Extensions of Ada for SIMD Parallel Processing

Carolyn Cline
Howard Jay Siegel

Purdue University
School of Electrical Engineering
West Lafayette, IN 47907, USA

Abstract

In order to program SIMD (single instruction stream - multiple data stream) parallel machines used for tasks such as speech and image processing, a language with explicit parallel constructs is often desirable. The language Ada, developed by the Department of Defense, is used here as a basis for such a language. Extensions of Ada which allow the user to specify such things as interprocessor communications and activation of processors are proposed.

1. Introduction

As the speed of integrated circuits approaches a fundamental limit, the need for highly parallel computer systems becomes more and more apparent. Many architectures have been proposed and some machines have been constructed. A highly parallel computer system should have available a language which allows the user to fully exploit the capabilities of the machine.

A number of languages have been proposed and many have been used on actual machines. Languages such as Glypnir [10] and Pfor [5] were used to construct highly complex programs on early machines like the Illiac and PEPE. Many of the early parallel languages relied heavily on the knowledge of the underlying architecture. Later languages, such as DAP Fortran [7] and LRLtran [23], are also tied to the architecture on which they were intended to be implemented. The language Actus [13] was designed to hide the hardware from the user as well as allowing a varying extent of parallelism.

Actus contains many of the features desirable for a general parallel language. It is, however, not suited for processors which can operate on matrices in parallel since only one dimension of an array can be accessed in parallel. The lack of a general method of specifying the movement of data among processors and memory is also apparent in Actus as well as other parallel languages. Much research has been done on the implementation of various forms of networks used to provide communication among processors and memories in large-scale parallel machines. A general parallel language should have a means of specifying the particular type of interprocessor communications desired. To fully explore the capabilities of different networks and to use them efficiently, a user must have direct access to the functioning of the actual network. However, the language must not force the user to be knowledgeable about the particular physical details of the network implementation. Hence, a language should have a flexible and general method of specifying interprocessor communications which hides the details of the network from the user, as well as providing a means of access to the network directly. This allows one to experiment with parallel algorithms for a given architecture, rather than have the architecture hidden.

The lack of a sufficiently general and flexible method of specifying the desired interprocessor communications, as well as the lack of a general specification of multidimensional parallelism, points out the need for a more general parallel language. Most of the existing languages were designed with a particular architecture in mind and therefore designed with the intent of maximizing their performance on one architecture at the expense of generality. A language which does not reflect any particular architecture would be helpful in the study of highly parallel algorithms. Also, a language which allows programming of problems on a general "SIMD" type of parallel architecture would a very useful tool in the design and development of such machines. Such a language should allow specification of problems without particular consideration to the underlying architecture of the machine on which they are to be executed. The language should also reflect the state of the art in conventional language design.

The language proposed as a base for a general parallel language is Ada [1,6]. Many existing parallel languages have used Fortran [7] or Pascal [14] as a base. Fortran provides few structured programming facilities, and has no data structuring facility other than the array. It is also unwieldy when dealing with very complex program control flow. Pascal greatly improved on those problems. However, standard Pascal does not provide a separate compilation facility, thus large programs cannot be split into separate components easily. Also, it does not have arrays whose size is dynamic [9]. Ada has been designed to eliminate problems such as this which occur in existing languages. It is an extremely powerful language which is suited to many different forms of computing. As one of the most advanced languages available it is a good base for a powerful and general language for use on parallel machines. Ada is being used as the standard language by the Department of Defense, and promises to be one of the more widely used languages in the future. The tasking facilities available in Ada also make it suitable for use as a basis for developing a language for use on "multiple-SIMD" parallel architectures.

In extending Ada the structure and design philosophy of Ada has been adhered to as much as possible. However, the instructions and concepts used to make Ada

This material is based on work supported by the National Science Foundation under Grant ECS-8120695. Ada is a registered trademark of the Department of Defense.

0730-3157/83/0000/0366$01.00 © 1983 IEEE

more parallel can also be used with other languages as base languages.

The proposed constructs will be presented in separate sections, one for each construct. Section 2 gives a brief explanation of the general machine model being used. Then, Section 3 discusses the specification of interprocessor communications. In Section 4 the declaration of variables is described. Section 5 explains operators which act collectively on the set of all processing elements. In Sections 6 and 7, statements which allow access to subsets of processing elements, in unconditional and data conditional form, are presented. An example using the constructs proposed is given in Section 8. Section 9 summarizes the constructs proposed.

2. SIMD Model

The general organization of an *SIMD (single instruction stream-multiple data stream)* machine [8] is that of a control unit *(CU)* attached to a number of processing elements *(PEs)* all of which act in parallel (see Fig. 1).

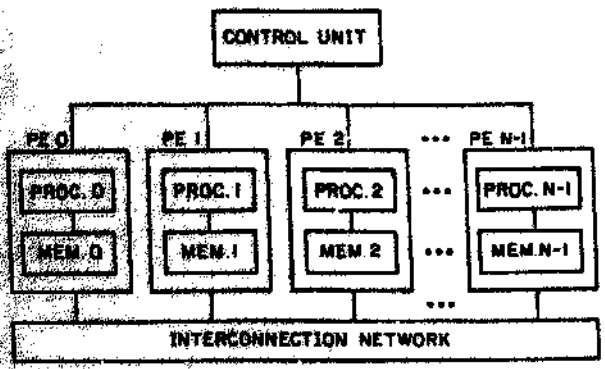

Figure 1. PE-to-PE SIMD machine configuration, with N PEs

There are N PEs, numbered from 0 to N−1. The CU will broadcast instructions to the PEs, and then all active PEs execute the instructions in a lock-step fashion. Each PE consists of a processor and a local memory. The PEs are connected to one another through some form of interconnection network. This network allows the transfer of data among the PEs. There are no assumptions made about the connectivity of the network or its physical operation. Examples of SIMD machines include [4], [2], [5].

In order to program an SIMD machine one needs a language which explicitly includes parallelism in data operations, as well as facilities for specifying other parameters such as which PEs are active or how data is to be transferred. A means of distinguishing between objects to be manipulated by the CU and objects to be manipulated by the PEs is necessary. Access to subsets of the PEs is needed and the representation of the subsets should be both easy to use and of sufficient flexibility to insure that a large number of the possible subsets can be represented. Finally, a convenient and general method for specifying interprocessor communications should be available.

An *MSIMD (multiple-SIMD)* machine is a parallel system which can be structured as one or more independent SIMD machines. The language needs are the same as those for an SIMD machine, except there is also a need for a convenient method of specifying the separate tasks to be performed in each of the separate SIMD machines. For reconfigurable SIMD/MSIMD machines (machines which may be configured as one large SIMD or many smaller SIMD machines), there must be an easy way to program both types of machines within the same basic language framework. Examples of machines that have been proposed that would be able to operate in MSIMD mode include [12], [21], [11].

3. Network Statements

An important component of any SIMD machine is the interconnection network, which allows the transfer of data from PE to PE. Many different types of networks have been proposed, with varying connectivities and varying numbers of switches [17, 18]. A general form of network control, which can be used for any type of network, should be available in an SIMD language. The actual physical implementation of the network should not have to be of concern to the user.

To move data through the network there are two actions to be taken. One, the particular interconnection pattern to be used must be indicated. This is done by specifying the interconnection function representing the pattern, where an *interconnection function "f"* is a bijection on the integers 0 to N−1. The use of interconnection function "f" means that PE i is to send data to PE f(i), $0 \leq i < N$. The other necessary information is which data is to be transferred and where to store it after it has been received.

These actions have often been specified in other languages by the use of a shifting mechanism applied to the indices of an array or by a shift operator applied to a variable [10]. The use of shift and/or rotate as the only interconnection functions seems severely limited. Many other types of functions, such as the cube [15] and the perfect shuffle [23], have been incorporated into the design of networks. A parallel language should allow use of these functions as well. By using two statements to perform the two actions in a transfer, a general specification of many functions is possible. These two statements are the "setnetwork" and the "transfer" statements.

The *setnetwork statement* contains the keyword setnetwork followed by an interconnection function which may have some parameters. For example, to perform a plus one, mod N shift (f(i) = i + 1 mod N)

setnetwork shift(1,N);

would be the appropriate statement. The function "shift" would be found in a standard library of interconnection functions. This type of specification is independent of the underlying physical network structure and implementation. The user need only know the function to be performed and the parameters necessary for that function as long as it is a standard one. For non-standard functions, the user will have to write his/her own machine dependent routine, which will require some knowledge of the actual interconnection network structure.

The *transfer statement* controls which data is sent through the network and where it is stored after being received from the network. For example, to transfer an M element array A in PE i to a similar array B in PE i+1, the setnetwork statement is first used.

```
setnetwork shift(1,N);
```

The transfer statement is then executed by the PEs.

```
transfer (A) into (B);
```

The array is transferred in one statement. The array will be transferred an element at a time with all active PEs transferring in parallel The items in the transfer statement may be arrays, records, variables or two lists of such entities. If lists are used a one to one matching of items in the first list to those in the second list occurs.

The use of two separate statements allows more flexibility in dealing with the network. The network need not be set each time a transfer is done. The network setting will remain the same until another setnetwork statement is encountered. To show how this might be useful consider sending array A in PE i to array C in PE i+2 mod 8. Then array C will be multiplied by a variable B. Lastly PE i will send the new values calculalated for array C to PE i+2 mod 8 which will store it in array D. This would be performed by

```
setnetwork shift(2,8);
transfer (A) into (C);
C:= C * B;
transfer (C) into (D);.
```

The network need only be set once no matter how many transfers follow.

These two statements provide use of a variety of network functions without necessitating knowledege of the actual network. However, one may wish to have access to the actual setting of the network in order to write one's own interconnection function. For this purpose, the two functions "send" and "receive" are proposed. It is assumed that associated with each PE are "in_net" and "out_net" registers. The *send* function loads the data and initiates the transfer. It will load the specified data to be sent through the network into the in_net register of the sending PE. The data will then be sent through the network and placed in out_net register of the recipient PE. The *receive* function will take the data in the out_net register of the receiving PE and assign it to the specified location within that PE.

As an example of how these might be used, consider performing a uniform shift. Assume the network is a multistage cube [20] and that each PE has a register, ADDR, which contains its physical address.

The multistage cube network will use routing tags which determine the path of the data through the network. The routing tag R can be found by performing a bitwise "exclusive-or" of the source address with the destination address. The network hardware interprets these tags to perform the data transfer. Thus, to transfer array F into array G (both of size L) using a uniform shift interconnection function, with 16 PEs (N = 16) the statements are

```
R:= ADDR XOR ((ADDR + 1) mod 16);
for I in  0 .. L-1 loop
send(F,I));
receive(G(I));
end loop;
```

All active PEs set their own R by executing the first statement. The loop control is done by the CU. The send and receive functions are executed by all active PEs. Users can employ the send and receive functions to experiment with the network as well as to construct non-standard interconnection functions.

The intention with respect to the setnetwork and transfer statements is to provide a convenient way in which to represent many different interconnection functions. The representation should be at a high level in order to hide the implementation details from the user. The two statements provide a form of specification which is independent of physical implementation as well as being applicable to a wide variety of functions. The send and receive functions will provide access to the actual working of the network for those who wish to directly control the network.

4. Declaration of Parallel Variables

Generally, scalar operations are performed in the CU and operations which are performed on many data streams are performed in the PEs. Thus there is a need to distinguish between CU objects and PE objects. All variables, arrays, records, etc. will be considered to be CU objects unless specified as PE objects. This is done by qualifying the declaration of a PE object with the prefix PE on the type. The declaration

```
B: PE_INTEGER;
```

would mean each PE has a variable B, in effect, the system has a 1 x N array. The declaration of an M x M PE array of integers would be

```
A: array (INTEGER range 0 .. M-1, INTEGER
     range 0 .. M-1) of PE_INTEGER;
```

with the prefix PE indicating this array is to be stored in the PEs. In this case each PE would have an M x M array of integers, so the system would have N such arrays. In contrast, the declaration of an M x M CU array of integers would be

```
A: array (INTEGER range 0 .. M-1,INTEGER
     range 0 .. M-1) of INTEGER;
```

Since there is no "PE" prefix, this array is to be stored in the CU. This is simply the normal form of an array declaration in Ada.

5. The Any and All Operators

Often there is a need for determining the status of the PEs as a group. For example, if each PE is processing a subfield of a radar image it may be desirable to know "if any" of the PEs has located an enemy missile. The desired operations for testing PE status are "any," "all," and "none." Operators such as these have been used in Pfor [5] and CFD [24]. Not all are needed as primitive operators since they can be combined with the "not" operator (in particular, "not any" is equivalent to "none"). The operators "any" and "all" are similar to the "or" and "and" operators, respectively, which test values of variables and expressions. The difference is that "any" and "all" test the value for all active PEs. Every active PE must have a true expression value for the "all" operator to yield a true value. At least one active PE must have a true expression value for the "any" operator to yield a true value.

The operators will follow the same usage rules as the Boolean operators in Ada except that the expression evaluated must have at least one PE entity in it. The precedence of the "any" and "all" operators will be the same as the Boolean operator "not." Use of parentheses is not required, and is determined only by the construction of the expression. Operators of the same precedence are

evaluated in left to right order. Since "any" and "all" will have a higher precedence than "and," "or," and "xor," constructions such as

any B and A

(where A is a CU variable and B is a PE variable) do not need parentheses. This will test whether "any" of the PEs have a true value for B, then perform an "and" of the result of the test with the CU variable A. Thus the expression will be true only if A is true and there is some B which is true.

6. The Mask Statement

In an SIMD machine, all active PEs will execute the current instruction. Often a specific subset of PEs will be desired rather than all PEs. For example, when doing parallel image correlation [22] the PEs holding the edges of the image will not perform certain operations that the other PEs will perform. A *masking scheme* is a method of specifying which PEs will be active at any given time. Languages such as Glypnir [10] and CFD [24] used a mode vector which contained a bit for each PE. The presence of a 0 or 1 in bit i indicated whether PE i was inactive or active, respectively. So, for example, to have the even numbered PEs active and the odd numbered PEs inactive, for N = 8 PEs, the mode vector would be 01010101. Users were allowed to perform logical operations on the mode vector. For example, the CFD statement

MODE = MODE .TURN OFF. .FIRST. I+2, .LAST. J-1

turns off the first I+2 and the last J-1 PEs. The notation is somewhat unwieldy. Also, for machines with large numbers of PEs (for example 1024) the manipulation and specification of a mode vector becomes quite inconvenient. The mode vector does have the advantage that a single mask can activate any subset of the PEs. Some form of masking which allows a large choice of subsets of PEs (that are typically needed in SIMD processing) to be activated by a single mask, while maintaining computational convenience, is desired.

The *PE address masking scheme* introduced in [15], is a flexible means of specifying subsets of N PEs in m = $\log_2(N)$ bits. Each position of the mask corresponds to a bit position in the binary addresses of the PEs. Each position will contain either a 0, a 1, or an X. The active processors will be those whose address matches the mask in each position, with either 0 or 1 matching X. For example, for N = 8, [X10] will match PEs 2 and 6.

These masks can then be used in an *enable statement* to specify which PEs should be active. Repetition factors enclosed in parentheses cause the preceding factor to be repeated within the mask specification. So, for example, if N = 16 the statement **enable**[X(m-1),0]; is equivalent to **enable**[X2X0]; Each specifies that all even numbered PEs should be active (and all odd PEs inactive).

Negative PE address mask statements are included to allow activation of more subsets of PEs [15]. These are similar to the regular mask statements above, but will activate those PEs which do not match the mask. So, for example, **disable**[0XX]; will activate all PEs except PE 0. This cannot be done with a positive mask statement.

A mask statement will have a scope which ranges over the block in which it occurs until another mask statement occurs within that block. Each mask statement will be decoded to form an N-bit vector, and a one in position i of this vector will indicate that PE i is enabled. One method for keeping track of the status of the PEs is by means of a run-time mask stack. When a block is entered the active/inactive status of the block is determined by the current top of stack, T, so a copy of the top of stack is pushed (i. e., the top two elements of the stack are T and T.) Any mask statements occurring within a block affect only those PEs which were active when the block was entered. Those PEs that are inactive upon entrance to a block will remain so throughout execution of the block. Thus, when an enable or disable statement is encountered the top of the stack is popped and the mask is logically "anded" with the new top of stack (T). The result is pushed onto the stack. Similarly the current top of stack is pushed on entry to a subprogram and popped upon return. Upon exit from the block, the current top of stack is popped, restoring T to the top of stack. If a block contains a sub-block the procedure is repeated.

As an example, assume that prior to entry to a block only the even PEs (N = 8) are enabled. Then the statements

```
enable[0XX];
statement_1;
enable[X0X];
statement_2;
```

will cause only PEs 0 and 2 to be active for statement_1 and PEs 0 and 4 to be active for statement_2. The previous status of the PEs (even PEs enabled) is restored upon exit from the block. The values of the stack for this example are shown in Fig. 2. The bit vectors shown indicate the active/inactive status of the PEs, with bit i corresponding to PE i.

7. The Where Statement

The PE address masking scheme is limited in that masks cannot be specified as a function of each PE's local data. Another type of masking is based on the individual

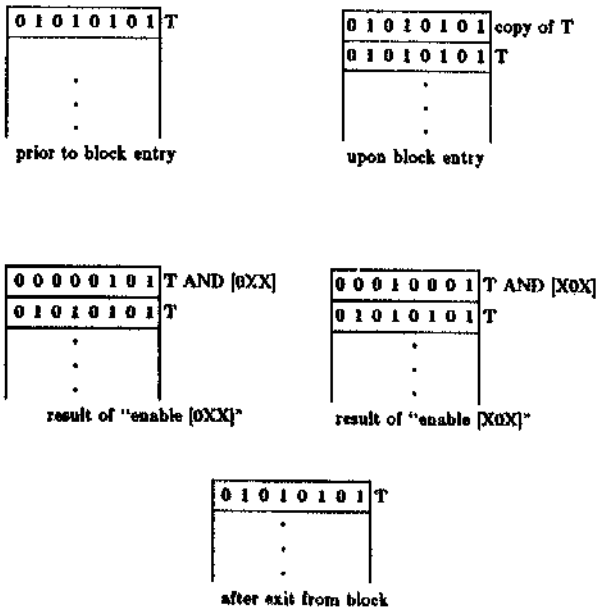

Figure 2. Example of mask stack

values of variables within each PE. The active/inactive status of the PEs is determined by the PE's local data, possibly in conjunction with CU variables. Such a conditional mask can be implemented by a *where statement* which has the form

where condition **do** statements;
elsewhere statements;
endwhere;

The elsewhere portion of the statement is optional.

The where statement will divide the currently active PEs into two disjoint sets, each of which will be active for either the "do" section or the "elsewhere" section but not both (PEs inactive prior to execution of the where statement will remain inactive, as discussed below). The "do" section will be executed first in the appropriate PEs and then the "elsewhere" section will be executed. They are not executed simultaneously. This is because the "do" and "elsewhere" code must be broadcast sequentially since there is a single instruction stream. Where statements, while seemingly similar, perform an action quite different from that of a traditional if statement. An if statement performs either the "then" or the "else" section in all active PEs. A where statement can perform both the "do" and the "elsewhere" sections, but in different sets of PEs.

Where statements can be used in conjunction with the PE address mask statements and may also be nested. One way to implement this is by again using a run-time stack to maintain the status of the PEs. When a "where" is encountered, the top of stack, T, is logically "anded" with the negation of the result of the expression evaluation for each PE. The result of the "and" operation is pushed onto the stack on top of T. Then the result of the expression evaluation "anded" with the previous top of stack (T) is pushed once onto the stack. The "do" portion is then handled in the same way as described for a block in the previous section. When the "elsewhere" statement is encountered one pop is performed to remove the value pushed on the stack for the "do" section. The "elsewhere" is also handled like a block. When the "endwhere" is encountered again one pop will be performed to restore the status of the PEs in effect before the where statement was executed.

The stack is a simple and efficient way to keep track of the status of the PEs. Use of both PE address masks and data conditional masks allows a great deal of flexibility in accessing subsets of the PEs. Both types of masks can be implemented quite readily through use of the stack.

6. FFT Example

As an example of how the constructs presented might be used, consider a parallel FFT algorithm [19] which uses the radix two decimation-in-frequency technique. The input sequence $\{s(m)\}$ is divided into two halves, $\{s_1(m')\}$ and $\{s_2(m')\}$. The M-point DFT can then be calculated by computing the two M/2-point DFTs of $\{s_1(m') + s_2(m')\}$ and $\{[s_1(m') - s_2(m')]\, W_M^{m'}\}$, where $0 \le m' < M/2$ and W_M is the "twiddle" factor. For example, Fig. 3 shows the computations necessary for a 16-point FFT.

In order to perform the FFT on an SIMD machine the "twiddle" factors are assumed to be pre-computed and stored in the appropriate PEs. PE i initially has s(i) stored in the A register and s(i + M/2) stored in the C register. Each PE will also have its physical address stored in ADDR. S and Y are variables used to store the

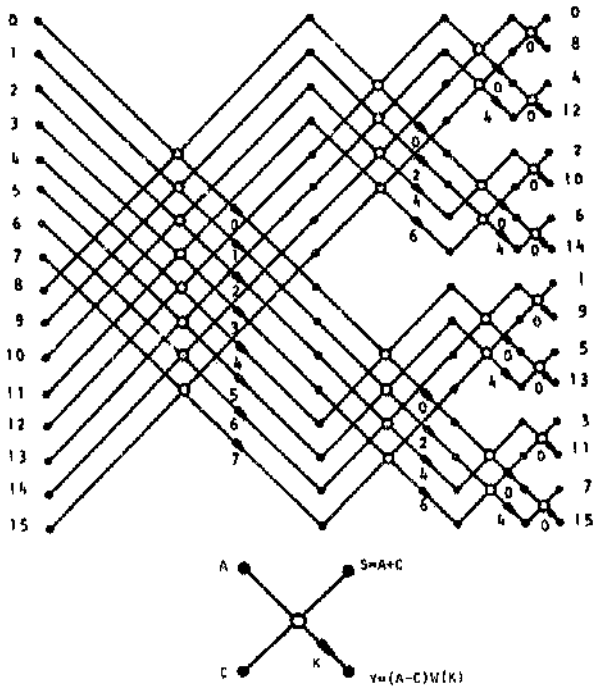

Figure 3. Computation of a 16 point FFT

results of each "butterfly" operation (see Fig. 3). TEMP is used in the transfer operations. The interconnection functions will be the cube functions [15], defined by $\text{cube}_i(p_{n-1}\cdots p_{i+1}p_ip_{i-1}\cdots p_0) = (p_{n-1}\cdots p_{i+1}\bar{p}_ip_{i-1}\cdots p_0)$, where $p_{n-1}\cdots p_1p_0$ is the binary representation of P, $0 \le P < N$, and $0 \le i < n$. The data transfers necessary with such functions for a 16-point FFT are shown in Fig. 4.

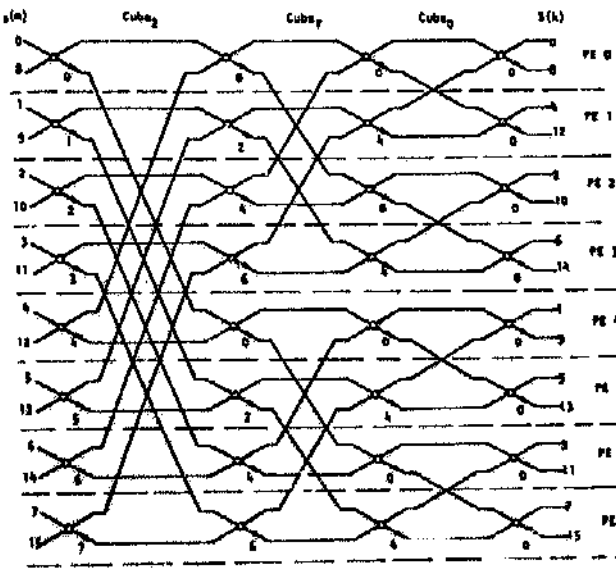

Figure 4. Data transfer for 16 point FFT

```
I : INTEGER := 0;
Z : INTEGER := LOG2(M) - 1;
K : PE_INTEGER range 0..M/2 - 1;
S,Y,A,C,TEMP : PE_FLOAT;
K := ADDR;
S := A + C;
Y := (A - C) * W(K);
for I in 1 .. Z loop
        enable [X(I-1), 0, X(Z-I)];
        A := S;
        TEMP := Y;
        enable [X(I-1), 1, X(Z-I)];
        C := Y;
        TEMP := S;
        enable [X(Z)];
        setnetwork cube(Z-I);
        transfer (TEMP) into (TEMP);
        enable [X(I-1), 0, X(Z-I)];
        C := TEMP;
        enable [X(I-1), 1, X(Z-I)];
        A := TEMP;
        enable [X(Z)];
        K := (ADDR * (2**I)) mod M/2;
        S := A + C;
        Y := (A - C) * W(K);
end loop;
```

Figure 5. Statements for 16 point FFT

An algorithm to perform the FFT is shown in Fig. 5. The CU variable Z controls the loop which constitutes the main part of the algorithm. In each iteration the "butterfly" operation is performed and the results are stored in the appropriate variables for use by the next stage (one result is saved locally and one must be sent through the network). After the last step the computed FFT values are stored in the PEs in bit reverse order.

To demonstrate how the algorithm operates, consider the statements associated with the last iteration of the loop. These are shown in Fig. 6, along with a drawing of the "butterfly" and transfer operations for PEs 0 and 1. At this stage in the algorithm, I = 3 and there is one last transfer stage followed by the last "butterfly" stage. The first enable statement in the loop will activate PEs 0, 2, 4, and 6 and they will save their S data locally in A and put their Y data in TEMP in preparation for the transfer. Next PEs 1, 3, 5, and 7 will be activated and they will save their Y data locally in C and put their S data in TEMP. Then all PEs are enabled and the network is set to a $cube_0$ function. All PEs transfer their TEMP data, which then needs to be stored in the appropriate variable in each PE. To do this, the even PEs are activated by an enable statement and TEMP is stored in C (this is S from the previous stage). The next enable statement activates the odd PEs, which then store TEMP in A (this is Y from the previous stage). Finally, the last butterfly is computed and the computation of the FFT is complete.

9. Conclusions

A minimal set of features which make Ada suitable for use with SIMD type architectures has been presented. Many of the features are present in existing parallel languages. However, the use of a general method of specifying interprocessor communications is not found in other languages. Also, the use of machine independent constructs makes the language specified a more general language than existing parallel languages. The intent is not to propose a language for a particular architecture but rather one which is of use in the specification of problems for many different architectures. The language is intended to be applicable to both varying SIMD architectures and varying SIMD algorithms. The intention of Ada to be portable, readable and widely applicable is also that of the parallel language presented here. However, the features presented are not expressly tied to the use of Ada as a base, and could be incorporated into other languages.

```
enable [XX0];
A := S;
TEMP := Y;
enable [XX1];
C := Y;
TEMP := S;
enable [XXX];
setnetwork cube(0);
transfer (TEMP) into (TEMP);
enable [XX0];
C := TEMP;
enable [XX1];
A := TEMP;
enable [XXX];
K := (ADDR * (2**3)) mod 8;
S := A + C;
Y := (A - C) * W(K);
```

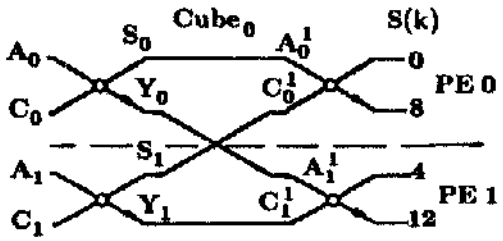

Figure 6. Statements for final iteration of loop

10. References

[1] J. G. P. Barnes, *Programming in Ada*, Addison-Wesley Publishing Co., London, 1982.

[2] K. E. Batcher, "STARAN Series E," *1977 Int'l. Conf. Parallel Processing*, pp. 144-153, Aug. 1977.

[3] K. E. Batcher, "Bit serial parallel processing systems," *IEEE Trans. Computers*, vol. C-31, pp. 377-384, May 1982.

[4] W. J. Bouknight, S. A. Denenberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, and D. L. Slotnick, "The Illiac IV system," *Proc. IEEE*, vol. 60, pp. 369-388, Apr. 1972.

[5] J. R. Dingeldine, H. G. Martin, and W. M. Patterson, "Support and operating system software for PEPE," *1973 Sagamore Computer Conf. Parallel Processing*, Aug. 1973.

[6] U. S. Department of Defense, *Reference Manual for the Ada Programming Language*, Washington, D. C., Jul. 1982.

[7] P. M. Flanagan, D. J. Hunt, S. F. Reddaway, and D. Parkinson, "Efficient high speed computing with the distributed array processor," in *High*

Speed Computer and Algorithm Organization, edited by D. J. Kuck, D. H. Lawrie, and A. H. Sameh, Academic Press, New York, pp. 113-127, 1977.

[8] M. J. Flynn, "Very high speed computing systems," *Proc. IEEE*, vol. 54, pp. 1901-1901, Dec. 1966.

[9] K. Jensen and N. Wirth, *PASCAL User Manual and Report*, Springer-Verlag, Berlin, 1974.

[10] D. H. Lawrie, T. Layman, D. Baer, and J. M. Randal, "Glypnir - a programming language for Illiac IV," *Comm. ACM*, vol. 18, pp. 157-164, Mar. 1975.

[11] G. J. Lipovski and A. Tripathi, "A reconfigurable varistructure array processor," *1977 Int'l. Conf. Parallel Processing*, pp. 125-138, Aug. 1977.

[12] G. J. Nutt, "Microprocessor implementation of a parallel processor," *Proc. 4th Ann. Symp. Computer Architecture*, pp. 147-152, Mar. 1977.

[13] R. K. Perrott, "A language for array and vector processors," *ACM Trans. Programming Languages and Systems*, vol. 1, pp. 177-196, Oct. 1979.

[14] A. P. Reeves, J. D. Bruner, and M. S. Poret, "The programming language parallel Pascal," *1980 Int'l. Conf. Parallel Processing*, pp. 5-7, Aug. 1980.

[15] H. J. Siegel, "Analysis techniques for SIMD machine interconnection networks and the effects of processor address masks," *IEEE Trans. Computers*, vol. C-26, pp. 153-161, Feb. 1977.

[16] H. J. Siegel, "Controlling the active/inactive status of SIMD machine processors," *1977 Int'l. Conf. Parallel Processing*, p. 183, Aug. 1977.

[17] H. J. Siegel, "Interconnection networks for SIMD machines," *Computer*, vol. 12, pp. 57-65, June 1979.

[18] H. J. Siegel, *Interconnection Networks for Large Scale Parallel Processing: Theory and Case Studies*, D. C. Heath and Co., Lexington, Mass., 1983.

[19] L. J. Siegel, P. T. Mueller, and H. J. Siegel, "FFT algorithms for SIMD machines," *17th Ann. Allerton Conf. Communication, Control, and Computing*, pp. 1006-1014, Oct. 1979.

[20] H. J. Siegel and R. J. McMillen, "The multistage cube: A versatile interconnection network," *Computer*, vol. 14, pp. 65-76, Dec. 1981.

[21] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Computers*, vol C-30, ppp. 934-947, Dec. 1981.

[22] L. J. Siegel, H. J. Siegel, and A. E. Feather, "Parallel processing approaches to image correlation," *IEEE Trans. Computers*, vol. C-31, pp. 806-816, Mar. 1982.

[23] H. S. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Computers*, vol. C-20, pp. 153-161, Feb. 1971.

[24] K. G. Stevens, Jr., "CFD - A Fortran-like language for the Illiac IV," *ACM Sigplan Notices (Proc. Conf. Programming Languages and Compilers for Parallel and Vector Machines)*, pp. 72-76, Mar. 1975.

[25] R. G. Zwakenberg, "Vector extensions to LRLtran," *ACM Sigplan Notices*, vol. 10, pp. 77-86, Mar. 1975.