High Level VAL Constructs in a Static Data Flow Machine

by

Kenneth Wayne Todd

© 1981 by the Massachusetts Institute of Technology

June 1981

This research was supported by the Department of Energy
under the contract DE-AC02-79ER10473.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, MA 02139

High Level VAL Constructs in a Static Data Flow Machine

by

Kenneth Wayne Todd

This report was submitted as a thesis to the Department of Electrical Engineering and Computer Science on 19 February 1981 in partial fulfillment of the requirements for the Degree of Master of Science

Abstract

The Dennis-Misunas Form 1 Data Flow Machine can best be described as a static and scalar machine. Despite these two limiting characteristics, it is still possible to translate the whole of the functional programming language VAL into the base language of this machine. Methods for translating the various high level constructs of VAL are presented which exploit the parallelism inherent in programs written in VAL mainly by pipelining through a single expression (vertical parallelism) rather than employing many copies of that same expression (horizontal parallelism), although the latter is not ruled out. These methods are tested by translating two different versions of a vector dot product algorithm, and the results obtained from running these translations on an interpreter are analyzed.

Thesis Supervisor: Jack B. Dennis

Title: Professor of Electrical Engineering and Computer Science

Key words: VAL, static data flow machine, instruction cells, pipelining, sharing, streams.

Acknowledgments

I would like to thank my father who encouraged me not to quit, and my thesis advisor who would not let me.

Table of Contents

Index to Figures **5**

1. Introduction **6**

1.1. The VAL Programming Language *6*
1.2. The Static Data Flow Computer *10*
1.3. Instruction Cells *12*

2. The Translation of VAL into Instruction Cells **17**

2.1. Simple Expressions *17*
2.2. The Let-in Construct *19*
2.3. The If-then-else Construct *20*
2.4. Functions *23*
2.5. The Forall-eval Construct *29*
2.6. Arrays *34*
2.7. Records *39*
2.8. Unions/The Tagcase Construct *41*
2.9. The Forall-construct Construct *43*
2.10. The Iter Construct *48*
2.11. A Few Notes About Pipelining *57*

3. An Example **59**

4. Conclusion **64**

Appendix 1. Instruction Cell Opcodes **65**

Appendix 2. Array Operations Implemented Using Forall-Construct **70**

References **74**

Index to Figures

Figure 1.1. The Dennis-Misunas Form 1 Data Flow Machine. 11
Figure 1.2. Blow-up of an Instruction Cell. 14
Figure 1.3. An IADD Instruction and its Operands. 14
Figure 1.4. Instruction Cell Abbreviations Used in this Thesis. 15
Figure 2.1. The **let** Construct. 20
Figure 2.2. The **if** Construct. 22
Figure 2.3. Calling the Function. 24
Figure 2.4. Arbitrating the Function Calls. 25
Figure 2.5. Pipelining the Return Address Around the Function Body. 26
Figure 2.6. Storing the Return Address of the Function Caller. 27
Figure 2.7. Returning the Results of the Function. 28
Figure 2.8. Testing for Bad Low and High Index Values in the **forall eval**. 31
Figure 2.9. Pipelining Through The Element Expression of the **forall eval**. 32
Figure 2.10. Producing the Final Result of the **forall eval**. 33
Figure 2.11. Binary Tree of *OP Instructions. 33
Figure 2.12. Snapshot of the Heap. 36
Figure 2.13. Operations on the Heap. 37
Figure 2.14. The Tagcase. 42
Figure 2.15. Setting Up the **forall construct**. 45
Figure 2.16. Stream Generation for the Element Expression in the **forall construct**. 46
Figure 2.17. Initializing the Elements of the Array Created by the **forall construct**. 47
Figure 2.18. The **iter** Construct. 49
Figure 2.19. A Pipelined Implementation of the **iter** Construct. 51
Figure 2.20. Storing the Iteration Result in the Buffers. 52
Figure 2.21. The Buffer Head Generator. 53
Figure 2.22. The Buffers for the **iter** Construct. 54
Figure 2.23. The **if** Construct as an Iter-End. 54
Figure 2.24. The **if** Construct as an Iter-End (cont.). 55
Figure 2.25. A Redefinition Arm in the **iter** Construct. 56
Figure 2.26. A Terminating Arm in the **iter** Construct. 56

1. Introduction

1.1 The VAL Programming Language

Traditional programming languages such as FORTRAN and ALGOL reflect the concept of store which is present in von Neumann computers. These languages perform their computations by manipulating and changing variables located in this store. Such changes are called *side effects.* It is because of these side effects that the flow of data is so difficult (if not impossible) to trace, and without knowledge of this flow, it is also a difficult task to determine parts of programs which are data independent and can thus execute concurrently.

Functional programming languages, on the other hand, lack this concept of a global storage area that can be changed at will. These languages still make use of a storage area of some sort, but this area is kept hidden from the programmer. Variables no longer exist in their traditional sense as objects whose values vary but instead as *value names, i.e.*, names denoting values. When a value name is declared, it is assigned a value which it retains for the duration of its life. This is known as the *single assignment rule.*

The programming language VAL [1] (*Value-Oriented Algorithmic Language*) is one such functional language. It is a high-level language designed particularly for expressing algorithms on computers that are capable of achieving parallel execution, especially those machines based on data flow architecture. The current version of VAL leans mainly towards numerical applications, but future versions of the language will address other areas of application.

Since it is a functional language, the basic programming unit of VAL is the function. Each function is passed a set of one or more argument values and computes a set of one or more values from these argument values which are returned to the caller. The function is granted access to only

those values that are passed to it as arguments, meaning that there are no "global variables". In addition, changing the value of these arguments is prohibited since VAL is side effect free.

There are no explicit input/output facilities in VAL since such would cause side effects. Instead, the VAL user calls the desired function by supplying values for its arguments. The function then begins to execute, possibly calling other functions in the process, and terminates upon the completion of computing the resultant values. These values are then returned to the user and most likely are displayed on the user's terminal.

The body of a function consists of various subexpressions which are combined in some mathematical sense like addition or subtraction. Besides function calls, these subexpressions are built from the **let**, **if**, **forall**, **tagcase**, and **for** constructs, and like function calls, each is functional in the sense that each produces a set of values from the argument values and value names defined locally within the function. The **let** allows for the declaration and definition of one or more value names that are to appear within the scope of an expression. The **if** construct returns the value of one of n expressions, depending on the values of $n-1$ Boolean expressions. The **forall** is used to generate one or more sets of values of uniform type within each set and to return them either as arrays in the **construct** case or to return the results of applying some operation on each set in the **eval** case. The **tagcase** uses the tag of a **oneof** value to determine which one of several expressions is evaluated and returned. The **for** is used to perform iteration, where the results of one iteration cycle depend on the results of the previous cycle.

As an example of the **let** and **if** constructs, the following VAL function is given:

```
function quadratic_formula(A, B, C: real returns real, real)
   let determinant: real := B * B - 4.0 * A * C
   in
      if determinant < 0.0 then undef[real], undef[real]
      else (-B + SQRT(determinant)) / (2.0 * A),
         (-B - SQRT(determinant)) / (2.0 * A)
      endif
   endlet
endfun
```

The let declares the value name *determinant* to be of type real and defines its value as the determinant of the binomial Ax^2+Bx+C. This value name is then used within the body of the expression proceeded by the "in" of the let, which is an if construct. If the determinant is less than zero, then the values returned by the if construct are **undef[real], undef[real]**; otherwise, the values returned are the roots of the binomial. These values are then returned to the let construct, which in turn are returned by the function.

The following function is an example of the **for** construct:

```
function factorial(N: integer returns integer)
   for
      i: integer := N;
      product: integer := 1
   do
      if i <= 1 then product
      else
         iter
            product := product * i;
            i := i - 1
         enditer
      endif
   endfor
endfun
```

This expression calculates the factorial of *N*. Both *i* and *product* are *loop names* whose values may change with each iteration cycle. Initially, *i* has the value of *N* and *product* has the value of 1. At the beginning of each cycle, *i* is tested to see if its value is less than 1. If so then the iteration terminates and the value of the **for** and hence the value returned by the function is that of *product*; otherwise, as

specified by the "iter", the value of *product* is redefined to be that of the current value of *product* times the current value of *i*, the value of *i* is redefined to be one less than its current value, and another cycle of the iteration is performed.

The factorial can also be calculated using the **forall** as follows:

```
function factorial(N: integer returns integer)
    forall i in [1, N]
    eval times i
    endall
endfun
```

This **forall** generates a set of values {1, 2, . . . , *N*} as specified by the expression "*i*" that are all multiplied together as specified by the "**times**". This value is then returned by the **forall** to the function, which returns this value to the caller.

The last example that is given now is of the **tagcase**. Let *X* be of type

```
oneof[A: integer; B: real; C: boolean; D: boolean]
```

This function computes some nonsensical value depending on the tag of *X*:

```
function tagcase_example(
    X: oneof[
        A: integer;
        B: real;
        C: boolean;
        D: boolean]
    returns real)
    tagcase S := X
        tag A: real(S) / 3.0
        tag B: S / 2.0
        otherwise: 0.0
    endtag
endfun
```

If *X* has tag *A* and value 6 then the value of the **tagcase** is 2.0. If *X* has tag *B* and value 7. then the value of the **tagcase** is 3.5. If the tag of *X* is either *C* or *D*, then 0.0 is the value of the **tagcase**. Once the value of the tagcase has been computed, it is returned to the function, which returns it to the

caller.

1.2 The Static Data Flow Computer

The computer that this thesis will attempt to translate VAL for is known as the Dennis-Misunas Form 1 Data Flow Machine [5, 6], which is shown in Figure 1.1. The machine consists of *N* processing units, *M* memory modules, and two communication networks. From the *instruction cell memory modules* flow operation packets, each containing an opcode, its operands, and the addresses of where to send the results. These enter the *arbitration network*, whose job it is to route each packet to one of the *processing units*. The processing units receive these operation packets from the arbitration network, perform the specified operations on the given operands, and create result packets consisting of either the result or an acknowledge signal and the destination address. The *distribution network* receives these result packets and uses the destination address of each to send the result or acknowledgment to the proper location in the instruction cell memory modules.

There are two things to note about this machine. First, it is a totally *scalar* machine in that it operates solely on scalar values: Booleans, integers, reals, and characters. It has no built-in facilities for handling array, record, and union types in that it lacks a structure memory and processor. Second, it is a *static* machine, meaning that the generation of instruction cells that make up a program is completed before the cells are loaded into the memory modules and execution commences. There is no runtime generation of cells.

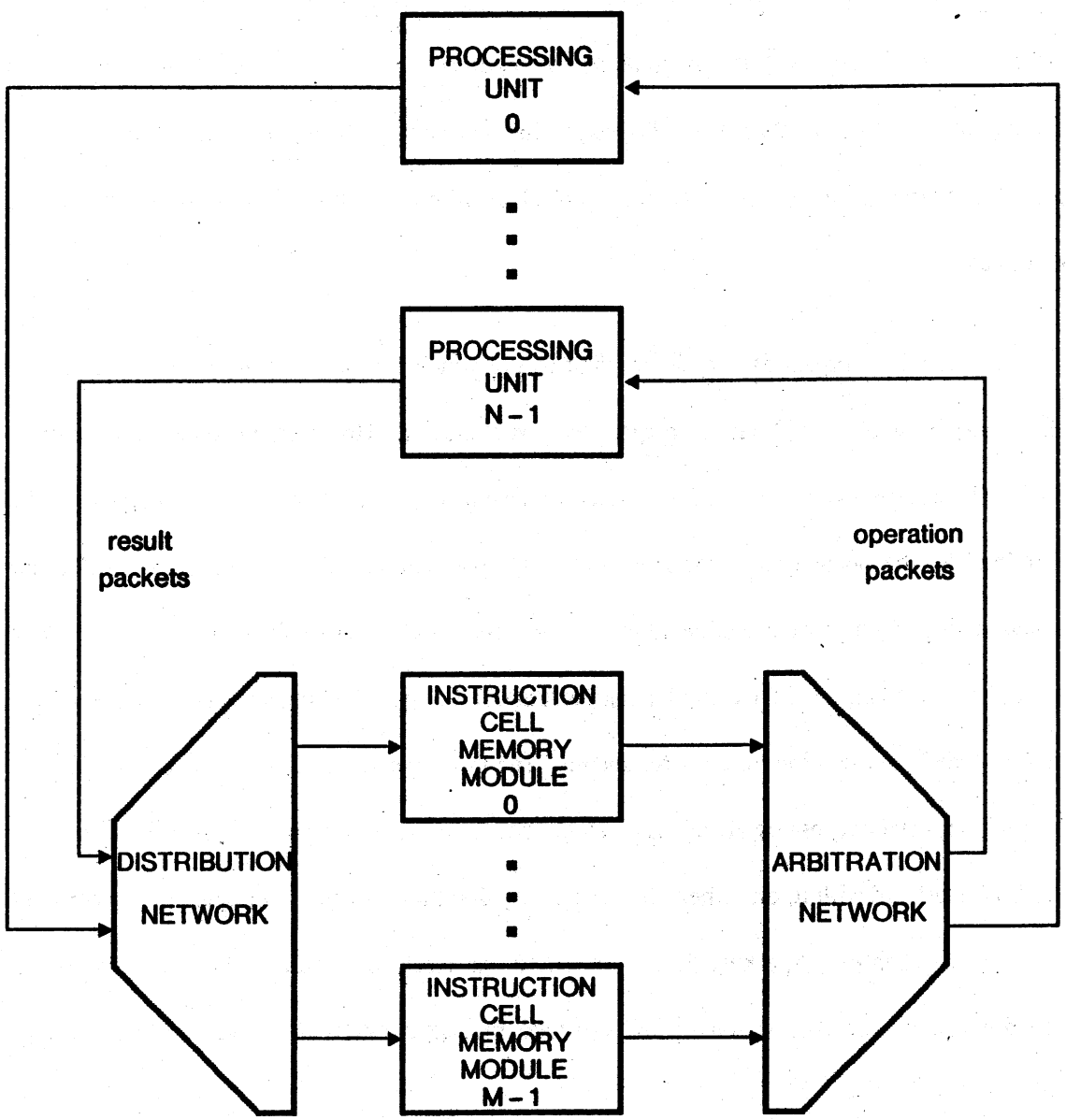

Figure 1.1. The Dennis-Misunas Form 1 Data Flow Machine.

1.3 Instruction Cells

Each instruction cell module contains a number of *instruction cells*. A cell consists of an *opcode*, places to hold the actual values of its *operands*, *destination addresses* that specify where to send the result of applying the opcode to the operands, and some control and accounting information. Appendix 1 lists the set of opcodes that will be used throughout this thesis. This is a basic set, performing elementary scalar operations along with instructions for controlling the flow of data values.

There are three operands per cell, the first two of which are long ones which can contain any scalar value, be it of type Boolean, integer, real, or character. However, no type information is retained about these operands: such is either irrelevant as in the ID instruction or types can be determined by the opcode as in the ADD and IADD instructions. This allows for a smaller size instruction set and a high degree of flexibility in using the heap (to be explained later on). Since VAL is a strongly-typed language, the compiler can perform all the type checking necessary; thus, any type checking done by the machine itself is redundant. The third operand is a short one of type Boolean and serves a special purpose by giving each cell the power to gate its outputs. Each operand has two status bits associated with it, one when clear indicating that the corresponding operand is a constant and the other when clear marking that the operand has received its value. For simplicity, if an operand is either a constant or unused by its opcode, then its constant and received bits are always clear.

The destination addresses are simply the numbers of those instruction cell that are to receive something from the execution of the cell, and the control information determines what that something is, if it is to be anything at all. For each destination address, there is a four bit *use field* that is associated with it. Two bits of this field are used to determine whether the destination cell is to

receive an acknowledgment or the result, and if it is to get the result, then to which operand to send it to. The other half of the use field is used in conjunction with the third operand to provide the gating power. One bit, the *T-bit*, when set indicates that the destination cell is to receive whatever has been preordained if the third operand is **true**. The other bit, the *F-bit*, works in the same fashion but for the value of **false**. Thus, if only the T-bit of a use field is set then the behavior of the cell toward the corresponding destination is like that of a T-gate. Likewise, setting only the F-bit yields an F-gate. If both bits are set then the result or acknowledgment is sent unconditionally, and if neither are set then the destination address is unused.

Another part of the control information contains the number of acknowledge signals that the cell has yet to receive before it can execute. This number is set after the cell fires to the number of cells that are sent the result of execution. As each receiving cell executes, it sends an acknowledge signal back to the source cell. With each reception of an acknowledgment, the source cell decrements this number by one. When it finally turns zero, the cell is assured that all of the receiving cells have consumed the result and are ready for another one. The cell can now execute again, pending the arrival of all of its operands. (The MERGE and SER instructions deviate from this procedure slightly and will be explained later.) Thus, the acknowledge signals prevent a cell from producing results faster than its receivers can handle them.

The final part of the control information concerns the resetting process of the cell after the operation packet has been created and sent on its way. The values of the constant bits are copied into the corresponding received bits, and the value for the acknowledgments needed is reset to one of two values, depending on whether the value of the third operand is **true** or **false**.

Figure 1.2 shows an enlargement of the notation that will be used to represent the cells appearing in the diagrams throughout this thesis. Figure 1.3 shows an example of such, whose

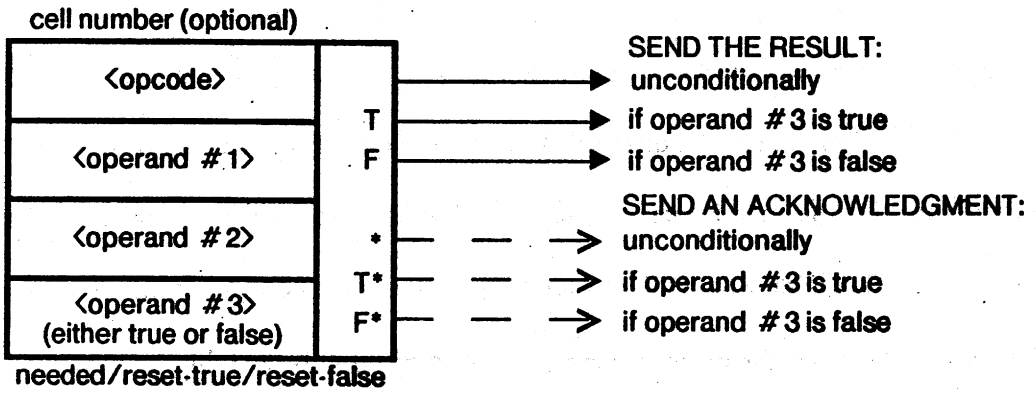

Figure 1.2. Blow-up of an Instruction Cell.

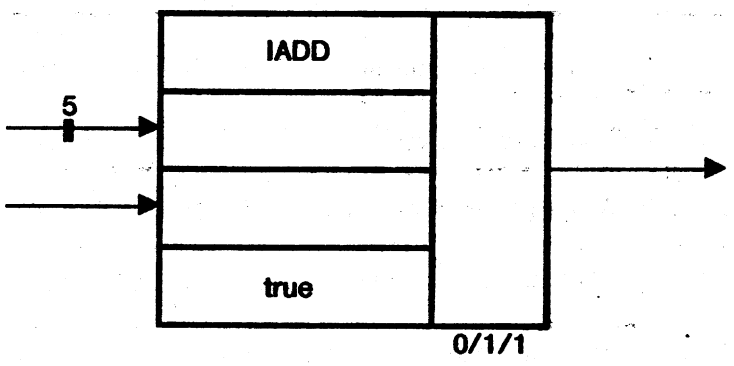

Figure 1.3. An IADD Instruction and its Operands.

opcode is IADD, which has received the value 5 for its first operand, which has yet to receive the value of its second operand, and whose third operand is a constant **true**. Its acknowledgments needed value is initially zero, and its acknowledgment reset values for both values **true** and **false** of the third operand are one.

Figure 1.4 shows five abbreviations that will be used in the diagrams to help keep them simple. The first one replaces a result and acknowledge line with a a single thicker line. The second one replaces many identical cells with a single one and the third is its acknowledge line shorthand corresponding to the first abbreviation. The fourth is used for those cells that do not need any gating

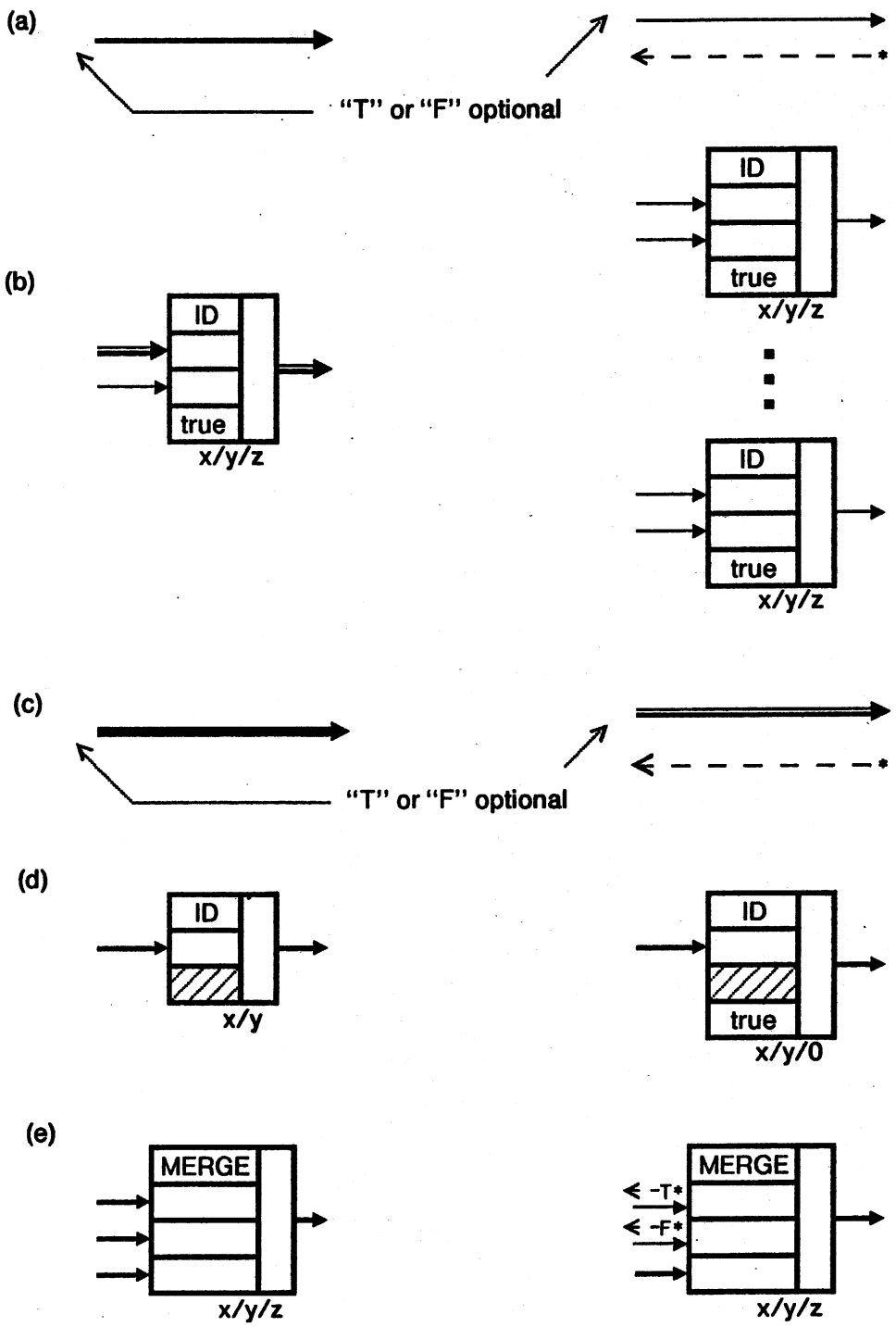

Figure 1.4. Instruction Cell Abbreviations Used in this Thesis.

capacity and thus do not care about the values of their third operands. Notice also how an unused operand is denoted by the shading. The last one is used in conjunction with the MERGE and SER instructions and will make sense once these two operations have been explained.

2. The Translation of VAL into Instruction Cells

2.1 Simple Expressions

The simplest of expressions are those that use only operators for which there are instruction that perform the specified operations. For example, if *A* and *B* are both expressions of arity one (1-tuples of values), then the following are all simple expressions, providing that the operations being performed are defined for the types of *A* and *B*:

A	min(*A*, *B*)	is pos_over(*A*)
~*A*	max(*A*, *B*)	is neg_over(*A*)
A | *B*	*A* < *B*	is over(*A*)
A & *B*	*A* <= *B*	is pos_under(*A*)
−*A*	*A* > *B*	is neg_under(*A*)
abs(*A*)	*A* >= *B*	is under(*A*)
A + *B*	*A* = *B*	is unknown(*A*)
A − *B*	*A* ~= *B*	is zero_divide(*A*)
A * *B*	integer(*A*)	is arith_error(*A*)
A / *B*	real(*A*)	is miss_elt(*A*)
mod(*A*, *B*)	character(*A*)	is undef(*A*)
		is error(*A*)

For each of these expressions, there exists a single instruction cell type that can perform the desired operation. With two exceptions, the translation is straight forward: the value of *A* is sent to the first operand of the cell and the value of *B*, if required, is sent to the second operand. In the case of the expression "*A*", no cell is needed since the value of this expression has already been obtained, *i.e.*, its value is simply the value of *A*. As for the case of "−*A*", *A*'s value is sent to the second operand of either a SUB or ISUB instruction depending on *A*'s type, and the first operand is set at a constant zero.

A slight problem turns up in that many of the above binary operators are also well defined for an arbitrary number of operands. The compiler handles this situation by building a balanced binary tree out of instruction cells of the proper type. The operands are grouped in pairs and the compiler

generates instruction cells to perform the operation on each of the pairs to produce a set of partial results, the number of which is at most half one plus the number of operands. It then recursively performs this process of grouping and cell generating with the partial results until just one value is left, *i.e.*, the final result. For example, the expression

$$A + B + C + D + E + F$$

is translated as

$$((A + B) + (C + D)) + (E + F)$$

and not as

$$((((A + B) + C) + D) + E) + F$$

A problem still remains in that the "+" and "−" operators have the same priority and can appear in expressions together without any parenthesization, where each "−" indicates the addition of the negated quantity to the right of the operator. For this situation, the compiler performs the same recursive process of grouping and cell generation as before, except that when a group is preceded by a minus sign, the operation performed on the two elements of the group is the opposite of the one indicated, *i.e.*, if an addition is supposed to occur, a subtraction takes place instead, and *vice versa*. In case the first operand is preceded by a unary minus, it is initially subtracted from zero. For example, for the expression

$$-A + B - C - D + E - F - G + H - J - K$$

the compiler performs the following transformation steps:

$$(0 - A) + (B - C) - (D - E) - (F + G) + (H - J) - K$$

$$((0 - A) + (B - C)) - ((D - E) + (F + G)) + ((H - J) - K)$$

$$(((0 - A) + (B - C)) - ((D - E) + (F + G))) + ((H - J) - K)$$

The same rules for the "+" and "−" operators apply correspondingly to the "*" and "/" operators and the "=" and "~=" operators as well.

2.2 The Let-in Construct

The let is the simplest of the high-level constructs included in VAL. In its general form it looks like this:

```
let Z := let_definitions(X)
in in_expression(Y, Z)
endlet
```

Here, X, Y, and Z are vectors of scalar values, possibly of dimension zero. The values of Z's elements are computed as some functional combination of the values of the elements of vector X. Vector Z is local to the let construct, and its value is defined only within the in expression. Once the value of Z has been derived, it is used along with the value of vector Y to determine the value of the let construct (another vector), as specified by the in expression. Note that the vectors X and Y need not be disjoint.

The let contributes nothing to the completeness of VAL. Indeed, the above construct could simply be replaced by the following with the end result being the same:

```
in_expression(Y, let_definitions(X))
```

Hence, the purpose of the let construct as far as the programmer is concerned is to declare and define temporary value names to appear in a given expression in order to help promote the clarity and readability of the program. For the data flow machine, however, the main function of the let construct is to evaluate expressions only once that occur repeatedly within a scope.

The translation into instruction cells of the let construct is shown in Figure 2.1. The boxes labeled "let definitions" and "in expression" are used to stand for instruction cells linked together in some fashion to produced the desired results from the given inputs. The two numbers underneath the boxes are their acknowledgments needed and reset-true values. Since it is unknown in the context of the figure how many places use the values of Y, Z, and the let construct, the reset values

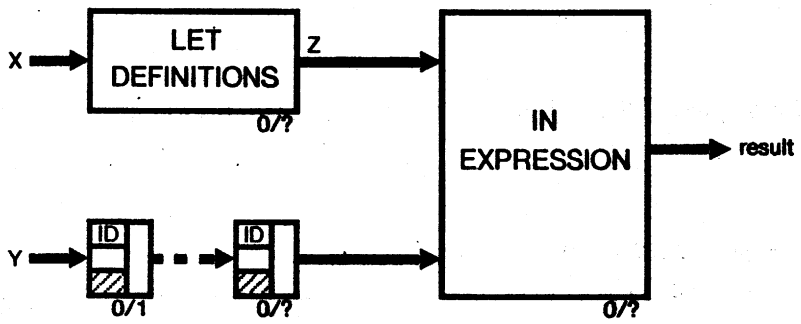

Figure 2.1. The let Construct.

---

are left as question marks. In reality, the compiler needs only to keep a reference count to determine these reset values. Note also that *Y* is sent through a series of ID instruction cells. This is to turn the construct into a pipeline. Should there be no need or desire for pipelining, *Y* can be fed directly into the in expression.

To inform the compiler whether or not to make this and other constructs pipelinable, there is envisioned to exist an *advice file* for the program. This file would be headed by the words "optimize throughput" if pipelining is desired, and either "not optimize throughput" or "optimize space" if no generation of ID buffers is wanted.

2.3 The If-then-else Construct

The general form of the if construct appears below.

if *predicate*(*P*)
then *true_expression*(*X*, *Y*)
else *false_expression*(*Y*, *Z*)
endif

Once again, *P*, *X*, *Y*, and *Z* are all vectors of scalar values, with *X*, *Y*, and *Z* all being disjoint.

The semantics of this construct at first appear simple: evaluate the predicate; if it results in the

value **true**, then the value of the construct is the value of the true expression and if its value is **false** then the false expression's value is the construct's value. However, VAL has expanded its set of Boolean values to include **undef[boolean]** and **miss_elt[boolean]**. The construct must now be defined to produce an intelligent result when the predicate evaluates to one of these error values. The following **let** construct tells the story:

```
let pred: boolean := predicate(P)
in
    if is error(pred) then undef[*]
    elseif pred then true_expression(X, Y)
    else false_expression(Y, Z)
    endif
endlet
```

The problem of bad Boolean values is now solved since the test filters out all error values and causes the value of the **if** construct to be **undef** of the proper types and arity (denoted by the "*" in "**undef[*]**").

There are two basic ways to translate a conventional if-then-else into instruction cells. In the first, the data flow machine evaluates the predicate, true expression, and false expression in parallel and then uses the result of the predicate to choose whether the value of the true or false expression will be gated to the outside world. This has the advantage of achieving a high degree of parallelism, but at the expense of performing needless computation in that both the true and false expressions are both evaluated but only the results of one are used. This is not something to be overly concerned about, unless the thrown away computation is particularly long, the worst case of which being when it turns out to be infinite.

The other way to transform the if-then-else is given in [3] and [7]. The predicate is evaluated first and then its value is used to gate the inputs to the true and false expressions, letting only the proper set of inputs through to the selected expression. This way avoids needless computation but at

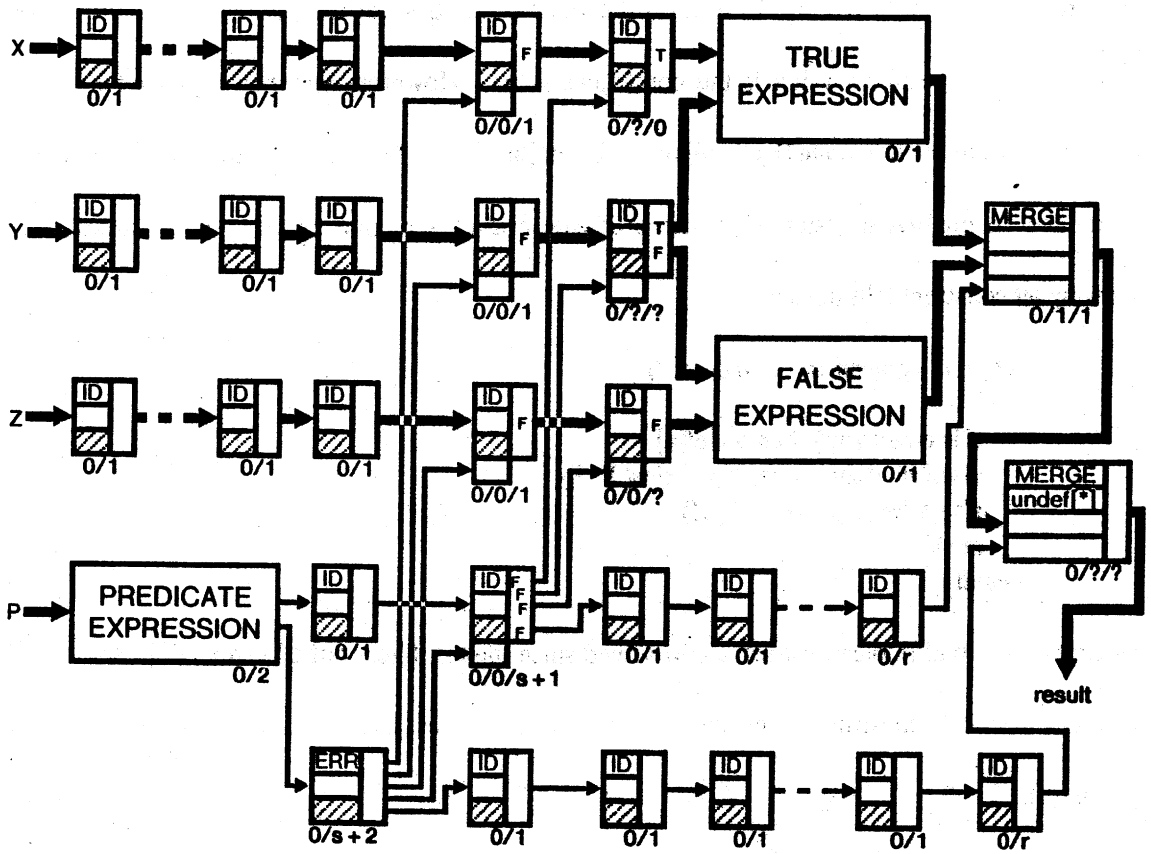

Figure 2.2. The if Construct.

---

the price of not being as parallel as the first one. Using this method, Figure 2.2 shows how the VAL if construct can be translated into instruction cells.

There are a few things to note about this diagram. First, the value of "s" is the sum of the sizes of *X*, *Y*, and *Z* and "r" is the arity of the result. Second, like the let construct, identity operators have been added to achieve maximum throughput when pipelining. Because it is pipelined, the MERGE instruction cell has been introduced. Before it can execute, its acknowledgments needed value must be zero, its third operand must have been received, and if the value of that operand is true then the first operand must have been received, and if its value is false, then the second operand must be present. The result of its execution is simply the value of the first operand if the third operand has

the value of **true**, or the value of the second operand if the third is **false**. The resetting processing of the cell is just like all the others, except that the value and received status of the one operand that went unused remains untouched. This is all done so as to allow simultaneous execution of both the true and false expressions by different activations of the **if** construct while preserving the FIFO property of the construct. For example, if an activation of the **if** construct which takes the false branch is followed by an activation which takes the true branch, and the second activation produces its results before the first does, then the MERGE will prevent the results form the second one from being put out by the **if** before the results from the first one have been released.

For some predicates, particularly the error tests, it can be determined at compile time that they will never yield error values as results. For such cases, the error test is not needed and the conventional if-then-else translation is adequate.

As a closing remark, an expression with the form

if *A* **then** *B* **elseif** *C* **then** *D* **else** *E* **endif**

is just a recursive version of the **if** construct above. It is translated as

if *A* **then** *B* **else (if** *C* **then** *D* **else** *E* **endif) endif**

2.4 Functions

There are two basic ways to handle functions in the Form 1 data flow computer. The first is the simplest and most straight forward of the two: at each invocation site of the function, the compiler textually replaces the function call by the body of the function. The other is significantly more complicated: only one body of the function is used which is shared among all calling sites. The compiler generates the appropriate instruction cells so as to arbitrate between calls to the function from the different activation sites.

Clearly, the first method yields the better runtime of the two and is the best method if a function is called from only one point or if it is small in size. On the other hand, for larger functions with multiple activation sites, the second method utilizes the resources of the instruction cell memory modules more efficiently. This section presents the translation of this second method which is based upon work presented in [9].

Figure 2.3 shows how the function call is set up and viewed by the caller. The function takes n arguments and returns m values. The value of the i^{th} argument is the result of the execution of cell number $X-i$, and the value of the i^{th} returned value is stored in the first operand of cell number $X+i$. When the function is ready to be called, *i.e.*, when cells $X-1$ through $X-n$ lack only an acknowledgment to commence execution, the caller sends an acknowledgment to cell X, which causes this cell to fire. The result from executing cell X, which is also the integer X, is sent to a binary tree of SER instructions so that requests to call the function from different calling sites can be arbitrated. See Figure 2.4.

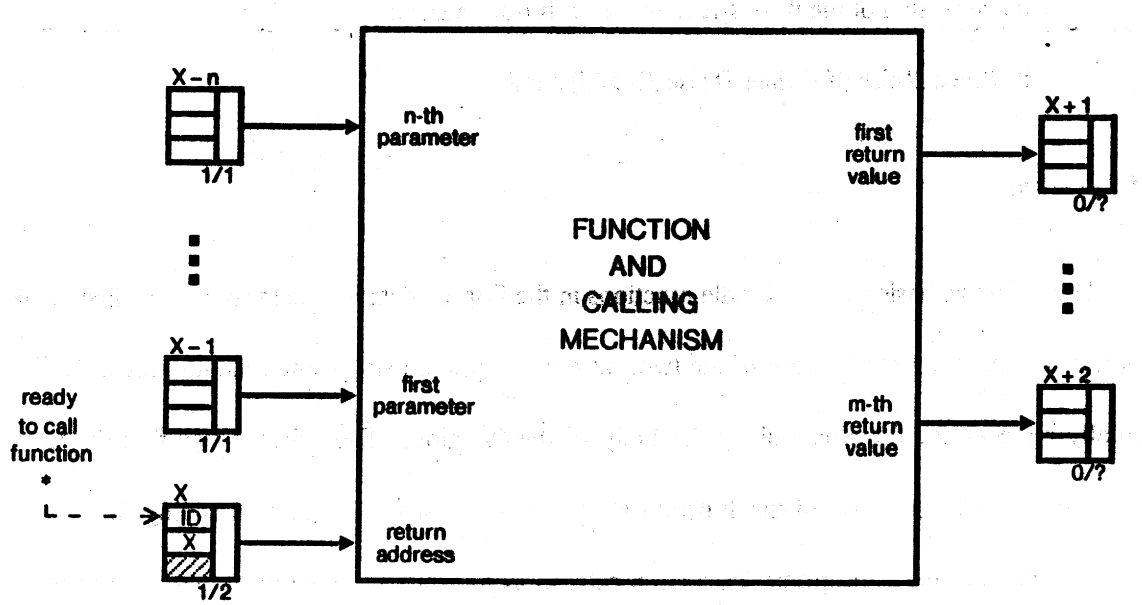

Figure 2.3. Calling the Function.

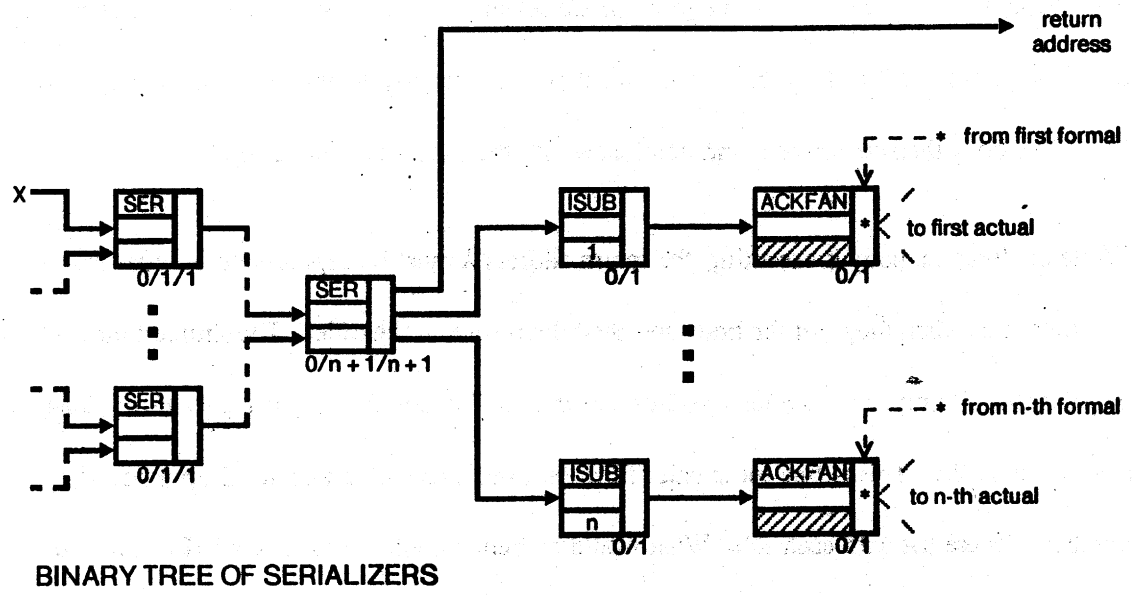

Figure 2.4. Arbitrating the Function Calls.

At this point, an explanation of how the SER instruction functions is needed. Before it can fire, its acknowledgments needed value must be zero and at least one of the first two operands must have been received; the third operand is always received. When it executes, if only one of the first two operands is present then the result is the value of that operand; otherwise, the result is the value of the first operand if the third operand's value is **false**, or the second operand if the third operand is **true**. The value of the third operand is then set to **true** if it is the first operand that is consumed, or to **false** if the second operand is used. Resetting the acknowledgments needed value and the operand that is used is performed in a fashion just like the MERGE. This method of arbitration prevents the monopolization of the function body by a single caller that might otherwise occur in using the SELECT instruction of [9] instead of SER.

The request *X* to use the function body filters through the binary tree of SER instructions until it finally reaches the root. It is then sent to a series of ISUB instructions, whose results are sent to ACKFANs. Each ACKFAN causes an acknowledgment to be sent to the cell number given as the value

of its first operand. Thus, cells $X-1$ through $X-n$ are acknowledged, causing each of these cells to execute. The values produced by these cells are then sent directly to the function body. The execution of the body then commences and terminates with the production of the results.

While the function body is executing, the return address X must be kept around in order that it can meet the results when they exit the body and send them back to the caller. The simplest method to accomplish this is to pipeline the return address around the function body by using ID instructions, as shown in Figure 2.5. This is fine for smaller functions, but larger functions wind up executing a large number of these IDs with each call. What would be better would be to have a FIFO queue by which the return addresses could temporarily be stored. Figure 2.6 shows how such a queue can be implemented. The queue itself is a series of N consecutive ID instruction cells, starting at cell number M and ending at $M+N-1$. The store phase takes the first return address it receives and puts it in the

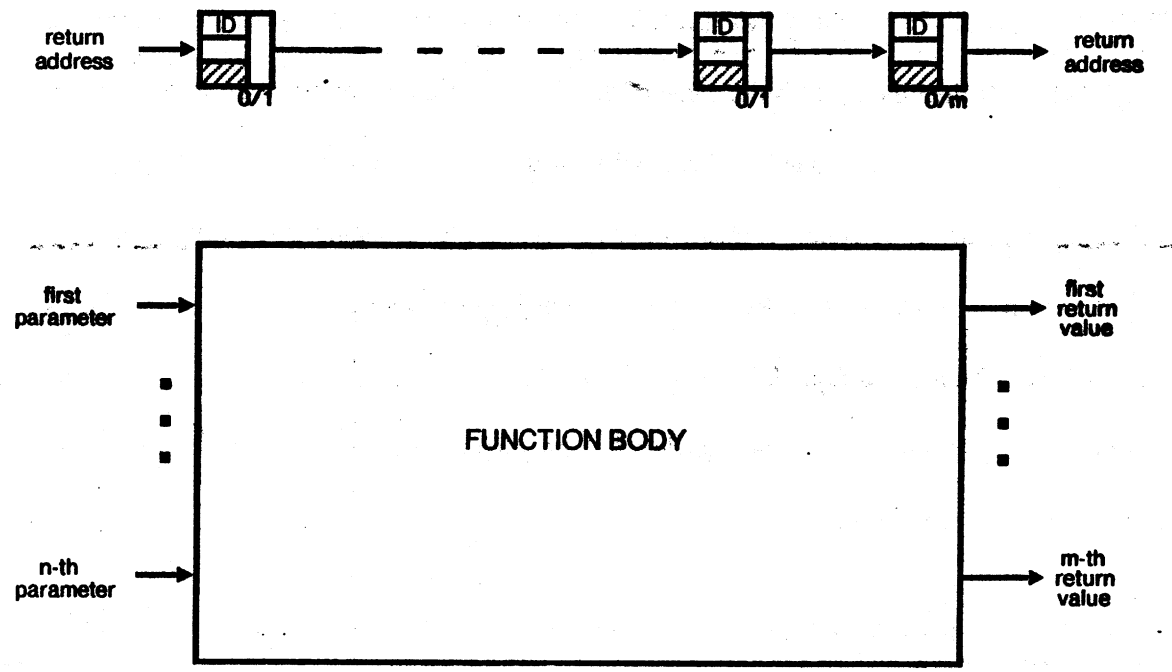

Figure 2.5. Pipelining the Return Address Around the Function Body.

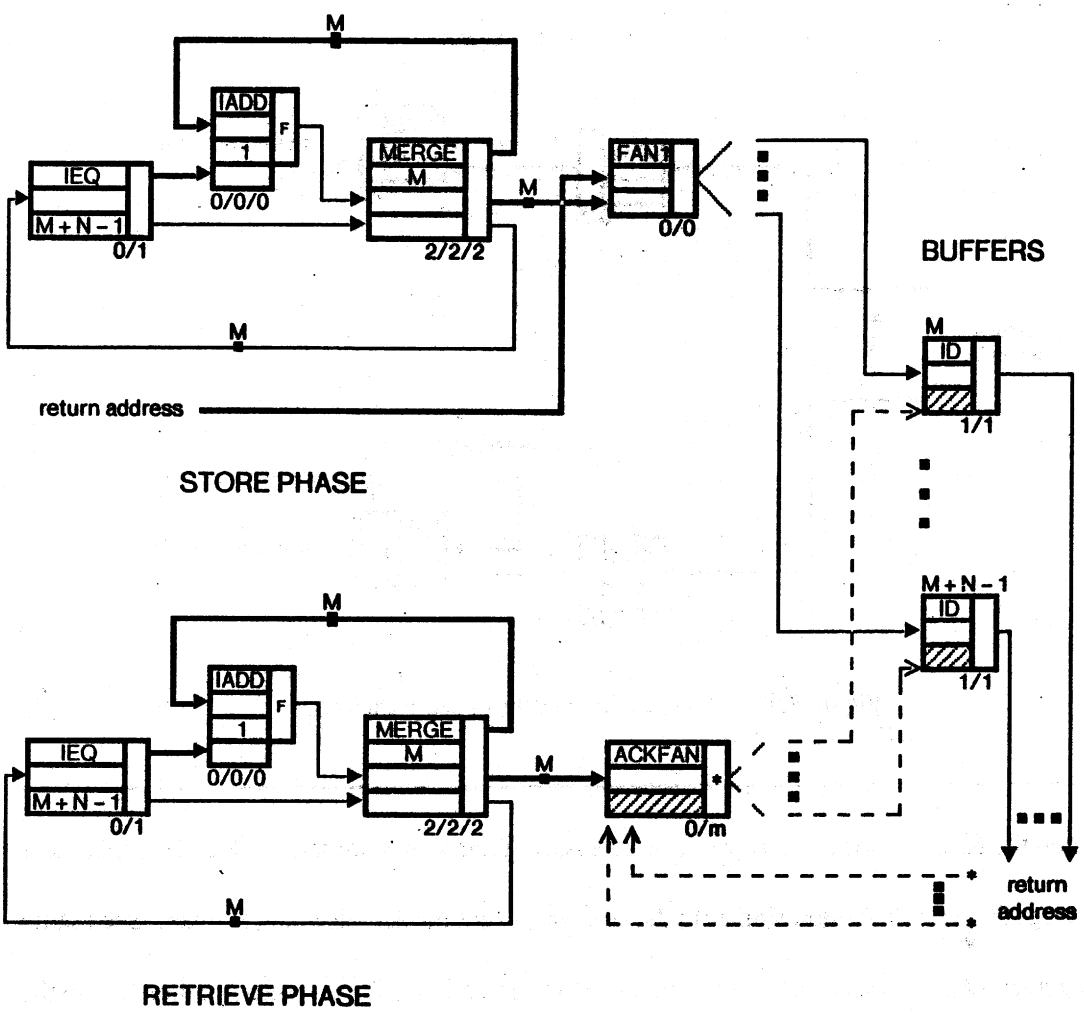

Figure 2.6. Storing the Return Address of the Function Caller.

---

first operand of cell number M, the second address it gets in cell $M+1$, and so on. After it has filled cell $M+N-1$, it starts back at cell M. The retrieve phase works in the same manner, except that it causes the cells to execute, sending the return addresses to greet the return values of the function. Using this method, only nine cells fire per function call to save the return address at the cost of only eight extra cells. It is important that the size N of the queue be large enough so that the queue does not become filled beyond its capacity. Should such an event occur, a new return address would overwrite an old one stored in the queue.

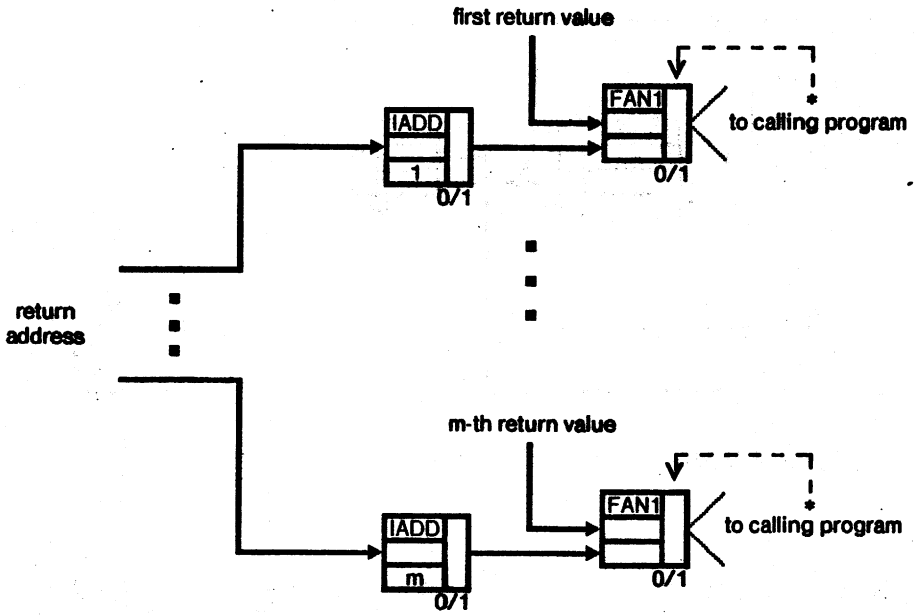

Figure 2.7. Returning the Results of the Function.

When the function body finally produces its results, their values are sent back to the caller (cells $X+1$ through $X+m$) in a fashion similar to the one which sent the actuals to the formals. See Figure 2.7. The return address is sent through a series of IADD instructions. Their results are then sent to the second operands of FAN1 instructions, whose first operands have obtained the return values. The FAN1s execute, sending the values of their first operands to the first operands of the cell numbers specified by their second operands. Thus, the values produced by the functions are returned to the caller.

Actually, this implementation of functions is not totally free from deadlock. For example, let there be two expressions which share the same function with the output of one expression being the input to the other one. Supposed the first expression receives a constant stream of inputs and produces a stream of outputs faster than the second expression can process them. Things would back up into the first expression and eventually into the function body. If the second expression would

then call the shared function, if would not be able to complete because of the backing up.

To solve this problem, a slight modification needs to be made to the sharing scheme. Instead of cells $X+1$ through $X+m$ sending acknowledgments to the FANIs of Figure 2.7, they send them to cell X. This prevents any site from calling a shared function unless the call can be completed and the values returned.

2.5 The Forall-eval Construct

The general form for the **forall eval** construct is

```
forall i in [lo_expression(Y), hi_expression(Z)]
eval forall-op element_expression(i, X)
endall
```

where X, Y, and Z are all vectors, not necessarily disjoint.

Using the **let** and **if** constructs, the **forall eval** can be rewritten to more explicitly reveal what computation is being performed.

```
let
   lo: integer := lo_expression(Y);
   hi: integer := hi_expression(Z)
in
   if is error(lo) | is error(hi) then undef[*]
   elseif lo > hi + 1 then undef[*]
   else init
      OP element_expression(lo, X)
      OP element_expression(lo + 1, X)
         .
         .
         .
      OP element_expression(hi, X)
   endif
endlet
```

If either the low or high index is an error value or if the low index is greater than the high index plus one, the value of the construct is **undef** of the proper types and arity. Otherwise, the result is the

value obtained by applying the element expression to each of the integers in the range [*lo*, *hi*] and performing the selected operation on the resulting values. Table 2.1 gives the values for *init* and OP corresponding to the choice made for "forall-op". Notice that *init* is simply the identity for the chosen operator.

The basic plan for evaluating the **forall eval** construct is to pipeline all $hi-lo+1$ activations of the element expression through one copy of its body. First, though, the tests for bad low and high index values must be performed. This part of the translation is straight forward and is shown in Figure 2.8.

Having gotten the easy part out of the way, the complex part now needs to be tackled. The plan itself is simple: generate a *stream* of inputs (*lo*, *X*), (*lo*+1, *X*), . . . , (*hi*, *X*) to be fed into the element expression which will produce a stream of results from out of the element expression. As each value is put out by the element expression, it is OP-ed with the partial result, whose value is initially *init*, to produce a value which the partial result then takes on. A stream of Boolean values is also needed, to decide when the partial result is the final result so that it can be sent on its way.

Figure 2.9 accomplishes all of this: generating the input stream, operating on the output stream, and returning the final value when it has been calculated. The *OP is the opcode that corresponds to OP, "init" is defined as before, and "x" is the size of vector *X*. This scheme also

---

Table 2.1. The Values for *init* and OP Based on the Selection of "forall-op".

forall-op	*init*	OP
plus	0 or 0.0	+
times	1 or 1.0	*
min	pos_over[*]	min
max	neg_over[*]	max
or	false	|
and	true	&

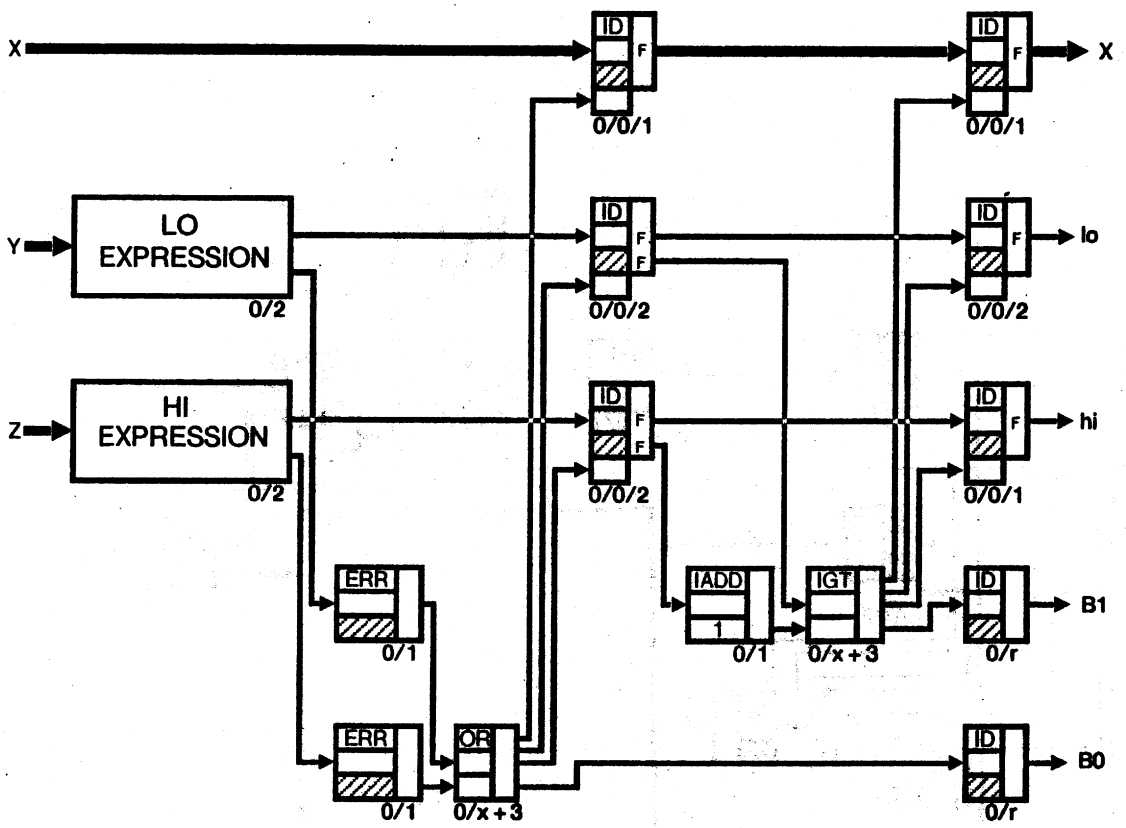

Figure 2.8. Testing for Bad Low and High Index Values in the **forall eval.**

---

allows for a second activation of the **forall** to begin its use of the body of the element expression as soon as the first one has terminated its stream generation.

When finally computed, the sub-result is then sent through a chain of two MERGEs that are needed because of the error tests performed earlier. This is shown in Figure 2.10.

For a higher degree of parallelism, it is possible to pipeline through several copies of the element expression, the number of which could be specified in the advice file for the source program. In the general case of n copies numbered 0 through $n-1$, the "lo" of Figure 2.9 is replaced by "lo+i" in the i^{th} copy, and the value of the second operand of the IADD instruction is changed from 1 to n in all copies. The sub-results are then OP-ed through a binary tree of *OPs as shown in Figure 2.11

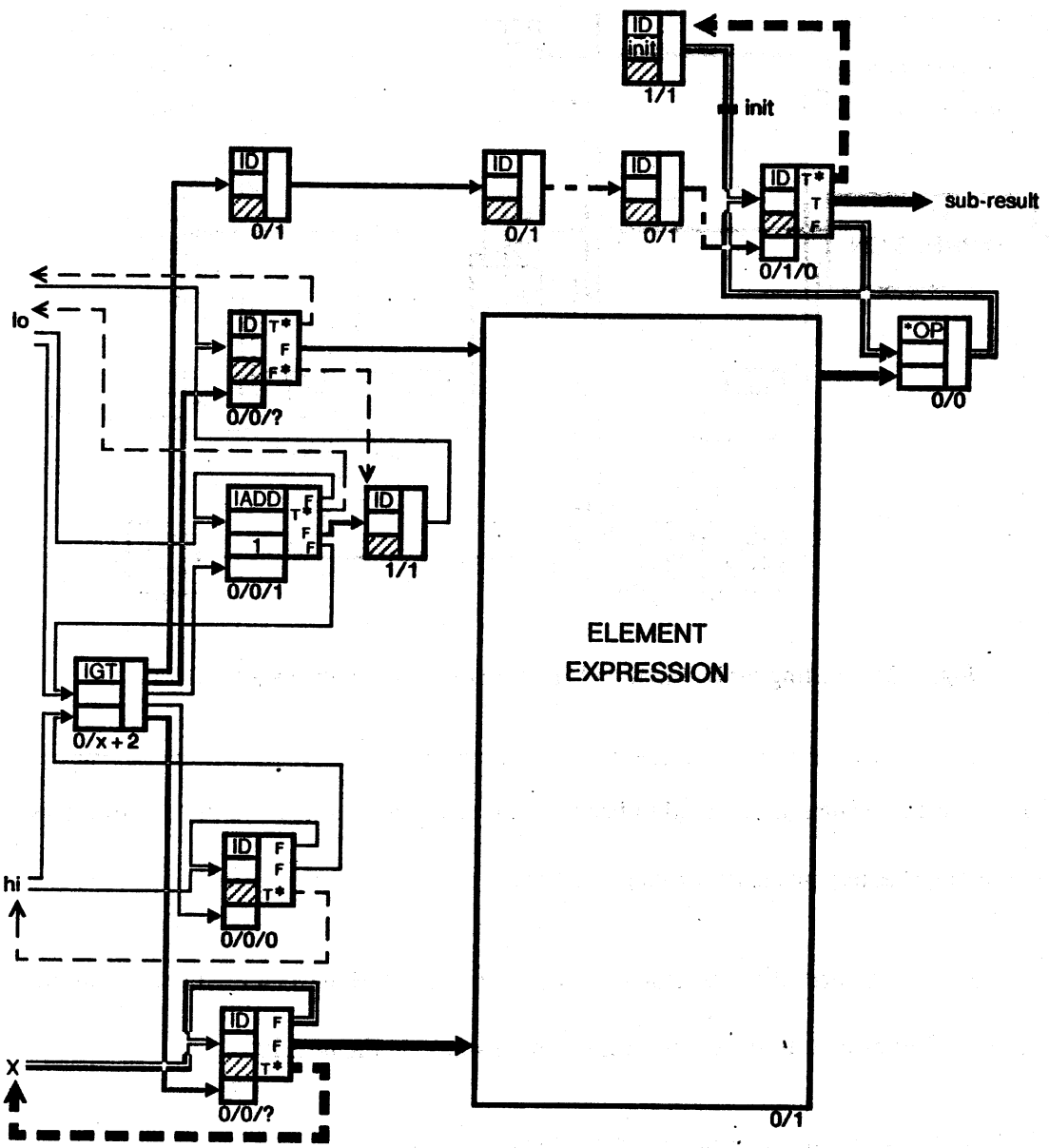

Figure 2.9. Pipelining Through The Element Expression of the **forall eval.**

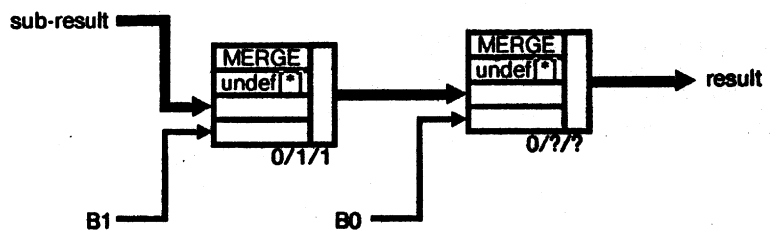

Figure 2.10. Producing the Final Result of the **forall eval.**

---

before they are sent on to the MERGEs.

Multi-dimensional **forall** evals are treated recursively, just as the **elseif** is treated in the **if** construct. For example,

forall *i* **in** [*A*, *B*], *j* **in** [*C*, *D*]
eval forall-op *E*
endall

is equivalent to

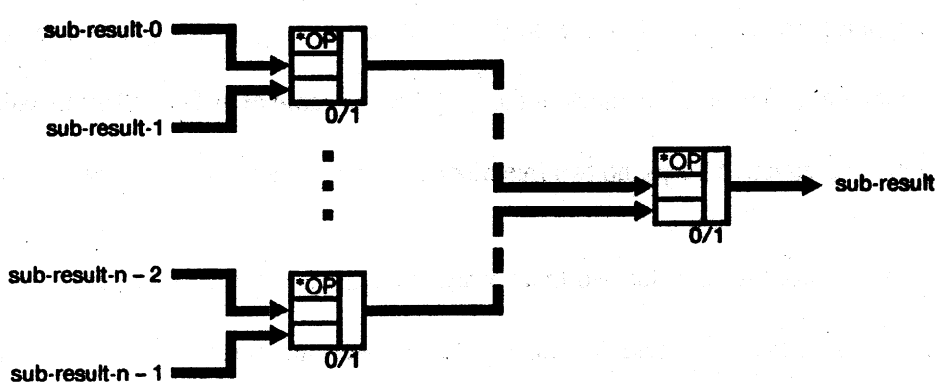

Figure 2.11. Binary Tree of *OP Instructions.

```
forall i in [A, B]
eval
   forall j in [C, D]
   eval forall-op E
   endall
endall
```

2.6 Arrays

The VAL language supports dynamic arrays. Since the Form 1 data flow machine is totally static in nature it cannot support anything dynamic. Thus, if dynamic arrays are going to be implemented, it is going to have to be done with the aid of program transformations performed in software.

The inclusion of dynamic arrays in a language suggests using a heap of some form implemented on the computer supporting the language. In a static data flow machine, a heap can be constructed by using an odd number of FAN1 instruction cells, the first operands of which hold some piece of data, be it the value of an element of an array, a cell number (pointer) within the heap, or total garbage.

The heap itself is divided up into a series of consecutive blocks, of which there are two types: *free* and *allocated.* A free block consists of an even number of consecutive FAN1 cells. The data held by its first cell is a pointer to the last cell in the block, while the last cell points to the first cell of the next free block in the heap. The last free block in the heap has the value of its last cell set to **undef[*]**. As with all properly implemented heaps, no two free blocks are ever adjacent to each other.

The other type of block is one allocated to an array. Like the free block, it consists of an even number of cells. Its first two cells hold the values of the lower and upper bounds of the array respectively, and the rest of the cells hold the array elements. Should an array actually be odd in size,

the last cell in the block goes unused.

The reason for having every block an even number in size is to avoid the situations that create free blocks of size one. Note, though, that the heap itself is odd sized. That odd cell is at the front of the heap and is used to point to the first free block in the heap, or it contains the value **undef[*]** if the whole heap has been allocated. As an example, a snapshot of the heap with two free blocks and one allocated block is shown in Figure 2.12.

To manage the heap, there is a single system function called *heap* which is shared by all users of the heap but not directly accessible by them. When an array is created, a call is made to the manager in the form of *heap*(**true**, *lo*, *hi*), where *lo* and *hi* are the lower and upper bounds of the array to be built respectively. If the bounds are legal, then the manager allocates a block in the heap, sets the first operands of the first and second cells to the values of the lower and upper bounds, and returns the cell number of the first cell in the allocated block; otherwise, the function call produces **undef[*]**. The value **undef[*]** is also returned if there is no free block big enough to hold the prospective array. Thus, an array is simply an integer whose value is interpreted as a pointer to the base of an allocated block in the heap. To free an allocated block, *heap*(**false**, *base_cell_no*, 0) is called, which frees up the allocated block that begins with cell number *base_cell_no*.

It remains to be said how to get at and set the data held by the cells. As shown in Figure 2.12, each cell has received its first operand, has not received its second, has its third constant at the value **true**, needs no acknowledgments and never expects any. All each one needs before it can fire is its second operand, which because the cell is a FAN1, is the number of the cell to receive the value of the first operand, *i.e.*, the data held by the cell. Figure 2.13 shows how this data can be both accessed and changed. Note that the heap behaves like magnetic core memory in that it has the attribute of destructive readout. The FAN2s retrieve the given cell's data and the FAN1s replenish it, either with

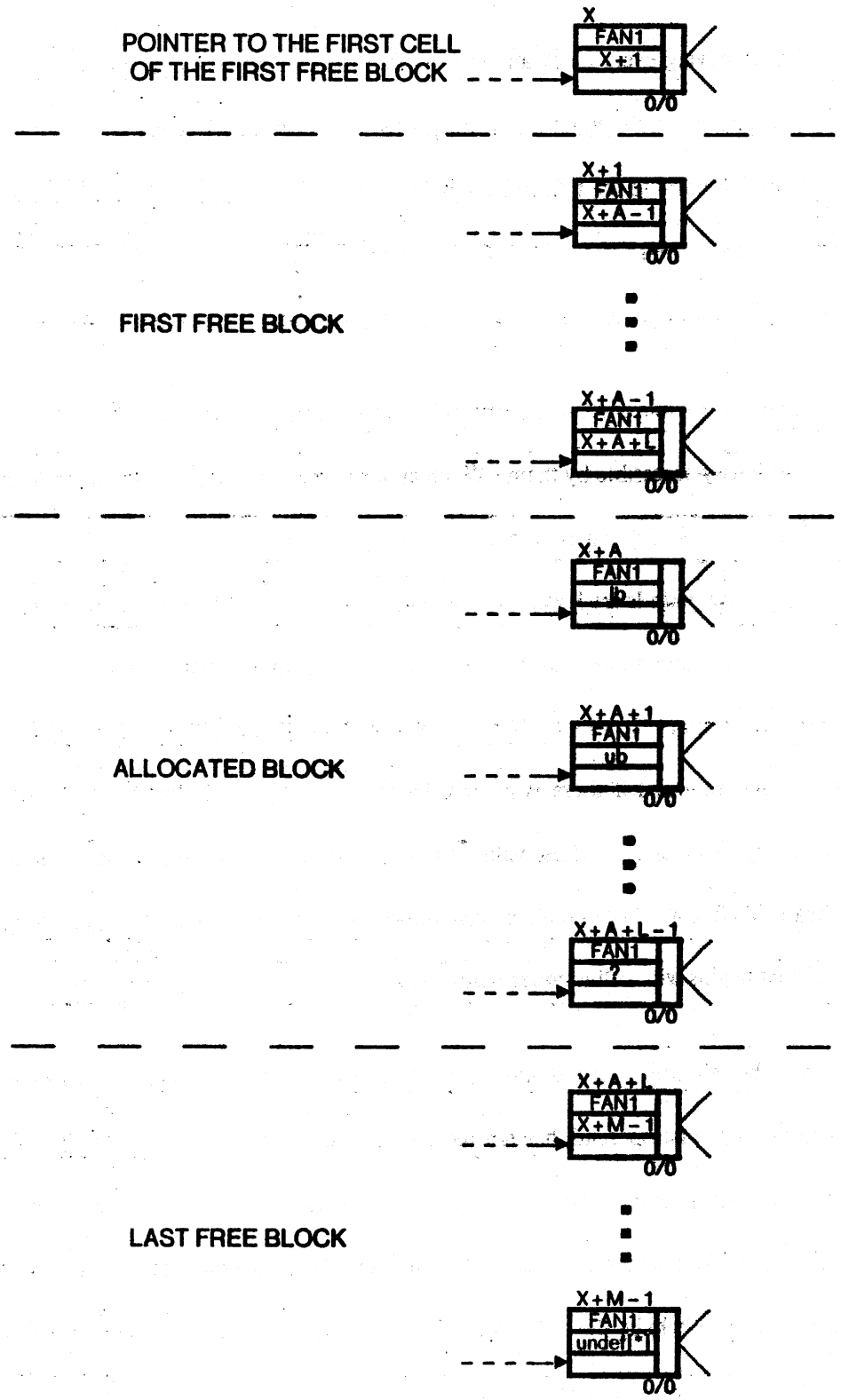

Figure 2.12. Snapshot of the Heap.

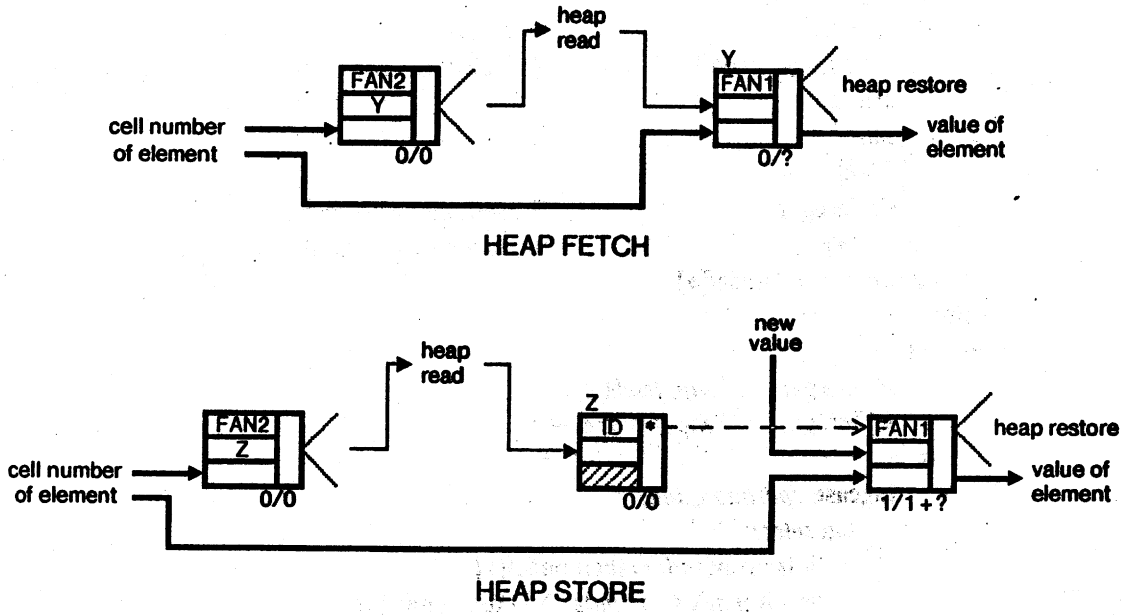

Figure 2.13. Operations on the Heap.

---

the cell's old value for a heap fetch or with a different value in the case of a heap store.

Like the function *heap*, fetching and storing element values in the heap can also be performed by system calls not directly accessible by the VAL programmer. The heap fetch is written as the function *heap_fetch*(*cell_no*) and the heap store appears as *heap_store*(*cell_no*, *value*). Thus, if *A* is an array in the machine, then *heap_fetch*(*A*) returns the lower bound of *A* and *heap_fetch*(*A*+1) yields *A*'s upper bound.

To access a specific element of the heap, either an array element or its bounds, the following function is used instead of the more primitive *heap_fetch* so as to avoid simultaneous and illegal accesses to the heap:

```
function array_access(
        A: integer;                     % the array
        operation_tag: oneof[           % determines the operation to be performed
            select,
            liml,
            limh];
        index: integer                  % used only with select
        returns *)                      % integer or type of A's elements
    if is error(A) then undef[*]
    else
        let
            lo: integer := heap_fetch(A);
            hi: integer := heap_fetch(A + 1)
        in
            tagcase operation_tag;
                tag select:
                    if is error(index) then undef[*]
                    elseif index < lo | index > hi then undef[*]
                    else heap_fetch(A + index - lo + 2)
                    endif
                tag liml: lo            % array_liml(A)
                tag limh: hi            % array_limh(A)
            endtag
        endlet
    endif
endfun
```

The programmer is not allowed direct access to this function nor is there any need for such access. Instead, access is gained through three basic array operations: element selection, index of highest, and index of lowest. Thus, *A*[*J*] translates directly as *array_access*(*A*, make *operation_tag*[*select*: nil], *J*), array_liml(*A*) as *array_access*(*A*, make *operation_tag*[*liml*: nil], 0), and array_limh(*A*) as *array_access*(*A*, make *operation_tag*[*limh*: nil], 0). By prohibiting the programmer to call this function directly, compile time checking can be preserved and optimizations in the translation of this function can be performed such as omitting the error test for *operation_tag*. Only one copy of this function exists per VAL program, which is shared among all arrays.

Now that a heap is implemented, a way to create and initialize arrays must be devised. To keep things simple, all arrays except those constructed using the array_empty primitive are created by

using **forall construct**, whose translation into instruction cells will be explained in a later section. Since VAL contains some array constructors that do not use the **forall**, these must be translated into equivalent forms that do use **forall construct**. These translations can be found in Appendix 2. As for the **array_empty[*]** constructor, it is translated directly as the call *heap*(**true**, 1, 0).

2.7 Records

Records in VAL are a means for grouping together logically related data items that are not necessarily of the same type. Since the Form 1 machine does not support compound types, it does not support records. It does, however, support scalar types from which records are created. Therefore, the simplest way to translate a record from VAL into instruction cells is to break it up into its components. For example, the record

record[*real_part*: **real**; *imaginary_part*: **real]**

is broken up as

real_part: **real**;
imaginary_part: **real**

In the case of a record having other records as components, those component records can also be broken up as long as there are no recursive or circular record definitions.

Some records may be large and heavily utilized, and the above implementation may yield a bulky program. Other records might be recursively defined or contain circular definitions. For records like these, it is possible to use arrays to hold the values of their components. Given a record with *n* components, each component is assigned to a number from 1 to *n*. An array is then created with lower and upper bounds of 1 and *n* respectively, and each element of the array corresponds to a particular record component. The record is then treated exactly as if it were an array. For example, in the previous record, the value of *real_part* would be held in the array element indexed at 1 and the

element indexed at 2 would hold the value of *imaginary_part*.

To create an *n*-component record using an array implementation, a **forall** construct could be used. The index name (the value name appearing just before the **in** expression of the **forall**) is set to vary over the range of 1 to *n*. Within the body, either a tree of **if** constructs or a **tagcase** construct is used to test for the different values of the index name, with the appropriate component value being stored at the proper location within the heap for the corresponding value of the index name.

To select a component of a record, a simplified version of the *array_access* function would be used which lacks the bounds fetching and the range testing:

```
function record_access(
        X: integer;                    % the record
        component: integer             % the component of the record
        returns *)                     % type of an element of R
    if is error(X) then undef[*]
    else heap_fetch(X + component + 1)
    endif
endfun
```

Like *array_access*, only one copy of *record_access* exists per program and the programmer is allowed access to it only through the VAL record component selector. To perform the record **replace** operation, an array append is performed.

For large records implemented in the latter fashion, the end result is possibly a slower executing program than one that uses the former method, but also one that is smaller in size since only a pointer to the record and not the entire record itself is being passed around. The choice of which implementation to use could be specified in the advice file for those records where a choice is possible.

2.8 Unions/The Tagcase Construct

An element of type union consists of a *tag* and a *value.* The type of the value can be one of several types, and the tag is used to select which type that value is. For the Form 1 machine, a union can be implemented as if it were a record with two components. Each tag of the union is encoded as an integer which is stored in the first component of this record. The second component is used to hold the value part of the union. If one or more selections for the value are record or union types, then the union can be implemented as an array whose first element holds the tag and which is large enough to hold the largest sized value.

A straight forward method for translating the **tagcase** would be to convert it to an equivalent **if** construct where the Boolean expressions test for different values and ranges of the tag value. If this approach were to be used, it would be hoped that the testing would take on the form of a balanced binary tree so as to reduce the worst case execution time. The drawback to this method is that it is using many one-out-of-two decoders (**if** constructs) to create a one-out-of-many decoder (the **tagcase**).

Another possible translation which avoids this problem is shown in Figure 2.14. Each of the t tags is assigned to an integer in the range from 0 to $t-1$. After using an **if** to insure that the tag is not an error value, the value of the tag is added to the integers from A_0 to A_{y-1}, whose results are sent to the second operands of FAN1 instructions. These FAN1s are used to direct the value part of the **oneof**, vector Y of size y, to one of the t expressions corresponding to the arms of the **tagcase**. In a similar fashion, the external values used in the **tagcase** arms, vector X of size x, are sent to the chosen expression.

For a pipelined version of this, race conditions must be eliminated and the FIFO property must

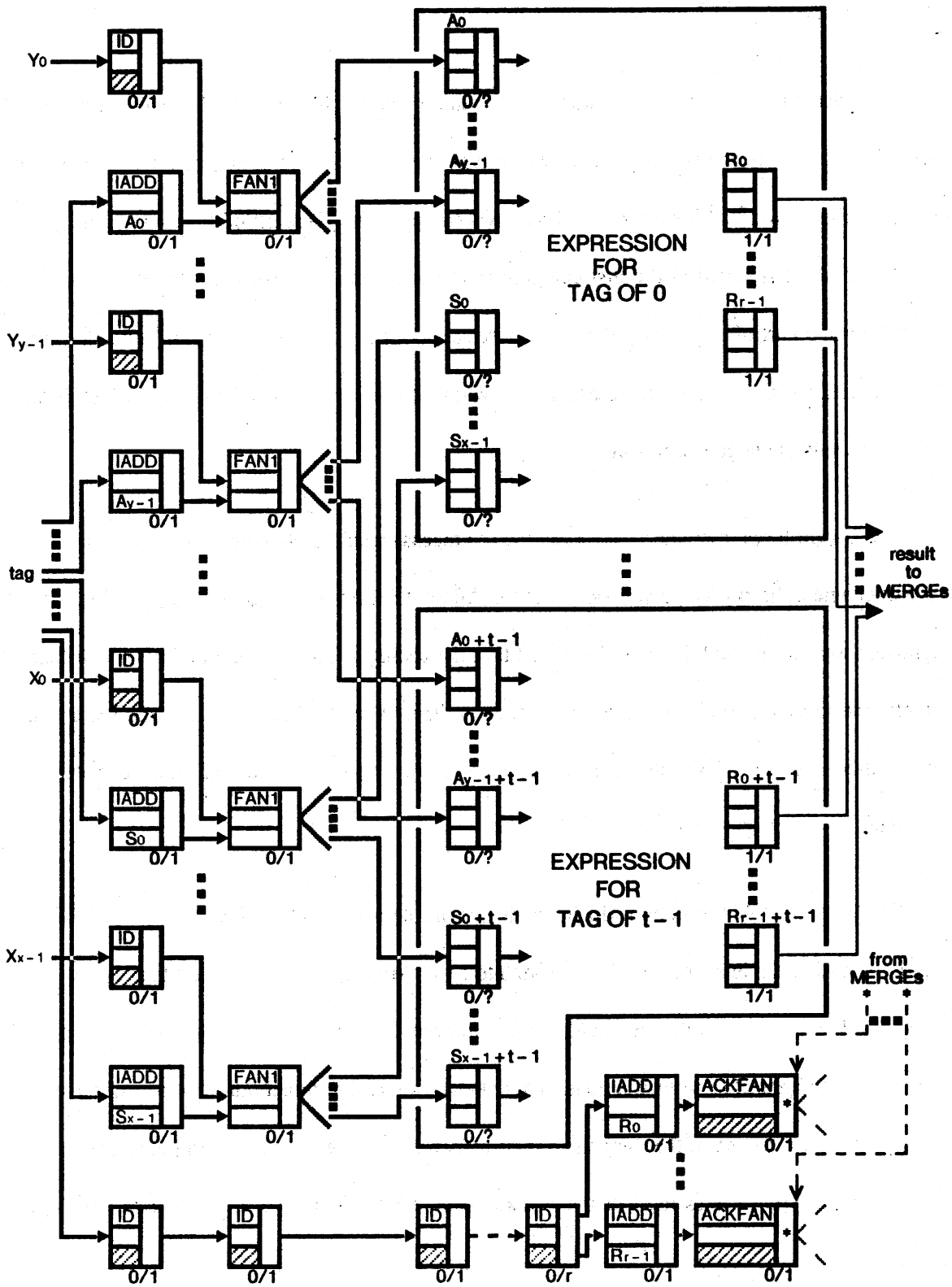

Figure 2.14. The Tagcase.

be preserved. To achieve this, the value of the tag is pipelined around the bodies of the arm expressions. Just before the next result is to be produced, the tag value is added to the integers from R_0 to R_{r-1}. Each result from these additions is sent to an ACKFAN. By sending acknowledgments to the proper set of cells, the firing of these ACKFANs determine which arm expression is to send its values to the MERGEs of the error testing **if**. Once the MERGEs have fired, the next set of return values are allowed to emerge from the arms. This is all shown in Figure 2.14 also. Since the tag values cannot get out of order, the results remain in their proper sequence, thus performing the desired task.

2.9 The Forall-construct Construct

Earlier it was shown how a heap can be implemented on the Form 1 computer. It was also shown how to translate a **forall eval** into instruction cells. The techniques used for these two constructs will now be used to translate the **forall construct.**

To the programmer, the **forall construct** appears as such:

forall *i* **in** [*lo_expression*(*Y*), *hi_expression*(*Z*)]
construct *element_expression*(*i*, *X*)
endall

In detail, however, it looks like this:

```
let
    lo: integer := lo_expression(Y);
    hi: integer := hi_expression(Z);
    A: integer := heap(true, lo, hi)
in
    if is error(A) then undef[*]
    else
        let
            heap_store(A, element_expression(lo, X));
            heap_store(A, element_expression(lo + 1, X));
                    .
                    .
                    .
            heap_store(A, element_expression(hi, X))
        in A
        endlet
    endif
endlet
```

The plan of attack here is just like the one used in the **forall eval**, except that the stream of values put out by the body of the element expression are stored in their proper locations in the heap instead of being operated on.

The first thing to be accomplished is to call the *heap* function. Since *heap* checks that the values for *lo* and *hi* are good, there is no need for the **forall** to perform this duty. Presenting *heap* with bad index values causes the function to return **undef[*]**. Thus, the **forall** needs only to test the returned value for **undef[*]**. This part of the construct is shown in Figure 2.15.

Generation of the input stream to the element expression is accomplished in the exact same manner as it is in the **forall eval**. This is shown in Figure 2.16. The rest of the translation, though, deviates from how the **eval** is performed. Besides needing the stream of Boolean values, a stream of cell numbers must be generated so that the elements of the stream leaving the element expression can be stored in the proper cells in the heap. This can be accomplished as shown in Figure 2.17. The values for the array elements are generated by the element expression from low to high, and as they are produced, they are stored at the correct locations in the heap via the FAN1 since the heap cell

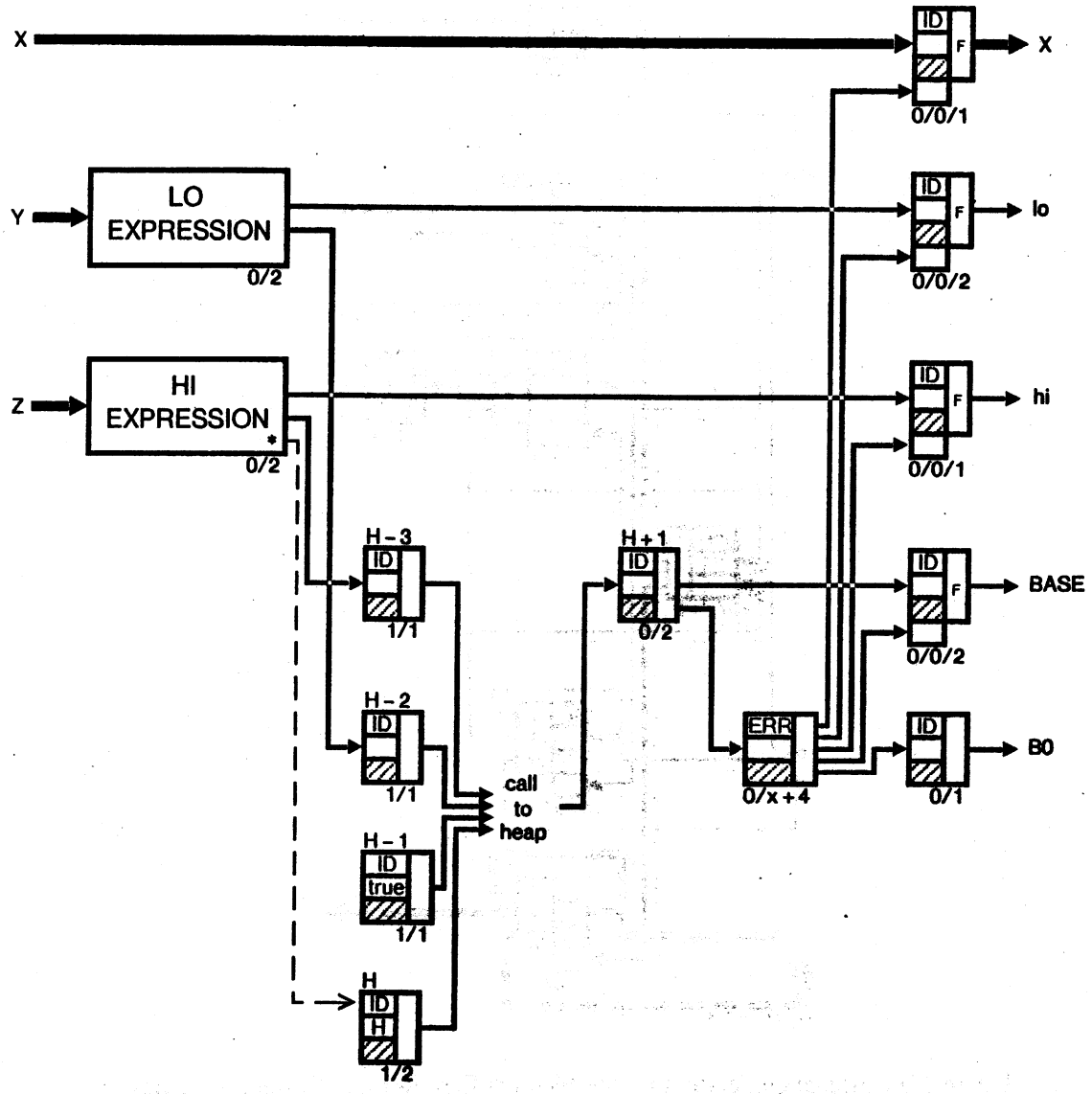

Figure 2.15. Setting Up the **forall construct.**

---

numbers are also generated from low to high.

Like the **forall eval**, it is also possible to use several copies of the element expression in order to achieve greater parallelism. For some integer n, $n \geq 1$, n copies of the cells of Figures 2.16 and 2.17 are generated, except for cells F1, F2, and F3. The two IADD instructions with their third operands shown have the value of their second operands set to a constant n. In the i^{th} copy, where i ranges

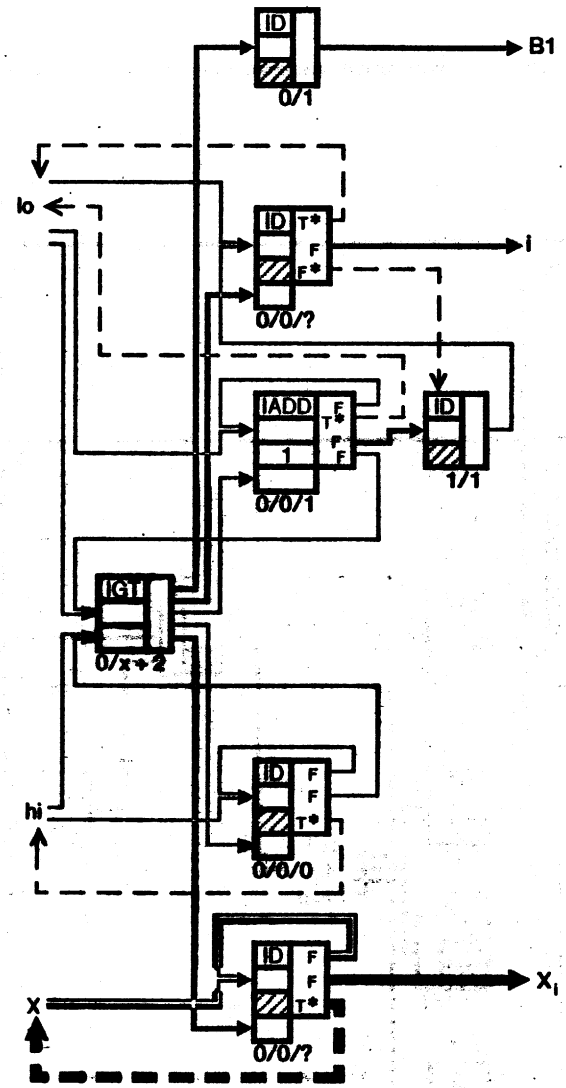

Figure 2.16. Stream Generation for the Element Expression in the **forall construct.**

---

from 0 to $n-1$, *lo* is replaced by $lo+i$ and the third IADD instruction has its second operand set at $1-i$. Just one copy of cells F1, F2, and F3 are used, and the only changes to these is in F2's acknowledgments needed and reset values which are both set to *n*.

Every element in the array must be initialized before the base pointer can be sent out to other parts of the program. Unlike I-structures [2], uninitialized array indexes do not contain "holes" but actual values that are pure nonsense. If the array would be prematurely released and some other part

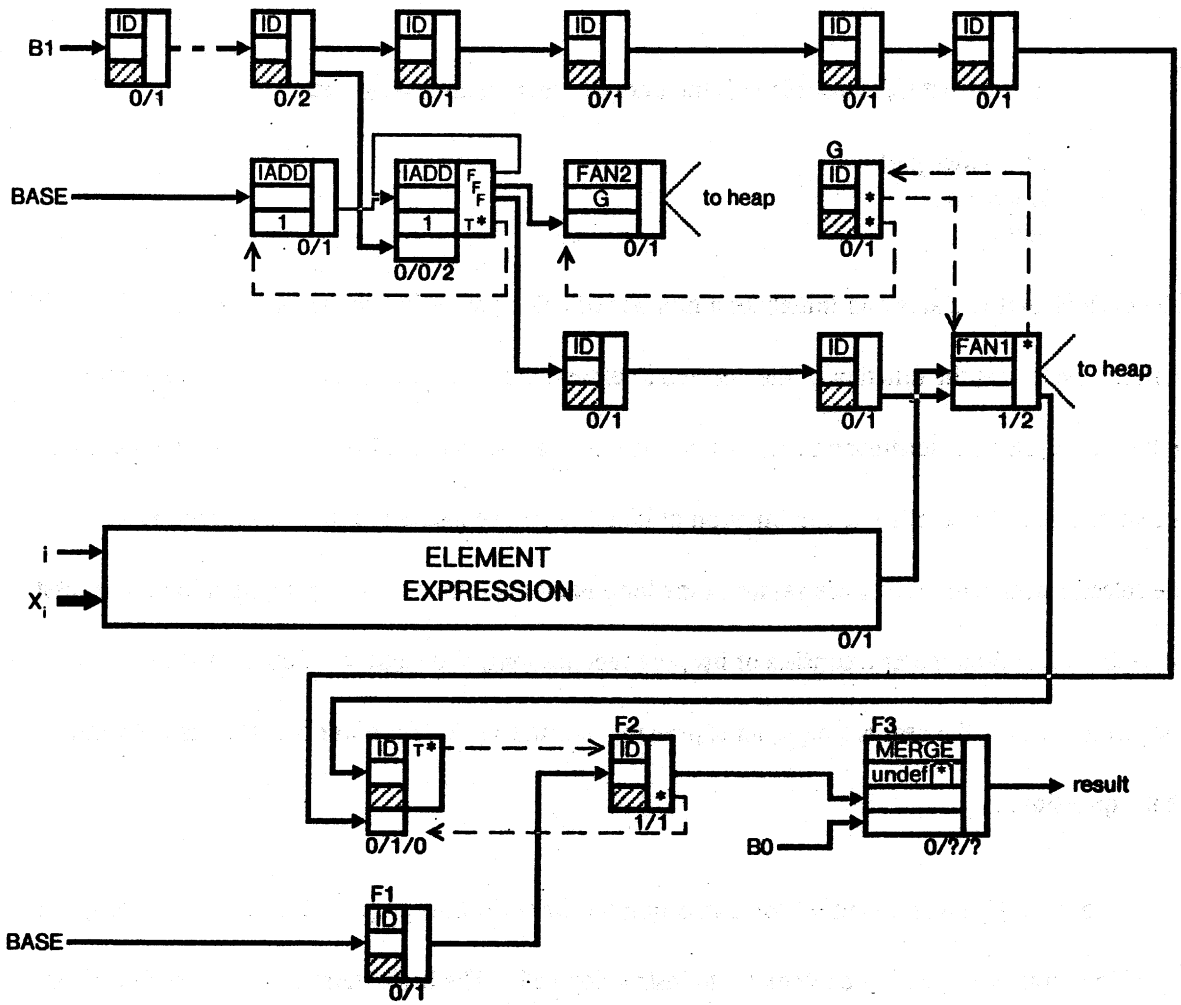

Figure 2.17. Initializing the Elements of the Array Created by the **forall construct.**

---

of the program tried to access an uninitialized element, that part of the program would receive garbage for the value of that array's element and not the value that would eventually be created.

Two other points should be mentioned. First, a **forall construct** that creates *n* arrays where *n* > 1, is best transformed into *n* **forall constructs** that create one array apiece since the *heap* function is not pipelinable and must be shared by all array constructors of the program. Second, the creation of multi-dimensional arrays is handled in the same manner as multi-dimensional **forall evals.**

2.10 The Iter Construct

The last construct to be examined is the iter construct. It appears as such:

for decldef-part
do iter-end
endfor

The decldef-part declares and initializes a number of value names known as *loop names.* With each iteration cycle, it is the function of the iter-end to either change the values of these loop values for use in the next cycle or to terminate the iteration and return a final set of values. The form of the iter-end is that of a nested tree of some combination of **if** and **tagcase** constructs with a slight modification. If the selected arm is to modify the values of the loop names and thus continue the iteration for at least one more cycle then the arm consists of **iter**, the redefinitions, and **enditer**. If the arm is to terminate the iteration, then the arm is simply an expression and the value of the **iter** construct is the value of that expression.

Montz in [7] gives a method for translating the **iter** into flow graphs. Using these graphs, it is possible to transform the **iter** construct into instruction cells. The basic translation is shown in Figure 2.18. The loop names are given by the vector Z, whose initial values are functions of vector Y. Vector X includes all external value names used within the iter-end. Both vectors X and Z enter the iter-end, and either X and a new Z emerge should the iteration be continued another cycle or the final result comes out and the iteration terminates. The purpose of the "iter?" lines is to signal whether the old activation of the **iter** is to continue, in which case "iter?" is the value **true**, or if the old one is done and a new one can now commence, in which case "iter?" is **false**.

Stoy in [9] suggests that pipelining through an **iter** construct can be advantageous. Since different activations of the iteration will most likely take a different number of cycles, he suggests that a set of buffers be used to hold the results of each activation, and that the contents of the buffers be

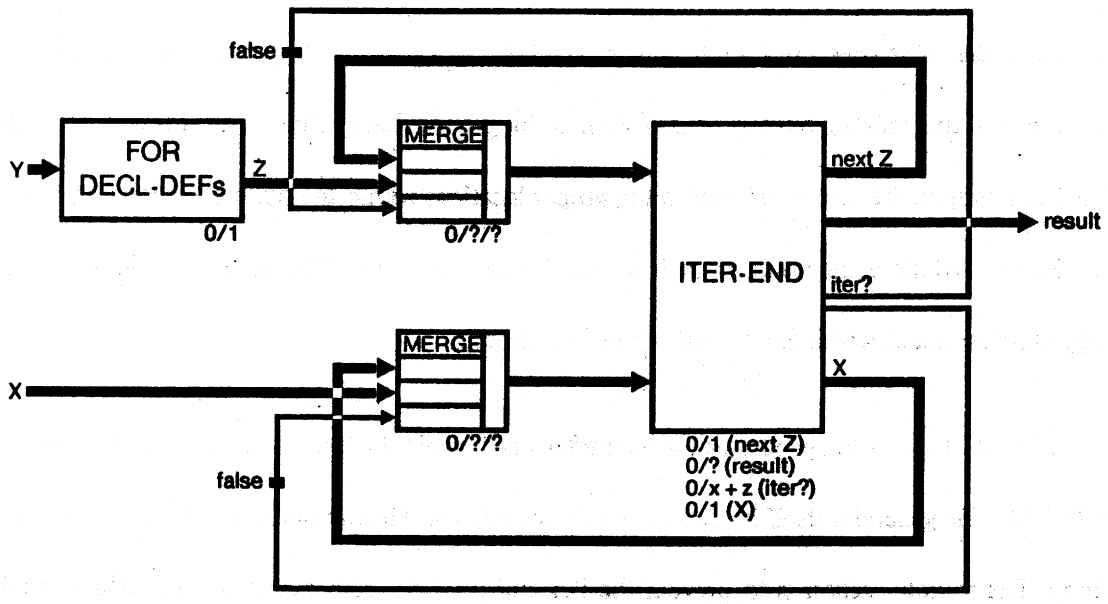

Figure 2.18. The iter Construct.

---

released in the proper order. The rest of this section will show how this method for pipelining can be implemented.

The instruction cells of Figure 2.18 are replaced by those of Figures 2.19 through 2.22. Before a new activation of the iter can enter the body of the iter-end, it must first be assigned buffers to hold its results. The "buffer head generator" performs this function by returning the cell number of the first one of the buffers assigned to the activation. After X and Z have become ready and the buffers have been assigned, the ID instruction whose operand is a constant **false** fires, sending the value **false** to the second operand of the SER instruction. This SER instruction along with the MERGEs is used to arbitrate the entrance to the iteration body between new activations and continued old activations, with the values of the old activations being stored in the first operands of the MERGEs and the new activation's values in the second operands. Eventually, assuming the iteration body is not full of infinite loops, the new activation is allowed to enter. When it finally comes out at the other end, one of two courses of action is taken. Should the iteration continue, X, the new value of Z, and the

pointer to the head of the buffer are all sent back around to the first operands of the corresponding MERGEs and the value true is sent to the SER instruction. This part of the scheme is shown in Figure 2.19. If instead the iteration has terminated, then nothing is sent back to the front. Instead, using the buffer head pointer, the results are sent to the proper locations within the buffers as shown in Figure 2.20. Once stored there, they wait for their eventual release, assuming that there are no infinite loops among those iterations which must terminate before this one.

Figure 2.21 shows the generator for the buffer head, while the buffers themselves are shown in Figure 2.22. The generator itself is divided into three sections. The top part generates pointers to the buffers. The middle section is to prevent the iter-end pipeline from becoming full, which would cause deadlock. The bottom is used to signal the next set of results that is to be sent out to the rest of the world. The bottom and top sections are similar to designs presented earlier in this thesis and will not be explained again. The center one signals the top one, telling it whether or not there is room in the pipeline. After each head pointer is sent out, 1 is subtracted from the number held in the first operand of the IADD instruction of the second section, whose value is initially one less than the activation capacity of the buffers. If the previous number is greater than zero (*i.e.*, the new number is greater than or equal to zero) then there is still room in the pipeline and the top section is signaled; otherwise there is no room in the pipeline and the top must wait for an activation of the iteration to complete. Each time a set of results has been consumed by the outside world, cell H fires, which causes 1 to be added to the number held in the IADD. If this number is now zero, then the top part must currently be waiting for a completion and thus it receives its awaited signal.

The buffers consist of $N*r$ consecutive ID instruction cells beginning with cell number M, where r is the arity of the result. The results for a given activation are stored in cells $M+((i-1)*r)$ through $M+(i*r)-1$, where i ranges from 1 to N. The next results to be output by the iter are those

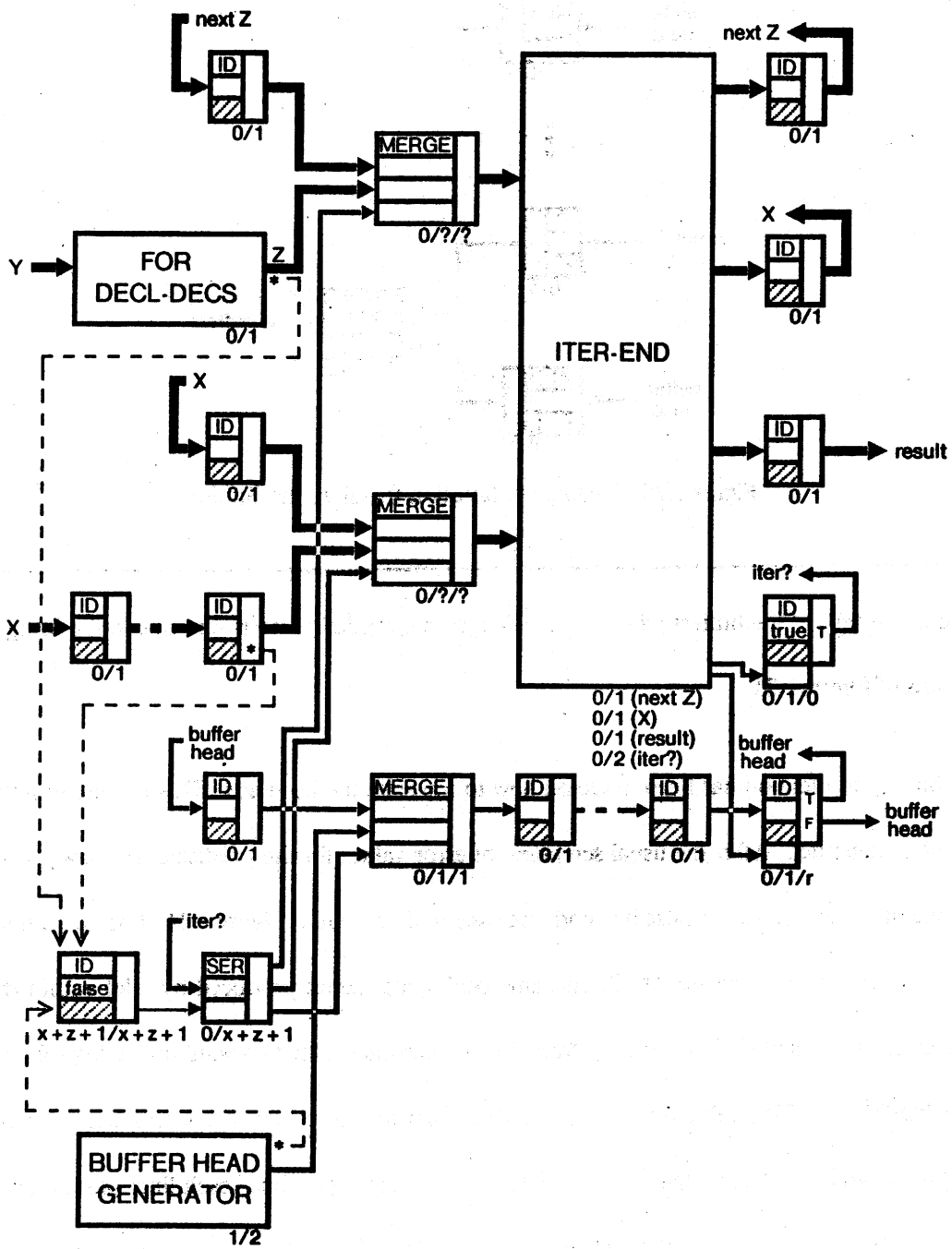

Figure 2.19. A Pipelined Implementation of the iter Construct.

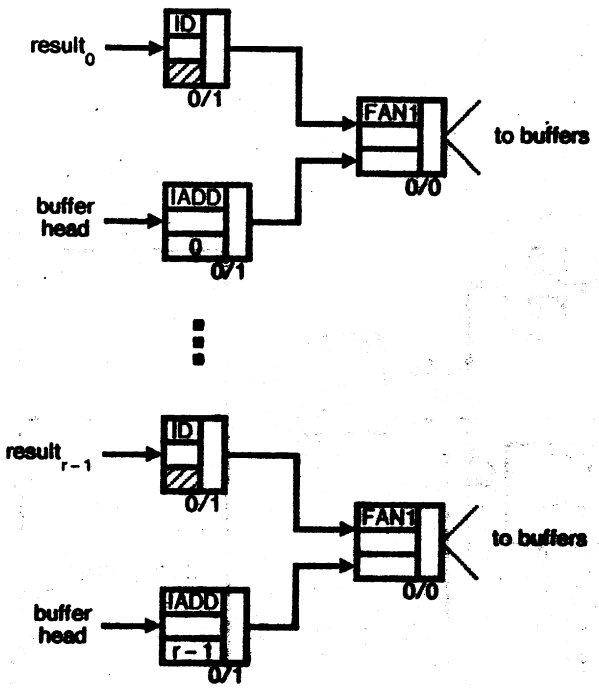

Figure 2.20. Storing the Iteration Result in the Buffers.

stored in the cells of the buffers whose acknowledgments needed value is zero due to the firing of the ACKFANs of Figure 2.21.

So far, no mention has been made on how to translate the iter-ends. This is done so now, first with the if construct. After the usual screening of error values for the predicate, as shown in Figure 2.23, one of either the true or false iter-ends are executed, as seen in Figure 2.24. It is noted here that vector S is the union of vectors X, Z, and any local value names produced by a let within the iter body. Of the six sets of lines protruding from the two iter-ends, only two wind up having values: one of the "iter?s" and if the value of the "iter?" is true then the corresponding X and next Z lines have the other values and if "iter?" has the value false then the corresponding result lines possess the other values. Using the predicate , the proper source for "iter?" is selected, and then using both the predicate and the selected "iter?", either X and the next Z or the result is output along with the value for "iter?". Should the predicate originally evaluate to an error, then the value of "iter?" becomes

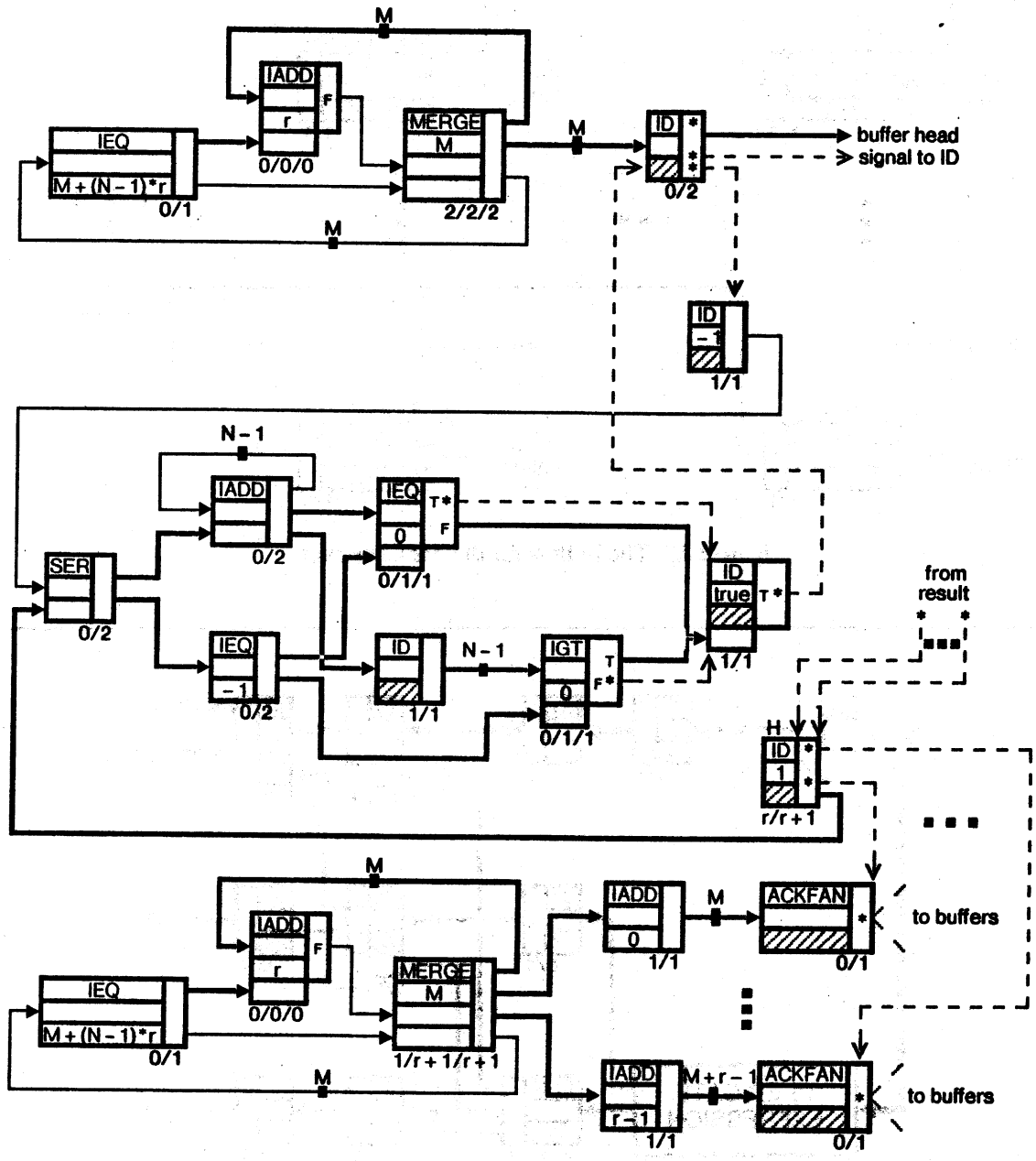

Figure 2.21. The Buffer Head Generator.

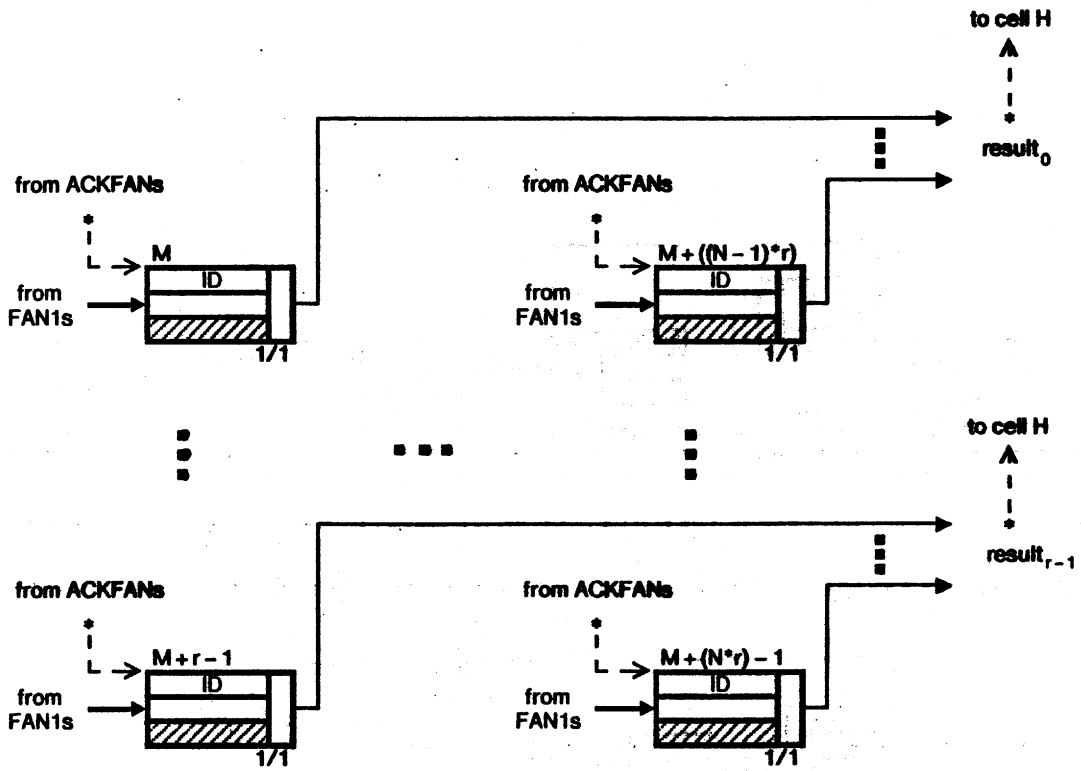

Figure 2.22. The Buffers for the iter Construct.

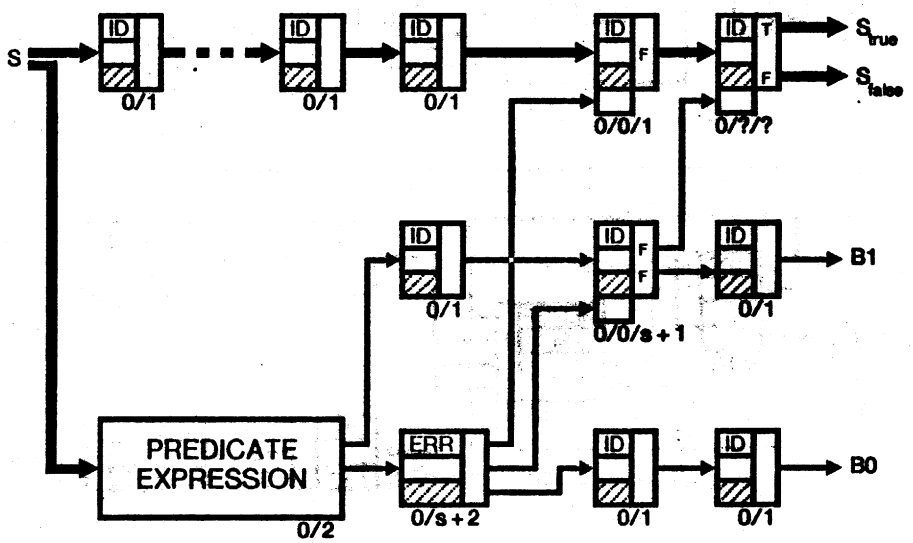

Figure 2.23. The if Construct as an Iter-End.

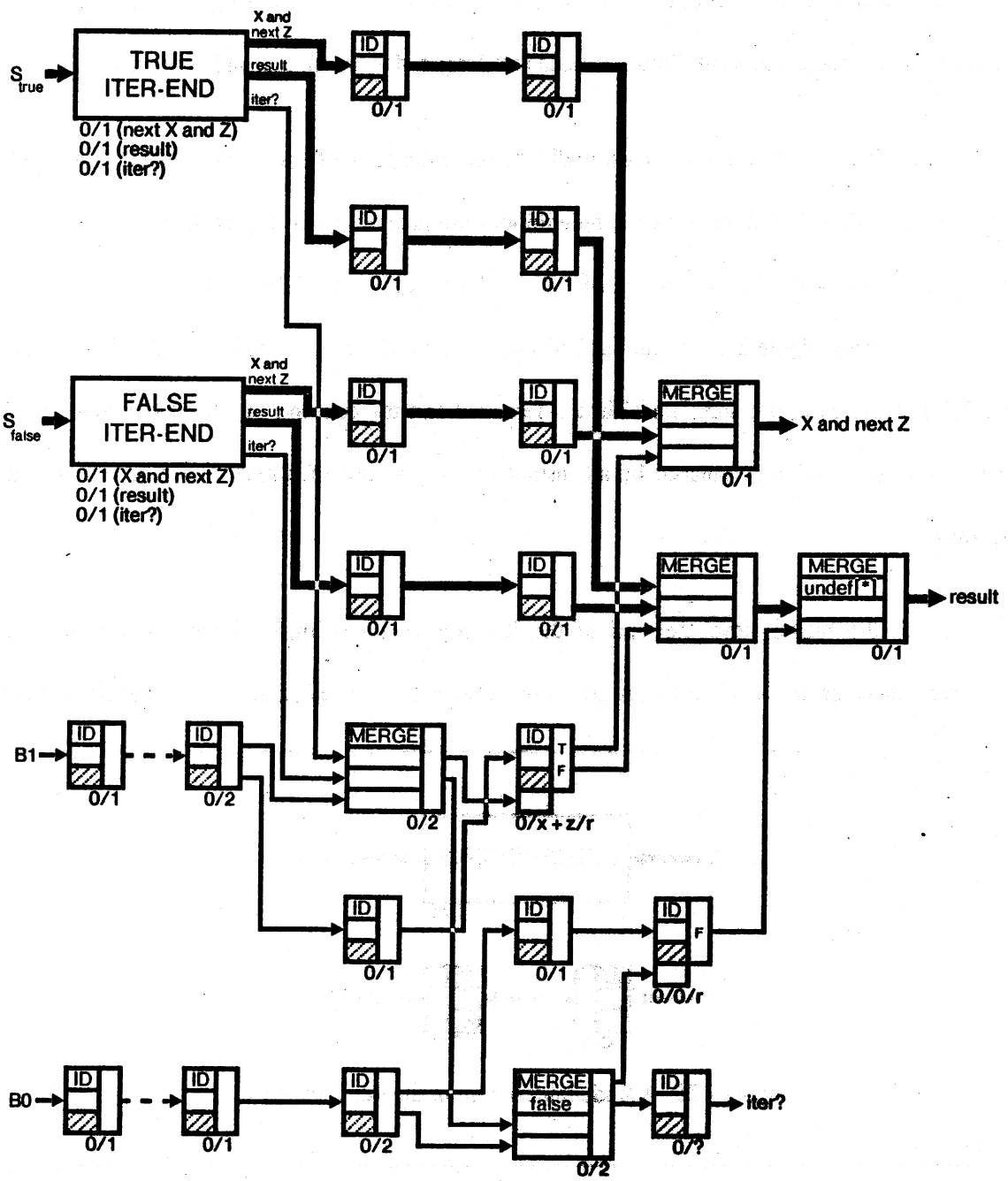

Figure 2.24. The if Construct as an Iter-End (cont.).

false and the result lines are filled with **undef[*]s.**

For a tagcase as an iter-end, a translation can be worked out by combining the translation given for the conventional tagcase with those methods just described in translating an if as an iter-end.

Eventually, the iter-ends filter down until they reach the point of being either an expression or a redefinition. Figure 2.25 shows how the redefinitions are handled while Figure 2.26 covers the case for the expressions. Notice that no value for "iter?" is produced. What is done instead can be illustrated by using Figure 2.24. If one (or both) of the iter-ends takes the form of either Figure 2.25 or 2.26, then the corresponding "iter?" line is removed and the operand of the MERGE that would receive the "iter?" value is replaced by a constant true in the case of a redefinition or false for an expression.

Should the highest level iter-end of the iter construct be simply either a redefinition or expression, then the above process does not work. This is not something to worry about since these

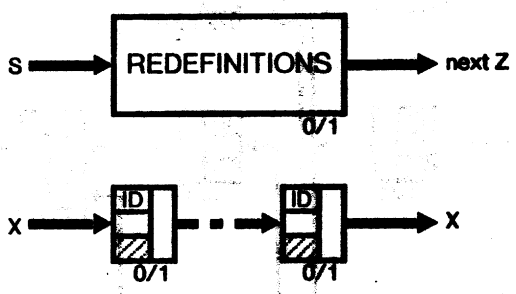

Figure 2.25. A Redefinition Arm in the iter Construct.

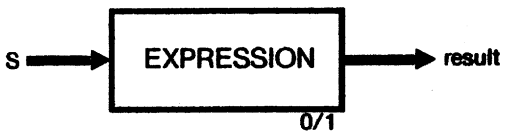

Figure 2.26. A Terminating Arm in the iter Construct.

two cases misuse the iter construct. Should the highest level iter-end be a redefinition then the iter is actually an infinite loop. On the other hand, should that iter-end be an expression, then the iter functions as though it is a let construct.

A question remains in how many of sets of buffers to allocate. To determine this number, the number of cells in the "buffer head" loop (or the "next Z" or "x" loops since they should all contain the same number of cells) of Figure 2.19 is counted. Let a *full cell* be one that has received all of its operands and an *empty cell* be one that has not received any of its operands. A full cell can fire only if there is an empty cell directly in front of it. Since it is desired to maximize the number of full cells that can fire at any given instant, each full cell in the loop should have an empty cell in front of it. It is also desired to have as many full cells and thus activations in the loop as possible. Thus, the loop should appear as a sequence of alternating full and empty cells. This puts the number of sets to use at $\lceil$(*the number of cells in the loop*)/2$\rceil$, where the extra cell is full should there be an odd number of cells in the loop.

2.11 A Few Notes About Pipelining

One of the main goals of the translations presented in this chapter is to take advantage of pipelining. This section briefly mentions a couple of relevant issues concerning this.

The first thing that needs to be noted is that these pipelined translations will end up with larger execution times because of all of the ID buffers if the machine does not contain a large enough number of processors. Because the routing networks will probably be built out of many 2×2 routers, the number of processors will most likely be a power of 2. From my experience of working with instruction cell programs and running them on an interpreter, for small programs at least 16 processors are needed. For larger sized programs, 32 processors is the bare minimum, with 64 or

3. An Example

Using the translation schemes presented in the previous chapter, it is now possible to translate VAL programs into instruction cells so that they can run on the Form 1 machine. Since this machine is still in the developmental stage, the best than can be done for now is to run the translations on an interpreter. This chapter will run and analyze three different translations of two different algorithms for calculating the dot product of two vectors.

The first algorithm uses an **iter** construct and appears below:

```
let
    lo: integer := array_liml(A);
    hi: integer := array_limh(A)
in
    for
        sum: real := 0.0;
        i: integer := lo
    do
        if i > hi then sum
        else
            iter
                sum := sum + A[i] * B[i];
                i := i + 1
            enditer
        endif
    endfor
endlet
```

Here, the products of the elements are explicitly added to a partial sum one at a time. This algorithm is similar to those that might be written in conventional languages like FORTRAN and ALGOL to perform the same task.

The other method exploits the inherent parallelism in computing the products by using a **forall** **eval**:

```
forall i in [array_liml(A), array_limh(A)]
eval plus A[i] * B[i]
endall
```

Two different translations were derived for the algorithm using the **iter** construct, one with pipelining and one without, while the standard translation was used for the algorithm using the **forall** eval using only one copy of the body. Each one was run using the interpreter for instruction cells as documented in [10] and the results from execution are displayed in Table 3.1. As shown there, five comparisons are made: the total number of cells used by each translation, the number of passes each needed to calculate the dot product of two six dimensional vectors and two three dimensional vectors, and the throughput of each in working with a stream of fifteen pairs of six dimensional vectors and a stream of fifteen pairs of three dimensional vectors. (A pass is defined as the execution of just those cells that are currently on the queue. As they execute, other cells become ready to fire and are appended to the rear of the queue but are not executed until the next pass.)

As expected, the **forall** translation yielded the best performance record while the non-pipelined **iter** finished last overall. The interesting part, thus, is how the pipelined **iter** performed in relation to the other two. For a single pair of vectors, it produced the worst time of the three in both cases as would be expected because of the amount of overhead. In the stream tests, however, after getting off to a slow start it quickly caught up and exceeded the performance of the non-pipelined translation by the fourth pair in the worst case. By the time the dot product of the last pair of the stream of six dimensional vectors was produced, its execution time was less than half of that of the non-pipelined **iter** and it was even challenging the throughput of the **forall**. The statistics for the throughput of the stream of vectors of dimension three are not as impressive, but they do show a definite leaning toward the performance of the **forall**.

The price paid for the relatively good execution times of the pipelined **iter** was heavy in terms

Table 3.1. Comparison Between Three Different Dot Product Translations.

	Forall	Pipelined Iter	Non-Pipelined Iter
total number of cells used	238	472	225
output times (in passes) for:			
a single pair of 3-vectors	85	156	109
a single pair of 6-vectors	106	240	154
a stream of 15 pairs of 3-vectors:			
pair #1	87	162	109
pair #2	118	216	185
pair #3	149	247	261
pair #4	180	279	337
pair #5	211	296	413
pair #6	242	350	489
pair #7	273	384	565
pair #8	304	408	641
pair #9	335	429	717
pair #10	366	481	793
pair #11	397	526	869
pair #12	428	571	945
pair #13	459	616	1021
pair #14	490	656	1097
pair #15	519	696	1173
a stream of 15 pairs of 6-vectors:			
pair #1	114	274	154
pair #2	168	390	277
pair #3	222	442	400
pair #4	276	521	523
pair #5	330	560	646
pair #6	384	649	769
pair #7	438	696	892
pair #8	492	728	1015
pair #9	546	775	1138
pair #10	600	801	1261
pair #11	654	820	1384
pair #12	708	844	1507
pair #13	762	863	1630
pair #14	810	885	1753
pair #15	858	903	1874

of memory used due to overhead. While the forall and non-pipelined iter both used about the same number of cells in their translations, the pipelined iter used twice the number of either of the other two.

Thus, it can be seen that pipelined translations of iter constructs should be used only when it is expected that those iters will be receiving a steady stream of inputs and when there is sufficient memory available to accommodate the tremendous overhead involved. Otherwise, the non-pipelined translation should be the order. Above all else, though, the forall should be used whenever possible.

Had this algorithm been translated for and run on a conventional (von Neumann) computer, it would have consumed much less memory and probably would have run in less time. The lower memory consumption is due to a few of factors. First, the size of a standard instruction in a conventional machine is anywhere from two to eight bytes while an instruction cell uses 32 bytes to hold everything: opcode, operands, and destination addresses. Second, in order to pipeline through the bodies of instruction cell programs, ID instructions are added which are equivalent to "no-op" instructions in conventional machines. Third, the memory usage efficiency that is achieved from arrays in the Form 1 machine is only 12.5% (only 4 out of the 32 bytes in each cell is used to hold data) for arrays of integers and reals.

As for the increased execution speed, this is due to the method used to implement arrays in the Form 1 machine, which is accomplished mainly in software. A significant amount of time is spent restoring the value of the selected array element since reads into the heap are always destructive. A conventional computer can perform this in much less time since it does not have to worry about restoring the value of the element and can normally fetch it with a single instruction through the use of an index register. Not until after a structure processor and memory is added to the data flow machine which implements arrays in firmware and hardware will data flow programs of this type

surpass the performance of programs run on conventional machines.

4. Conclusion

This thesis has presented methods by which the various constructs of the present version of VAL may be translated so as to take advantage of pipelining. Future versions of this language plan to address the issues of input/output, file update, data base applications, streams, and string manipulation. Once these features and the constructs to implement them are included in VAL, translations for them will have to be developed if such is deemed plausible. If not, then they will have to wait for a higher form data flow machine to run on and the Form 1 will have to be content with running only a subset of the language.

The translations that have been presented have assumed that the instruction cells cannot handle a Boolean error value as the third operand. One possible direction for further research would be to modify the operational semantics of the instruction cells so that when presented with an error value as the value of the third operand they execute in a way that will rid the need for explicit error testing in the **if**, **forall**, **tagcase**, and **for** constructs. This will result in a saving of memory along with improved runtime.

Appendix 1. Instruction Cell Opcodes

Boolean Instruction Cell Opcodes

Opcode	*Operation Name*	*Type of Operands*	*Type of Result*	*Result Produced*
AND	Logical and	[1], [2]: **boolean**	**boolean**	[1] ∧ [2]
OR	Logical or	[1], [2]: **boolean**	**boolean**	[1] ∨ [2]
NOT	Logical negation	[1]: **boolean** [2]: not used	**boolean**	¬[1]
EQV BEQ	Logical equivalence/ Test for equality	[1], [2]: **boolean**	**boolean**	[1] ≡ [2]
XOR BUEQ	Logical exclusive-or/ Test for inequality	[1], [2]: **boolean**	**boolean**	[1] ⊕ [2]
BFAN3	Fan to operand #3	[1]: **boolean** [2]: **integer**	**boolean**	[1] sent to operand #3 of cell whose address is in [2]

Character Instruction Cell Opcodes

Opcode	*Operation Name*	*Type of Operands*	*Type of Result*	*Result Produced*
CEQ	Test for equality	[1], [2]: **character**	**boolean**	[1] = [2]
CUEQ	Test for inequality	[1], [2]: **character**	**boolean**	[1] ≠ [2]
CTI	Conversion from character to integer	[1]: **character** [2]: not used	**integer**	**integer**([1])

Integer Instruction Cell Opcodes

Opcode	*Operation Name*	*Type of Operands*	*Type of Result*	*Result Produced*
IADD	Addition	[1], [2]: **integer**	**integer**	[1] + [2]
ISUB	Subtraction	[1], [2]: **integer**	**integer**	[1] − [2]
IMULT	Multiplication	[1], [2]: **integer**	**integer**	[1] * [2]
IDIV	Division	[1], [2]: **integer**	**integer**	[1] ÷ [2]
IMOD	Modulus	[1], [2]: **integer**	**integer**	[1] **mod** [2]
IABS	Absolute value	[1]: **integer** [2]: not used	**integer**	| [1] |
IMIN	Minimum	[1], [2]: **integer**	**integer**	**min**([1],[2])
IMAX	Maximum	[1], [2]: **integer**	**integer**	**max**([1],[2])
IEQ	Test for equality	[1], [2]: **integer**	**boolean**	[1] = [2]
IUEQ	Test for inequality	[1], [2]: **integer**	**boolean**	[1] ≠ [2]
ILT	Test for less than	[1], [2]: **integer**	**boolean**	[1] < [2]
ILE	Test for less than or equal to	[1], [2]: **integer** [1], [2]: **integer**	**boolean**	[1] ≤ [2]
IGT	Test for greater than	[1], [2]: **integer**	**boolean**	[1] > [2]
IGE	Test for greater than or equal to	[1], [2]: **integer**	**boolean**	[1] ≥ [2]
ITR	Conversion from integer to real	[1]: **integer** [2]: not used	**real**	**real**([1])
ITC	Conversion from integer to character	[1]: **integer** [2]: not used	**character**	**character**([1])

Real Instruction Cell Opcodes

Opcode	*Operation Name*	*Type of Operands*	*Type of Result*	*Result Produced*
ADD	Addition	[1], [2]: **real**	**real**	[1] + [2]
SUB	Subtraction	[1], [2]: **real**	**real**	[1] − [2]
MULT	Multiplication	[1], [2]: **real**	**real**	[1] * [2]
DIV	Division	[1], [2]: **real**	**real**	[1] ÷ [2]
ABS	Absolute value	[1]: **real** [2]: not used	**real**	| [1] |
MIN	Minimum	[1], [2]: **real**	**real**	min([1],[2])
MAX	Maximum	[1], [2]: **real**	**real**	max([1],[2])
EQ	Test for equality	[1], [2]: **real**	**boolean**	[1] = [2]
UEQ	Test for inequality	[1], [2]: **real**	**boolean**	[1] ≠ [2]
LT	Test for less than	[1], [2]: **real**	**boolean**	[1] < [2]
LE	Test for less than or equal to	[1], [2]: **real** [1], [2]: **real**	**boolean**	[1] ≤ [2]
GT	Test for greater than	[1], [2]: **real**	**boolean**	[1] > [2]
GE	Test for greater than or equal to	[1], [2]: **real**	**boolean**	[1] ≥ [2]
RTI	Conversion from real to integer	[1]: **real** [2]: not used	**integer**	integer([1])

Untyped Instruction Cell Opcodes

Opcode	*Operation Name*	*Type of Operands*	*Type of Result*	*Result Produced*
DIST ID	Distribute Identity	[1]: any [2]: not used	type of [1]	[1]
MERGE	Merge	[1]: any [2]: any	type of operand used	if [3] = **true** then [1] else [2]
SER	Serializer	[1]: any [2]: any	type of operand used	if [2] is not received or ([1] is received and [3] = **false**) then [1] else [2]; [3] := if [1] is used then **true** else **false**
FAN1	Fan to operand #1	[1]: any [2]: **integer**	type of [1]	[1] sent to operand #1 of cell whose address is in [2]
FAN2	Fan to operand #2	[1]: any [2]: **integer**	type of [1]	[1] sent to operand #2 of cell whose address is in [2]
ACKFAN	Acknowledgment fan	[1]: **integer** [2]: **integer**	none	acknowledgment sent to cell whose address is in [1]
PO	Test for positive overflow	[1]: any [2]: not used	**boolean**	[1] = **pos_over[*]**
NO	Test for negative overflow	[1]: any [2]: not used	**boolean**	[1] = **neg_over[*]**
OVER	Test for overflow	[1]: any [2]: not used	**boolean**	[1] = **pos_over[*]** ∨ [1] = **neg_over[*]**
PU	Test for positive underflow	[1]: any [2]: not used	**boolean**	[1] = **pos_under[*]**
NU	Test for negative underflow	[1]: any [2]: not used	**boolean**	[1] = **neg_under[*]**
UNDER	Test for underflow	[1]: any [2]: not used	**boolean**	[1] = **pos_under[*]** ∨ [1] = **neg_under[*]**
UNKN	Test for unknown	[1]: any [2]: not used	**boolean**	[1] = **unknown[*]**

Opcode	*Operation Name*	*Type of Operands*	*Type of Result*	*Result Produced*
ZD	Test for division by zero	[1]: any [2]: not used	**boolean**	[1] = **zero_divide[*]**
AE	Test for arithmetic error	[1]: any [2]: not used	**boolean**	[1] = **pos_over[*]** ∨ [1] = **neg_over[*]** ∨ [1] = **pos_under[*]** ∨ [1] = **neg_under[*]** ∨ [1] = **unknown[*]** ∨ [1] = **zero_divide[*]**
ME	Test for missing element	[1]: any [2]: not used	**boolean**	[1] = **miss_elt[*]**
UNDEF	Test for undefined	[1]: any [2]: not used	**boolean**	[1] = **undef[*]**
ERR	Test for error	[1]: any [2]: not used	**boolean**	[1] = **pos_over[*]** ∨ [1] = **neg_over[*]** ∨ [1] = **pos_under[*]** ∨ [1] = **neg_under[*]** ∨ [1] = **unknown[*]** ∨ [1] = **zero_divide[*]** ∨ [1] = **miss_elt[*]** ∨ [1] = **undef[*]**
IN	Input	[1]: **integer** [2]: not used	type of value read in	[1] is printed; value is read in from the terminal
OUT	Output	[1]: **integer** [2]: any	none	[1] and [2] are both printed

Appendix 2. Array Operations Implemented Using Forall-Construct

Create/Fill

```
array_fill(LO, HI, V) ≡
        forall i in [LO, HI]
            construct V
        endall
```

Create by Elements

```
[J: V] ≡ array_fill(J, J, V)
```

Append

```
A[J: V] ≡
        let
            lo: integer := array_liml(A);
            hi: integer := array_limh(A)
        in
            forall i in [min(J, lo), max(J, hi)]
            construct
                if i = J then V
                elseif i >= lo & i <= hi then A[i]
                else miss_elt[*]
                endif
            endall
        endlet
```

Set Bounds

```
array_adjust(A, J, K) ≡
        let
            lo: integer := array_liml(A);
            hi: integer := array_limh(A)
        in
            forall i in [J, K]
            construct
                if i >= lo & i <= hi then A[i]
                else miss_elt[*]
                endif
            endall
        endlet
```

Extend Low

```
array_addl(A, V) ≡
        let lo: integer := array_liml(A) − 1
        in
            forall i in [lo, array_limh(A)]
            construct
                if i ~= lo then A[i]
                else V
                endif
            endall
        endlet
```

Extend High

```
array_addh(A, V) ≡
        let hi: integer := array_limh(A) + 1
        in
            forall i in [array_liml(A), hi]
            construct
                if i ~= hi then A[i]
                else V
                endif
            endall
        endlet
```

Remove Low

```
array_reml(A) ≡
        forall i in [array_liml(A) + 1, array_limh(A)]
        construct A[i]
        endall
```

Remove High

```
array_remh(A) ≡
        forall i in [array_liml(A), array_limh(A) − 1]
        construct A[i]
        endall
```

Set Low Limit

```
array_setl(A, J) ≡
        let offset: integer := J − 1
        in
            forall i in [J, offset + array_size(A)]
            construct A[i − offset]
            endall
        endlet
```

Concatenate

```
A || B ≡
        let
            A_hi: integer := array_limh(A);
            B_lo: integer := array_liml(B);
            offset: integer := B_lo − A_hi − 1
        in
            forall i in [array_liml(A), A_hi + array_size(B)]
            construct
                if i <= A_hi then A[i]
                else B[i + offset]
                endif
            endall
        endlet
```

Merge Defined Elements

```
array_join(A, B) ≡
        let
            A_lo: integer := array_liml(A);
            A_hi: integer := array_limh(A);
            B_lo: integer := array_liml(B);
            B_hi: integer := array_limh(B)
        in
            forall i in [min(A_lo, B_lo), max(A_hi, B_hi)]
            construct
                if i >= A_lo & i <= A_hi then
                    let A_ele := A[i]
                    in
                        if i >= B_lo & i <= B_hi then
                            let B_ele := B[i]
                            in
                                if is miss_elt(A_ele) then B_ele
                                elseif is miss_elt(B_ele) then A_ele
                                else miss_elt[*]
                                endif
                            endlet
                        else A_ele
                        endif
                    endlet
                elseif i >= B_lo & i <= B_hi then B[i]
                else miss_elt[*]
                endif
            endall
        endlet
```

References

[1] Ackerman, W. B. and J. B. Dennis. "VAL—A Value-Oriented Algorithmic Language Preliminary Reference Manual." Technical Report 218, Laboratory for Computer Science, MIT, Cambridge, MA, 13 June 1979.

[2] Arvind and R. E. Thomas. "I-Structures: An Efficient Data Type for Functional Languages." Technical Memo 178, Laboratory for Computer Science, MIT, Cambridge, MA, September 1980.

[3] Brock, J. D. and L. B. Montz. "Translation and Optimization of Data Flow Programs." *Proceedings of the 1979 International Conference on Parallel Processing*, August 1979, pp. 46-54. Also Computation Structures Group Memo 181, Laboratory for Computer Science, MIT, Cambridge, MA, July 1979.

[4] Chien, A. "Structuring the Fast Fourier Transform for Data Flow Computation." Computation Structures Group Memo 193, Laboratory for Computer Science, MIT, Cambridge, MA, June 1980.

[5] Dennis, J. B. and D. P. Misunas. "A Preliminary Architecture for a Basic Data-Flow Processor." *The Second Annual Symposium on Computer Architecture Conference Proceedings*, January 1975, pp. 126-132. Also Computation Structures Group Memo 102, Laboratory for Computer Science, MIT, Cambridge, MA, August 1974.

[6] Dennis, J. B., C. K. Leung, and D. P. Misunas. "A Highly Parallel Processor Using a Data Flow Machine Language." Computation Structures Group Memo 134-2, Laboratory for Computer Science, MIT, Cambridge, MA, January 1977 (revised June 1980).

[7] Montz, L. B. "Safety and Optimization Transformations for Data Flow Programs." Technical Report 240, Laboratory for Computer Science, MIT, Cambridge, MA, January 1980.

[8] Liskov, B., *et.al.* "CLU Reference Manual." Technical Report 225, Laboratory for Computer Science, MIT, Cambridge, MA, October 1979.

[9] Stoy, J. E. "Functions in the Form 1 Data Flow Machine." Unpublished communication.

[10] Todd, K. W. "An Interpreter for Instruction Cells." Unpublished communication, January 1981.

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 9/29/1995

M.I.T. Libraries
Document Services