NASA Contractor Report 181832

ICASE REPORT NO. 89-27

ICASE

IMPLEMENTING NESTED CONDITIONAL STATEMENTS IN SIMD MACHINES

David Middleton

Contract No. NAS1-18605
April 1989

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

Implementing nested conditional statements in SIMD machines

David Middleton
Institute for Computer Applications
in Science and Engineering[1]
NASA Langley Research Center

Abstract

SIMD computers consist of a very large number of processors executing a common sequence of instructions. Maintaining the full speedup potential of such machines is most sensitive to conditional execution in their programs, regions of code where some PEs perform no useful work. Techniques are presented for efficiently implementing nested conditional statements, specifically if and case statements, in SIMD machines, while adding minimal specialized hardware.

1 Introduction

An SIMD parallel computer typically provides a very large number of processing elements (PEs) at the cost of constraining them to execute a common sequence of instructions. Regions of code requiring conditional execution, such as occur in *case* and *if* statements, interfere with maintaining the full speedup potential of such machines. While conditional statements are being executed, those PEs whose data do not satisfy the predicate perform no useful work. This paper describes implementing nested *case* and *if* statements efficiently, both with respect to the number of instructions used and the amount of specialized hardware needed by the PEs.

Bruner and Reeves implement nested *if* statements in the PEs of the MPP [BR83] (distinguished by use of the *where* keyword from those performed in the central controller). Thinking Machines provide similar facilities in CM Lisp and C*. A specialised hardware

[1] This work was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-18605 while the author was in residence at ICASE.

stack for implementing nested conditional statements effectively has been suggested elsewhere [FMPC]. This work studies implementing nested conditional statements on abstract SIMD machines rather than specific ones, in order to determine the appropriate amount of hardware support.

In most SIMD computer designs, each processing element (PE) has some form of *Enable register* which controls whether the globally issued instructions may modify the data held in that PE. When an *if* statement occurs, for example, all PEs would evaluate the predicate and load the result into their Enable register. Those PEs for which the predicate is false would effectively ignore subsequent instructions until their Enable bit were reset to true.

In order to nest conditional statements, each PE must maintain a stack of these enable bits, with the topmost element indicating whether that PE is active. Section 2 describes the transformations to this stack associated with the keywords in *if* and *case* statements; Section 3 presents implementations of these transforms for various SIMD designs; Section 4 presents variations that use less PE memory.

2 The abstract stack of Enable bits

An *if* statement takes the form

if <*predicate*> **then** <*statement*> [**else** <*statement*>] **endif**,

where <*predicate*> and <*statement*> can contain other conditional statements. At the **endif**, every PE, including those that are disabled, pops its Enable stack. Consequently, every PE, including those that are disabled, must push a value on its Enable stack at the start of each *if* statement. At the **then**, every PE replaces its current Enable bit with <*current enable*>∧<*predicate result*> (the value of the predicate alone being undefined in disabled PEs). If an optional **else** is encountered, the Enable bit on the top of the stack is inverted in those PEs that were enabled immediately outside this *if* statement, as indicated by the second stack element. Figure 1 shows the transitions each keyword causes

to the different stack configurations that it can encounter. (Those configurations can arise from keywords other than the next one on the left; for example, **endif** may apply to a stack most recently changed by a **then** rather than an **else**, but the possible configurations resulting from each keyword are the same).

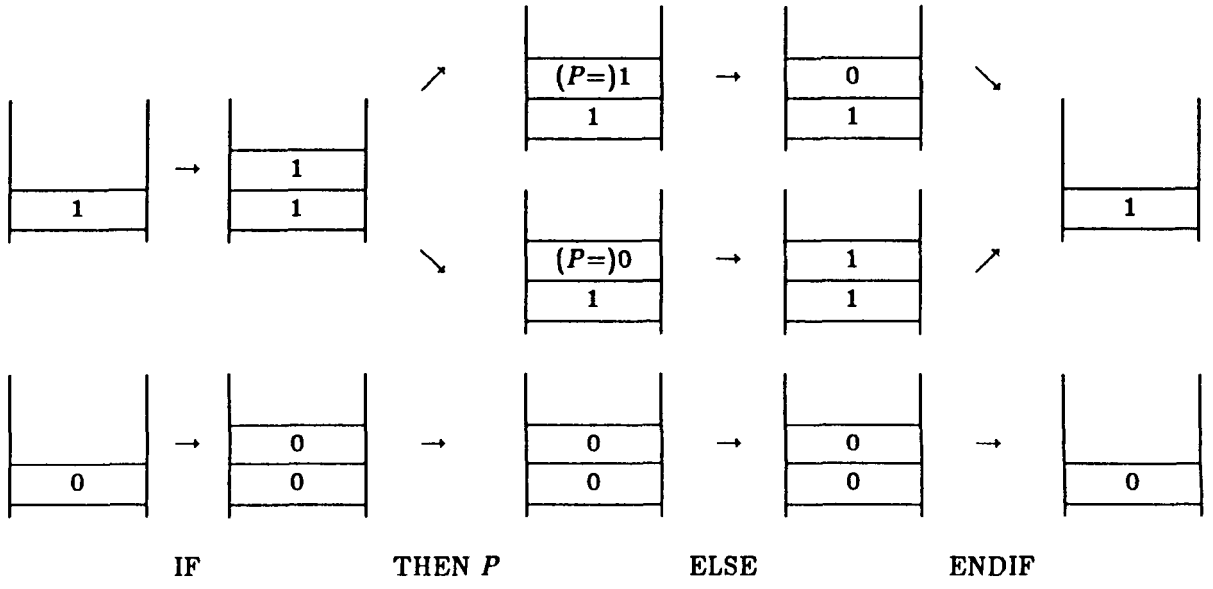

Figure 1. Enable stack transitions in an *if* statement

A common programming technique for SIMD machines is to enumerate all possible states of the PEs and to issue instructions for each case after enabling the appropriate PEs. Consequently, *case* statements are important. They take the form

caseif <*predicate*> **then** <*statement*>
[**elseif** <*predicate*> **then** <*statement*>]
⋮
[**default** <*statement*>]
endcase.

Figure 2. Enable stack for *case* statement

A *case* statement can be implemented with *if* statements, but the depth of the stack grows with the number of case branches. This growth can be avoided by exploiting the fact that the nested *if* statements all terminate together.

Within the scope of a *case* statement, the PEs may be in one of four states: (1) enabled and performing one of the branches of the *case* statement (or evaluating a predicate); (2) disabled, not yet having performed a branch; (3) disabled and having already performed a branch; and (4) disabled by a conditional statement enclosing the *case* statement. States 3 and 4 are equivalent until the *case* statement terminates and the PEs in state 3 are re-enabled. PEs in state 2 change to state 1 right before a predicate is evaluated, that is, at an **elseif** or **default**, and change back to state 2 at the **then** if the predicate is false. PEs in state 1 change to state 3 after performing their branch, that is, at an **elseif** or **default**.

A *case* statement uses two bits in the Enable stack as shown in Figure 2; the new Enable bit which distinguishes state 1 from the others, and a *Not_Yet* bit which distinguishes state 2 from states 3 and 4. Section 4 presents an alternative implemention of *case* statements that only pushes one bit onto the Enable stack.

Figure 3 shows the transitions associated with each keyword in a *case* statement, the various stack configurations being labeled with the corresponding PE state. A **caseif**

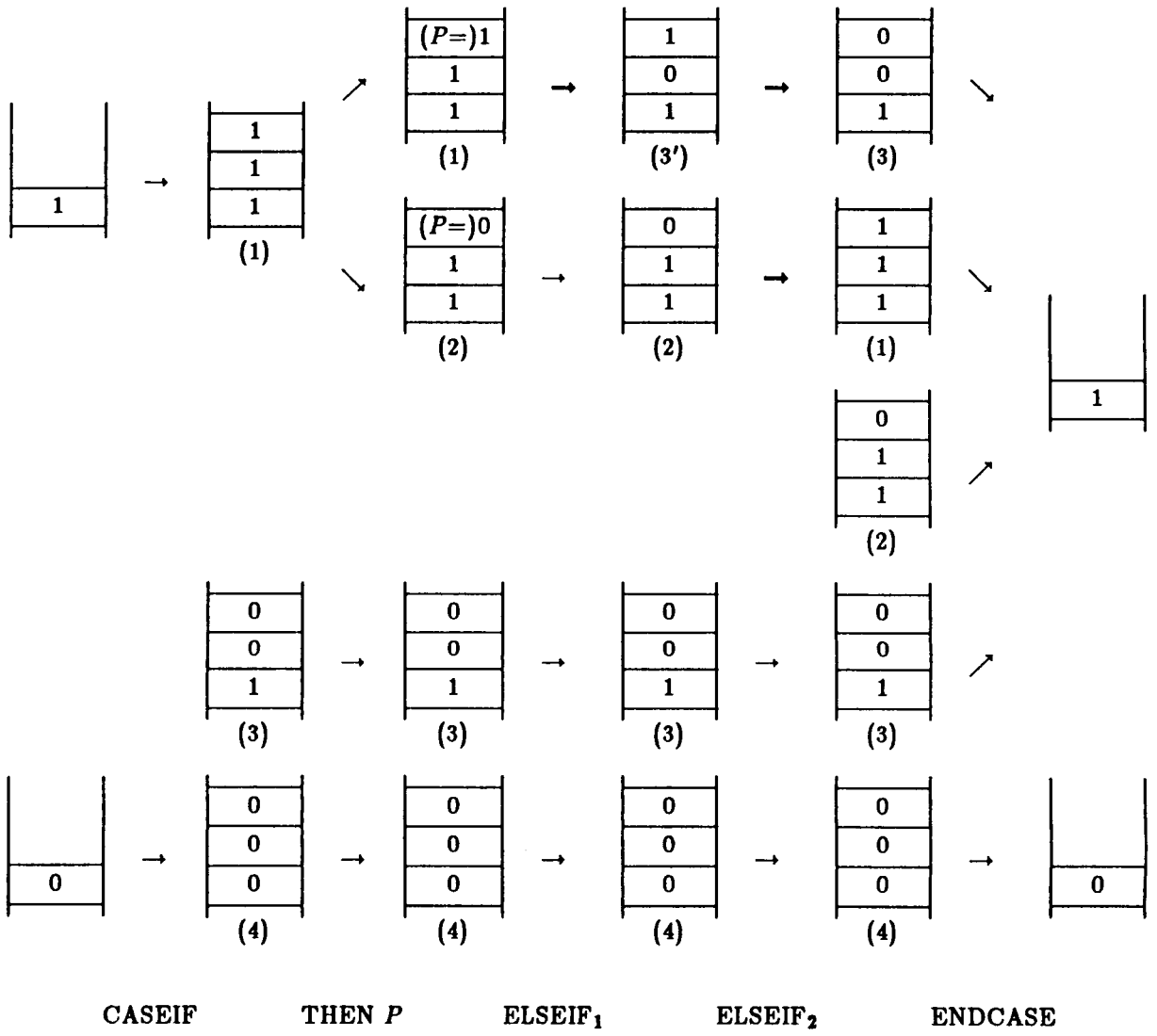

Figure 3. Enable stack transitions in a *case* statement

twice duplicates the Enable bit as it exists immediately prior to the *case* statement, and an **endcase** removes two elements to return the stack to its initial configuration. At a **then**, the predicate result replaces the Enable bit in enabled PEs, or equivalently, the *logical and* of the Enable bit and the predicate result replaces the Enable bit in all PEs. Two actions occur at an **elseif**. First, PEs in state 1, having now finished their branch of the *case* statement, change to state 3. Second, PEs in state 2 change to state 1 in

order to evaluate the subsequent predicate. These actions are accomplished by clearing the Not_Yet bit in enabled PEs, and then copying the Not_Yet bit to the Enable bit in all PEs. **Default** is logically equivalent to "**elseif** *true* **then**" which reduces to **elseif**.

3 Implementing the abstract stack in actual machines

The abstract stack described above is now implemented for realistic SIMD machines, with the twin aims of providing speed while requiring minimal additional hardware. The purpose is both to show how these conditional statements can be implemented in SIMD machines that are already designed and to suggest the appropriate amount of conditional-statement-specific hardware to be added to new designs.

We start with a very simple model of SIMD computer under the assumption that SIMD machines with less hardware would need to simulate the Enable stack anyway, and that their best approach would depend on the specific instructions available, and the related data paths among the PE registers and memory.

We do not consider the specific instructions necessary for evaluating predicates. Although the time spent evaluating predicates probably dominates the housekeeping costs associated with maintaining the Enable stack, the target applications dictate the important data types and so the common predicates[1]. The original motivation for this study was a proposal involving specialized hardware support for the Enable stack itself, rather than the general mechanisms which already support predicate evaluation.

Typical SIMD machines use a one-address instruction format: each instruction issued by the central controller specifies an operation, a single address into the local memory

[1] We also assume that the disabled PEs do not affect the evaluation of the predicate for the enabled PEs; for example, if the predicate involves communication operations or global reductions, such as the *IFANY* statement in the MPP, then the disabled PEs do not alter the result.

of each PE, and, implicitly, one or more special purpose registers in each PE's ALU[2]. One of these, the Enable register, must be set for any other register or memory location to be affected by the instruction. That is, instructions affecting the Enable register are performed in all PEs, while instructions affecting other PE state are performed only in PEs whose Enable register is set. We assume predicate results reside in another register, P, that can perform arbitrary logical operations (as in the MPP).

The topmost element of the Enable stack resides in the Enable register; the rest are stored in the local memory of a PE. The Enable stacks have the same height at all times, allowing a common stack pointer to be maintained by the central controller. Local addressing by individual PEs, as provided by the ILLIAC IV and the Connection Machine 2, provide no gains to implementing these stacks.

This first model of SIMD machine can implement the various keywords as shown below. Each keyword (with instruction count for comparison) is followed by the instructions for the PEs. Parentheses distinguish addresses into PE memory from PE register names. The expression within the parentheses is computed by the array controller concurrently with its issuing instructions and may involve side-effects such as pre-incrementing or post-decrementing the Enable stack pointer, SP. The prefixes `global` and `masked` show explicitly whether the operation occurs in all PEs or only those whose Enable register is set; they are redundant for this model since `global` is equivalent to the destination being the Enable register.

The difficulty for this model lies in saving the Enable register, which is needed to duplicate the top of the Enable stack for **if** and **caseif**, and to generate the new Enable bit from the old for **then** and **else**. Since writes do not occur in PEs whose Enable register is clear, saving the Enable register requires that the destination be cleared beforehand

[2] It seems likely that SIMD computers would follow the same evolution of other classes of computers and acquire multiple general purpose registers. The model described here reflects current SIMD designs.

(effectively using two instructions to save the Enable register, one when it is clear and a second when it is set). A program using these statements must preclear the stack at the outset (and during execution if the stack overflows).

```
if (1)      masked  (+SP)     ← Enable   ;; Push previous enable (≡ Enable reg) onto
                                         ;; PE mem. Use 1 if no path from Enable.
then (3)    masked  (SP+1)    ← P        ;; Unchanged if Enable not set.
            global  Enable    ← (SP+1)   ;; Enable ← Enable ∧ P.
            masked  (SP+1)    ← 0        ;; Clear Stack element.
else (6)    masked  (SP+1)    ← Enable   ;; Can use 1 instead of Enable.
            global  Enable    ← (SP)     ;; second stack element
            masked  P         ← ¬(SP+1)  ;; Invert top of memory stack
            masked  (SP+1)    ← P        ;; iff second stack element true.
            global  Enable    ← (SP+1)   ;; Reload Enable.
            masked  (SP+1)    ← 0        ;; Reset stack.
endif (2)   global  Enable    ← (SP)
            masked  (SP-)     ← 0
caseif (2)  masked  (+SP)     ← Enable   ;; Push Previous Enable (≡ Enable reg).
            masked  (+SP)     ← Enable   ;; Push Not Yet (≡ Enable reg).
then (3)                                 ;; As above.
elseif (2)  masked  (SP)      ← 0        ;; Clear Not Yet in memory.
            global  Enable    ← (SP)     ;; Load Not Yet into Enable.
endcase (3) global  Enable    ← (SP-2)
            masked  (SP-)     ← 0        ;; Clean up stack: clear and pop Not Yet;
            masked  (SP-)     ← 0        ;; clear and pop Current Enable location.
```

Endif and **endcase** acquire an additional one or two instructions respectively to maintain the unused stack entries at zero. **Then** requires moving the predicate result from the P register to the Enable register, but for this model, the value of the P register is undefined in disabled PEs (even if the Enable register were combined into the predicate result). **Then** takes the logical product of the Enable and P registers by performing a masked store of P into a previously cleared location (for which the stack is convenient). **Else**

conditionally inverts the Enable register according to a memory location, specifically the second element of the Enable stack.

An alternative approach to saving the Enable register under this model is to duplicate its value on the Enable stack (making that one element deeper). Such an approach might also be necessary for specific designs (like the MPP) whose Enable register is not easily accessible. Any new value computed for the Enable register is first pushed on the stack before being loaded. The actions for the keywords become

```
if (1)      masked  (+SP)    ← Enable   ;; Duplicate new stack top from Enable reg.
then (2)    masked  (SP)     ← P        ;; Update stack top in memory
            global  Enable   ← (SP)     ;; and in Enable register.
else (4)    global  Enable   ← (SP-1)   ;; second stack element
            masked  P        ← ¬(SP)    ;; Stack top in memory
            masked  (SP)     ← P        ;; only changed where second stack elt. true.
            global  Enable   ← (SP)
endif (2)   global  Enable   ← (SP-1)
            masked  (SP-)    ← 0
caseif (2)  masked  (+SP)    ← Enable   ;; Push Not Yet and Current Enable.
            masked  (+SP)    ← Enable   ;; (order reversed from previous set)
then (2)                                ;; As above.
elseif (4)  masked  (SP)     ← 0        ;; Clear stack top in memory: state 1 → 3.
                                        ;; (Was part of 'then' in previous set).
            masked  (SP-1)   ← 0        ;; Clear Not Yet in memory: state 1 → 3.
            global  Enable   ← (SP-1)   ;; Load Not Yet: state 2 → 1.
            masked  (SP)     ← Enable   ;; Update stack top in memory.
endcase (3) global  Enable   ← (SP)
            masked  (SP-)    ← 0        ;; Clear and pop Current Enable location.
            masked  (SP-)    ← 0        ;; Clear and pop Not Yet location.
```

If statements require only 9 instructions instead of 12 since there is less need to store the Enable register before changing it. However, *case* statements aquire an additional instruction per branch to maintain the duplicate top of stack.

Adding special purpose hardware to each PE can reduce this overhead for conditional statements. The second model of SIMD computer to be considered extends the first model by adding an Enable field to the instruction format, as the MPP does for many instructions. This field closely coincides with the distinction between instructions manipulating the Enable stack, which all PEs perform, and instructions performing useful work, which only some PEs perform, according to their Enable register. This extension effectively adds new instructions purely for manipulating the Enable stack; however, the hardware cost of "decoding" the Enable field is negligible.

Each PE determines whether its memory and registers (now including the Enable register) can be updated according to the logical sum of this instruction bit and the contents of the Enable register. The prefixes `global` and `masked` now refer to this Enable field of the instruction: `global` means the bit is set and the instruction takes effect in all PEs; `masked` means the bit is clear, so the instruction's affect depends on each PE's Enable register. A tick emphasizes those instructions exploiting this facility.

```
if (1)      global'  (+SP)    ← Enable   ;; copy stack top to memory
then (1)    masked'  Enable   ← P        ;; Enable ← Enable and P.
else (4)    global'  P        ← Enable   ;; Assumes paths exist between P and Enable.
            global   Enable   ← (SP-1)   ;; Load second stack element.
            masked   P        ← P̄        ;; P only changed where 2nd element true.
            global   Enable   ← P        ;; This sequence assumes P not live.
endif (1)   global   Enable   ← (SP-)
caseif (2)  global'  (+SP)    ← Enable   ;; Push Previous Enable.
            global'  (+SP)    ← Enable   ;; Push Not Yet.
then (1)                                 ;; As above.
elseif (2)  masked   (SP)     ← 0        ;; Clear Not Yet in memory.
            global   Enable   ← (SP)     ;; Load Not Yet.
endcase (1)            (SP-)             ;; concurrently in array controller.
            global   Enable   ← (SP-)
```

With this extension, there is no need to preclear the Enable stack nor to duplicate the Enable register on the memory stack. *Case* statements require 3 instructions per branch instead of the 5 or 6 required in the previous implementations. Except for **else**, the *if* statement keywords all reduce to one instruction; *if* statements require 7 instructions compared with 12 or 9 above.

Else remains unwieldy which suggests adding further special instructions. The necessary operation is $Enable \leftarrow Second \wedge \overline{Enable}$, where Second is the PE memory location holding the second Enable stack element. Since Enable implies Second, the above expression is equivalent to $Enable \leftarrow Second \oplus \overline{Enable}$ ($\oplus$ being exclusive or). The MPP exploits this to use a special facility which tests the equality of its Enable register, G, and its logic register, P. The MPP generates the value $P = G$ which is equal to $\overline{P} \oplus G$ and so can implement **else** with

```
else (2)    global' P       ← ¬(SP-1)
            global  Enable  ← P=G.
```

A better approach is to make the Enable register a general purpose boolean register, supporting all logical combinations of the Enable register with another source; this allows flexibility for implementing other conditional statements in the future. The action at an **else** becomes

```
else (1)    global' Enable  ← ¬Enable ∧ (SP),
```

and the *if* statement reduces to four instructions of overhead, one at each keyword.

4 Implementing the abstract stack densely

Some current SIMD machines are constrained by the amount of memory available in each PE. Two methods are presented for reducing the space used to implement nested conditional statements.

The first technique for saving space is to replace the stack of Not_Yet bits with a single statically allocated bit. As *case* statements nest, the stack of states in each PE follows a

particular pattern. At a **caseif**, PEs in states 2, 3, and 4, push a new state 4 on their Enable stacks. PEs in state 1 push a new state 1 which may be overwritten by states 2 or 3 during the statement's execution. Thus, every Enable stack consists of zero or more 1 states, a single state 1, 2 or 3, and zero or more 4 states. Since the Not_Yet bit is always zero in PEs in state 4 and always one in PEs in state 1, it need only exist explicitly for the Enable stack entry that is in state 1, 2 or 3.

The Not_Yet bit is only *read* at occurrences of **elseif**, to distinguish PEs in state 2 from the others; it is only *written* at a **caseif** where it is set on being allocated, or an **elseif** where it is cleared in enabled PEs. The actions associated with the *case* statement keywords are modified so that PEs in state 4 neither write nor read their Not_Yet bit, in order to leave the Not_Yet bit unchanged in any PEs having entered states 2 and 3. The operation *Enable* ← *Not_Yet*, associated with an **elseif**, is replaced with *Enable* ← *Not_Yet* ∧ *Previous Enable*. The Previous Enable is one in PEs in states 2 and 3, so their operation remains unchanged. The Previous Enable is zero in PEs in state 4, so the Not_Yet bit is irrelevant. Assuming instructions with an enable field, the actions associated with the different keywords become

```
caseif (2)   masked (Not_Yet) ← 1          ;; initialise for PEs in state 1
             global' (+SP)    ← Enable     ;; push Previous Enable on memory stack
then (1)     masked' Enable   ← P
elseif (3)   masked (Not_Yet) ← 0          ;; clear Not Yet in memory
             global  Enable   ← (Not_Yet);; load Not Yet
             masked' Enable   ← (SP-1)    ;;              ∧ Previous Enable
endcase (1)  global  Enable   ← (SP-)
```

PEs in states 2 and 3 may push state 4's on the stack; this will have no effect on the Not_Yet bit. PEs in state 1 do not use their Not_Yet bit without either setting it immediately beforehand at an **elseif**, or re-initialising it at a more deeply nested **caseif**. Thus, any changes made to the Not_Yet bit by other nested *case* statements have no affect.

The second technique for saving space replaces the Enable stack with a count of the number of disabling entries. This approach needs only logarithmic instead of linear space at the cost of taking logarithmic rather than constant time. This approach is unattractive because programs suited to SIMD machines are unlikely to use conditional statements nested deeply enough to justify the added delay of performing bit-serial arithmetic; nevertheless, the central controller need only perform addition to the necessary length, since the maximum possible count is known at all times.

A method is shown for the *case* statement; *if* statements can be implemented by interpreting them as

caseif <*predicate*> **then** <*statement*> [**default** <*statement*>] **endcase**,

which is as efficient as techniques designed specifically for the *if* and allows the two statements to be used together.

Each PE holds the following values: *Enable*, either the register or a separate memory location if the instruction set causes frequent saving to be necessary (we assume the second SIMD model in this section); *Not_Yet*, a single static memory location; and *Count*, the number of state 4's on the abstract Enable stack. An additional variable, Zero, is redundant but improves efficiency. The following values encode the different states:

State	Enable	Not Yet	Count	Zero
state 1	1	-	0	1
state 2	0	1	0	1
state 3	0	0	0	1
state 4	0	-	$n > 0$	0

Not_Yet is 1 for PEs in state 2, not having been altered since the **caseif** at this level initialised it, and 0 for PEs in state 3, not having been altered since an **elseif** at this level cleared it. Not_Yet is undefined for PEs in state 1 because a preceding, more deeply nested, *case* statement may have altered it, and is undefined for PEs in state 4, depending on whether this state occurs above a state 2 or a state 3.

The code for manipulating these values at the different keywords becomes

```
caseif (1)    masked  (Not_Yet) ← 1          ;; set if state 1; no cahnge if state 4.
    (# bits)  global' HalfAdd (Count), Enable ;; increment counter in disabled cells
       (1)    global' (Zero)    ← Enable

then (1)      masked' Enable    ← P          ;; as before.

elseif (3)    masked  (Not_Yet) ← 0          ;; clear Not Yet as before
              global  Enable    ← (Not_Yet);; Enable ←Not_Yet
              masked' Enable    ← (Zero)     ;;                ∧ Previous Enable

endcase (1)   global  Enable    ← (Zero)     ;; states 1,2 and 3 → 1.
    (# bits)  global' Decr      (Count), (Zero) ;; Pop a state 4; perhaps reset Zero.
```

The subroutine HalfAdd, executed with the enable field set in all its instructions, adds one bit to a number, in this instance, incrementing Count in disabled PEs. The subroutine Decr subtracts the *inverse* of a bit from a number, in this case, decrementing Count in disabled PEs. As a side-effect, it sets Zero when appropriate. These two operations are merged together to exploit possible arithmetic operations that may be provided by the SIMD machine. For example, the MPP provides single instructions that perform half and full adds; the test for a zero result can be interleaved with these operations as the bits pass through the arithmetic registers.

5 Conclusions

The efficient execution of nested conditional statements on SIMD machines requires little additional hardware to support the consequent Enable stack. Special purpose stack hardware is not required; the stack can be stored in the general PE memory and manipulated by standard instructions. An Enable field in the instruction helps noticeably, as does, to a lesser extent, providing general logical operations on the Enable register.

Separating the abstract Enable stack and its evolution from the actual machine facilities greatly aided designing the instruction sequences for the *if* and *case* statements. This approach should be used for implementing these statements on other SIMD designs or for implementing other conditional statements in general.

In the final set of instructions, the Enable instruction field, which is manipulated and issued by the array controller, rather than being part of the PEs, is applied to subroutine calls. This raises issues for the design of the array controller and software techniques as regards the several side-effects that a subroutine may have.

References

[BR83] J. D. Bruner and A. P. Reeves, "A Parallel P-code for Parallel Pascal and Other High Level Languages," International Conference on Parallel Programming, pp. 240-243, August 1983.

[FMPC] Second Symposium on the Frontiers of Massively Parallel Computation, Virginia, October 1988, and Computer Science TR88-048, UNC Chapel Hill.

[ ] Thinking Machines CM lisp, CM C*.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-181832 ICASE Report No. 89-27		

4. Title and Subtitle	5. Report Date
IMPLEMENTING NESTED CONDITIONAL STATEMENTS IN SIMD MACHINES	April 1989
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
David Middleton	89-27
	10. Work Unit No. 505-90-21-01

9. Performing Organization Name and Address	11. Contract or Grant No.
Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225	NAS1-18605
	13. Type of Report and Period Covered Contractor Report
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	14. Sponsoring Agency Code

15. Supplementary Notes
Langley Technical Monitor:
Richard W. Barnwell

Final Report

16. Abstract

SIMD computers consist of a very large number of processors executing a common sequence of instructions. Maintaining the full speedup potential of such machines is most sensitive to conditional execution in their programs, regions of code where some PEs perform no useful work. Techniques are presented for efficiently implementing nested conditional statements, specifically if and case statements, in SIMD machines, while adding minimal specialized hardware.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
conditional statements ; SIMD machines	60 - Comp. Oper. & Software 61 - Comp. Prog. & Hardware Unclassified - Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	16	A03

NASA FORM 1626 OCT 86