE IF and nested IF constructs. Of course, even more complex subset structures are possible by using combinations of IFs, ELSEs, and nested IFs.

Given the relationship between set operations and nested IF structures, we now consider the ease with which set operations can be performed when sets are represented as in Figs. 3 and 4. To perform set operations on the sparse vector representation of Fig. 3, one only has to do bit-wise logical operations on the VM values (e.g. set union is a bit-wise OR), and make sure that the selected values have been copied from memory into the vector register. If the data values are already in the vector register, only VM operations are required.

Performing set operations on the dense representations as in Fig. 4, however, requires the merging of index vectors and/or dense data vectors. Merging index vectors would likely require a

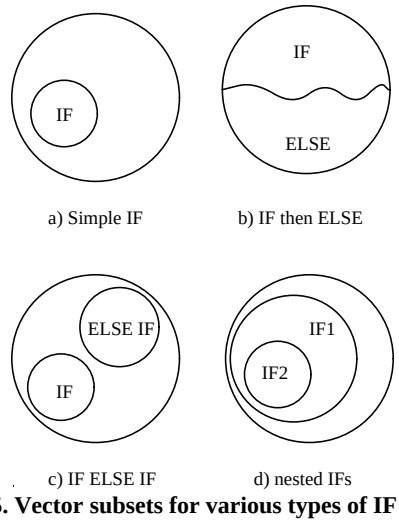

Fig. 5. Vector subsets for various types of IF loops.

special instruction for that purpose, or a sequence of 1)a conversion to VM, 2) logical operations on VM, and 3) conversion back to an index set. Even if the VM form is used, the dense data representation still poses problems.

Consider the following example loop. It has two examples of "subsetting". First, all the elements of the *b* array are loaded and tested; later a subset of *b* is used (in *c(i)=d(i) + b(i)*). Second, a subset of the *d* array is first modified, and at an outer IF-level, a superset is used to compute *c*.

```
for (i=1, i<=looplen, i++) {
  if (b(i) != 0) {
    if (c(i) == 0) {
      d(i) = a(i) + b(i);
    }
    c(i) = d(i) + b(i);
  }
}
```

Following is code to do implement the above loop using VM operations (strip mine code not shown).

```
V1      <- B(i)              .load b(i)
VM1     <- V1 != 0           .test b != 0
V2      <- C(i); VM1         .load c(i) under mask
VM2     <- V2 == 0; VM1      .test c == 0 under mask
V3      <- A(i); VM2         .load a under mask
V4      <- V1 + V3; VM2      .add under mask
D(i)    <- V4; VM2           .store d under mask
VM3     <- VM1 & !VM2        .mask for d elements in memory
V4      <- D(i); VM3         .load new d(i) under mask
V5      <- V4 + V1; VM1      .add under mask
C(i)    <- V5; VM1           .store under mask
```

First, all the *b* elements are loaded into V1 and tested to form VM1. Later V1 is used, unchanged, but with VM2 selecting the subset of *b*, as determined by the *c* values. The first subset of the *d* array is formed using the mask VM2 and is placed in V4 (and stored to memory). When the larger subset of *d* is needed, simple logical operations (VM3 <- VM1 & !VM2) select the elements that are in the set identified by the outer IF, but not in the set identified by the inner IF (these are already in V4). Then, V4 is loaded from memory using mask VM3 to select only the elements of array *d* that haven't already been loaded.

For comparison, a version using a dense representation for subsets is shown below. This method uses gathers and scatters.

```
V1      <- b                 .load b(i)
VM1     <- V1 != 0           .test b != 0
V2      <- IOTA(VM1)         .form index set for c
VL      <- pop(VM1)          .find vl for gather
V3      <- c, V2             .gather c(i) values
VM2     <- V3 == 0           .test c == 0
        .
        .
```

Generating the gather/scatter code begins well, but when we try to load the *a* array, we are stymied. The index set for array *a* is not easily formed without a register compress. VM2 is based on memory indices in different locations than VM1; hence we can't simply AND the two masks. There are ways of arriving at the correct subset, but these involve extra (and very clumsy) gathers and scatters to memory. An alternative is to load the full *c* array (potentially unsafe), test it, then AND the VMs.

If we use memory and register compress/expand instructions as in Method 7, the first several instructions are given below (a total of 22 instructions are required).

```
V1      <- b                 .load b(i)
VM1     <- V1 != 0           .test b != 0
V3      <- c, VM1            .compress-load c(i) values
A1      <- VL                .save VL
VL      <- pop (VM1)         .adjust VL for compressed register
V4      <- V3 e VM1          .expand c values
VL      <- A1                .restore VL
VM2     <- V4 == 0           .test c == 0
VM3     <- VM1 & VM2         .form subset of VM1 elements
V6      <- a, VM3            .compress-load a values
        .
        .
```

The subset selected by the test of *b* is identified by VM1. A compress-load instruction is used to load the selected subset elements of the *c* array. The *c* values are then register expanded to allow a later set operation on VMs. After *c* is tested (more values than required by the source code are examined), the resulting VM2 is ANDed with the VM1 (set intersection) to determine the sets operated on in the inner IF nest. Overall, this method works, but is clumsy (and slow) compared with the method using masked operations given above because the subsets have to be expanded in registers before the required set operations can be done.

This exercise re-enforces our earlier choice of masked operations and shows that this method is preferable to Method 7. Masked operation allow simple representations and operations on data subsets selected by complex nested IF structures.

4. Multiple Pipe Implementations

We now consider an important hardware implementation issue. To allow for implementations having a range of cost and performance levels, a vector instruction set should be efficiently implementable in both single and multiple pipe versions. In a multiple pipe version, the vector elements are interleaved among identical sets of functional units and memory ports. For example, in a four pipe implementation vector elements 0, 4, 8, 12, 16, etc. are operated on by pipe 0; elements 1, 5, 9, 13, 17 are operated on by pipe 1, etc. This type of implementation first appeared in the CDC 200 series processors [8], was popular in the Japanese vector supercomputers [1-4], and was adopted by Cray Research beginning with the CRAY-C90 [15].

Usually, each pipeline only has to operate on the elements that are naturally associated with it; however, some vector instructions result in "cross-pipe" communication where data or other information must be passed among the pipes. The degree and type of cross-pipe communication can have a significant bearing on the cost, complexity, and/or performance of a multi-pipe implementation. We will examine the instructions and methods under consideration for IF loops to see how well they extend to multi-pipe implementations.

4.1. Masked Operations

First, consider masked operations. Here, the VM register(s) have their bits interleaved in the same way as the vector register data elements. Each pipe only needs its VM bits to do the masked operations. At least for the masked operations, including masked loads and stores, no cross-pipe communication is required.

Density-time implementations pose some difficulties, however. If the pipelines work in lock-step, then they can only skip VM zeros if all pipes have zeros, and performance will suffer. Furthermore, this will require cross-pipe communication of the

mask values. A less restrictive approach allows the pipelines to proceed independently, synchronizing only when they all finish. With this implementation, the pipes may take different lengths of time to complete their work. Consequently, some pipes may sit idle while the one with the highest VM density finishes. Consider the following VM:

1011 0000 1000 1000 0101 0001 1011 0000

In a four pipe implementation, pipe 0 will hold VM bits 0, 4, etc. All four pipes hold the following VM subsets:

pipe 0: 10110010 pipe 1: 00001000
pipe 2: 10000010 pipe 3: 10001110

In this above example, pipe 0 will take four clock periods in a density-time implementation, pipe 1 will take one clock period, pipe 2 will take two clock periods, and pipe 3 will take four. If all four pipes are run independently, but synchronize at the end, the overall time will be four clock periods, the maximum time required by any of the pipes. In this example, the density of the entire VM is 11, so the speedup of four pipes versus a single pipe is a little less than three, not four as one might expect.

4.2. Gather/Scatter

An important step in a gather/scatter IF loop is the IOTA instruction that takes a VM and forms an index set. Because the VM bits are interleaved among the pipes, the VM values must be shared among all the pipes so that each pipe can determine which element numbers of the index values it should hold. Referring again to the above example VM, the iota vector is: 0,2,3,8,12,17,19,23,24,26,27. Interleaved among the pipes, the iota vector is:

pipe 0: 0,12,24 pipe 1: 2,17,26 pipe 2: 3,19,27 pipe 3: 8,23

In order for pipe 3 to determine that element 3 of the iota vector is 8, for example, it must be able to see all the VM bits up through the 8th. Sharing the VM bits is a form of cross-pipe communication, but it is a relatively small amount of information that must be shared. Once the index set is distributed among the pipes, the actual gather and scatter require no additional cross-pipe communication.

The memory compress/expand is similar to the gather/scatter, but the index vector is formed implicitly from the mask. Once again, the VM values must be communicated among the pipes, but nothing else.

4.3. Register Compress/Expand

Register compress/expand instructions use VM to control the movement of certain elements from one vector register into another. This means that in general, an element in any pipe can be transferred to any other pipe. In a four pipe implementation, if an element were equally likely to move to any of the four pipes, an average of 3 words of data would have to move every clock period if all four pipes were to work at full speed (i.e. 4 times the speed of a single pipe.) This large amount of cross-pipe data communication is a significant disadvantage for the compress/expand implementations.

5. Density Time Implementations

Thus far, vector ISAs that rely on masked operations appear to provide a number of advantages. For full performance, however, they rely on density time implementations of masked operations. Although true density-time implementations have some complications, approximate density-time implementations seem feasible. In this section, we propose and evaluate an approximate density-time implementation.

Fig. 6 shows density-time implementations of vector register read/write control and memory addressing logic. A block of logic counts the number of leading zeros in the VM; the zeros indicate elements that must be skipped to get to the next true VM value. (VM value of 1 enables a vector operation, and a 0 inhibits). The VM must be shifted by the leading zero count plus one: i.e. some variable amount between one and the maximum VL value. This is in contrast with the VL-time implementation which simply shifts VM by one every cycle and uses counters to keep track of the head and tail elements.

The leading zero count plus one is added to the current vector register head or tail pointer to arrive at the next valid vector operand. Logic is more complex for memory addressing logic. Here, the leading zero count must be multiplied by the stride before being added to the previously computed memory address. With this implementation, the logic path to compute successive addresses is much longer than in the VL-time implementation (which simply adds the constant stride to get consecutive memory addresses).

5.1. An Approximate Density-Time Implementation

The added complexity of a density-time implementation is a potential problem. To help solve this problem, a density-time approximation was developed with a much simpler implementation [16]. The approximation can skip ahead in VM, but only by powers-of-two. The implementation is shown in Fig. 7.

Fig. 7 shows the vector register control and memory addressing hardware. Here, the VM register is inspected for only leading zero counts of 1,3,7,15,..maxVL-1 (maxVL is the maximum hardware vector length.) That is, it checks for zero counts that are a power-of-two minus one. Hence, the arbitrary leading zero counter is replaced by a small number of leading zero

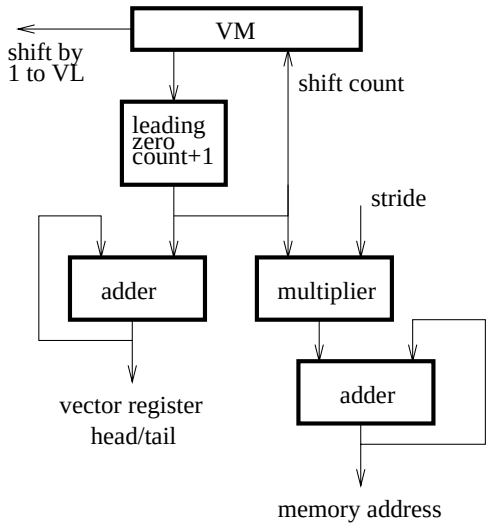

Fig. 6. Density-time implementation of of vector register control and memory addressing logic.

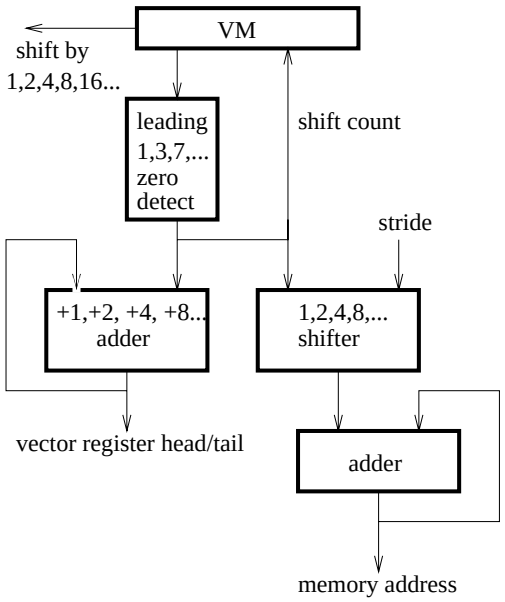

Fig. 7. Power-of-two approximations to density-time implementations of vector register control and address generation logic.

detectors (seven for length 64 vectors). Then, the appropriate power-of-two is added to the current count to get the next vector register head/tail count. Because only powers of two are being added, the adder logic is slightly simplified. Finally, the VM register is shifted only by powers of two, rather than arbitrary counts. This requires a number of gates, but few logic levels. The proposed method can take longer than actual density time when the number of leading zeros is not a power-of-two minus one. When this happens a no-op has to be performed. An analysis to follow will indicate how good the performance is.

In the memory address logic for the proposed approximation method, the major simplification is that the stride multiplier can be replaced by a shifter. As an example consider the VM:

1011 0000 1000 1000 0101 0001 1011 0000.

An actual density-time implementation takes 11 clock periods -- the number of ones in VM (assuming a single-pipe implementation). With the power-of-two approximation, the first element will be executed, then one zero will be skipped to get to the next one on the very next clock period. The third one will be done immediately. Then, there is a run of four zeros. The 3-zero detector will be active and the 7-zero detector will be inactive. Consequently three zeros will be skipped. The next element is still a zero, so there will be a no-op. This run of four zeros will take two cycles to skip. The processing of VM continues until, at the end, the last one is finished; then all remaining zeros can be skipped, and the entire vector is finished. Overall, this example vector operation takes 13 clock periods, or two clock periods more than actual density-time.

A potentially important implementation issue involves the ability of vector control logic to determine when a particular vector instruction will finish. In a VL-time implementation, it is known at instruction issue time that a vector instruction using one of the functional units will finish after VL clock periods (assuming a single pipe). In an actual density-time implementation the functional unit completion time is known from the number of ones (pop count) in VM (again, in a single-pipe implementation). However, with the power-of-two approximation, the completion time is harder to determine. It is a rather complex function of the specific VM bit pattern, and may not be known until just before the vector instruction finishes.

5.2. Performance

Because it is only an approximation to actual density time operation, we did some performance studies to see how well the power-of-two method performs. This was done by simulating the IF loop used in Fig. 1. Vectorization method 2 with masked operations was used. Results are shown in Fig. 8. This loop was simulated for VL-time, actual density-time, and approximate density-time. The approximate density time values are the averages for a number of runs where the true values in the vector mask were randomly generated. The overall performance of approximate density-time is quite close to actual density-time, and both density-time curves show a significant improvement over VL-time implementations when VM is sparse.

We conclude that the power-of-two approximation is a viable design alternative. Actually, there is a range of implementation choices between true density-time and the power-of-two approximation. For example, one could control the vector registers and functional units in true density-time, but generate memory addresses in approximate density-time.

6. Summary and Conclusions

We considered a number of vector ISA implementations for conditional operations (e.g. as in loops with IF statements). The initial study of simple IF loops led us to conclude that using masked operations was the best performing method *provided* a density-time implementation could be found.

Considering more complex IF structures also pointed out advantages of the masked operation approach because data subsets can be efficiently represented, and set operations are easily implemented. For multiple-pipe implementations, the masked operation approach introduces no cross-pipe data communication, but there may be some performance degradation if all the pipes have to wait for the pipe with the most work to be done.

The best performance of the the masked operation approach depends on a density-time implementation of masked operations. However, a true density-time implementation may be complex, especially for computing memory addresses where a multiplier is in the path. Consequently, we proposed and evaluated an approximation that only shifts the VM by powers of two. For sparse VMs, the approximation gives considerable speedup over VL-time (although actual density time would be even better). With dense VMs, the times are all roughly equivalent.

7. Acknowledgement

The authors would like to thank David Whitney for his many valuable insights and suggestions.

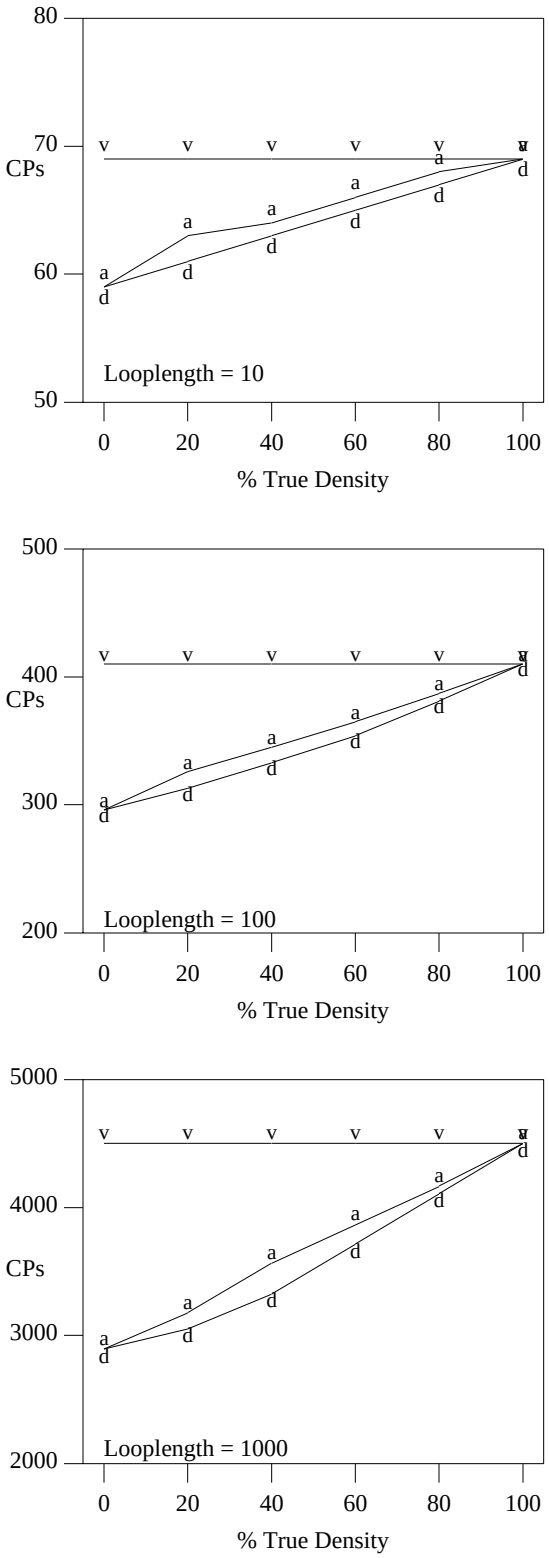

Fig. 8. Performance in clock periods for VL time (v) actual density-time (d), approximate density-time (a).

8. References

[1] C. Eoyang, R. H. Mendez, and O. Lubeck, "The Birth of the Second Generation: The Hitachi S-820/80", *Proc. Supercomputing '88*, pp. 296-303, Nov. 1988.

[2] K. Miura and K. Uchida, "FACOM vector processor system: VP-100/VP-200," *Proc. NATO Advanced Res. Workshop on High-Speed Computing*, June 1983.

[3] *Siemens Vector Processor System VP100/VP200 System Description*, 1984.

[4] T. Watanabe et al., "The Supercomputer SX System: An Overview", *Proc. Second International Conference on Supercomputing*, 51-56, 1987.

[5] *The Cray-1 Computer System*, Cray Research Inc., Publication No. 2240008B, 1976.

[6] *Cray-2 Engineering Maintenance Manual*, Cray Research, Inc., Publication No. HM-2032, 1985.

[7] *Cray X-MP Series Model 48 Mainframe Reference Manual*, Cray Research, Inc., Publication No. HR-0097, 1984.

[8] *CDC Cyber 200 Model 205 Hardware Reference Manual*, Control Data Corp, 1981.

[9] D. J. Kuck and R. A. Stokes, "The Burroughs Scientific Processor (BSP)", *IEEE Trans. on Computers*, C-31, 363-376, May 1982.

[10] Motorola AltiVec Technology website, http://www.mot.com/SPS/PowerPC/AltiVec/

[11] C. G. Lee and M. G. Stoodley, "Simple Vector Microprocessors for Multimedia Applications", *MICRO-31*, Dec. 1998, pp. 25-36.

[12] K. Asanovic, et al., "The T0 Vector Microprocessor," *Proceedings of Hot Chips VII*, pp. 187-196, Aug. 1995.

[13] J. Wawrzynek, et al., "SPERT-II, A Vector Microprocessor System," *IEEE Computer*, pp. 79-86, March 1996.

[14] R. Espasa, M. Valero, J. E. Smith, "Vector Architectures: Past, Present, and Future," *1998 International Conference on Supercomputing*, June 1998.

[15] *Cray Y-MP C90 System Programmer Reference Manual*, Pub. No. CSM-0500-000, 1992.

[16] J. E. Smith, "Density Dependent Vector Mask Operation Control, Apparatus, and Method," US Patent 5,940,625, Aug. 17, 1999.