Presented @ the Conf. on Information Sciences and Systems @Johns Hopkins, Mar. 1978.

# A DATA DRIVEN COMPUTER ARCHITECTURE

T.N. Mudge

Systems Engineering Laboratory
Department of Electrical and Computer Engineering
and CICE Program
The University of Michigan
Ann Arbor 48109

## ABSTRACT

This paper describes the initial phase of a study to determine the requirements of a computer architecture that exploits potential parallelism present in its data flow. The architecture is based on the single assignment rule (each variable in a program may be assigned only once). This has been shown to result in maximal parallelism. Unfortunately, several obstacles need to be overcome to make single assignment architectures practical. These are: implementing iterative procedures (e.g. DO loops) in the face of the single assignment rule, and coping with possible unbounded memory requirements. This paper outlines a technique for overcoming these obstacles by using recursion. To take advantage of the maximal parallelism requires a high bandwidth memory and multiple instruction set processors. A result-forwarding scheme between the instruction set processors is described which reduces the bandwidth requirements of the memory.

## 1. INTRODUCTION

This paper describes the initial phase of a study to determine the requirements of a computer architecture that exploits potential parallelism present in its data flow [M]. The architecture supports a base language that is a single assignment language.

The first single assignment language (SAL) was proposed by Tesler and Enea in [T]. A program written in a SAL must obey the following rule:

No variable is assigned values by more than one statement.

Their motivation for proposing a SAL was twofold. First, they wanted a language suitable for concurrent processes. Second, they wanted a language in which the sequencing of statements was handled in a unified manner. These two requirements are met by a SAL, since, in a SAL program, statements are available for execution as soon as their operands are ready. This means that the implicit parallelism in the data flow can be exploited, and that statement sequencing is unified (it is based solely on operand availability). Based on their work, Chamberlin [C] produced a compiler for a SAL called SAMPLE. This work brought out the following practical problems: possible unbounded memory requirements, and how to construct an efficient algorithm to select those statements that are ready for execution (ideally this requires an associative search). A group at C.E.R.T. in Toulouse, France is presently constructing a machine based on a SAL [S], called the LAU system. Two other proposals for SALs are given in [I] and [K1].

Figure 1 shows a SAL program using a three-address instruction format. The associated precedence graph is shown alongside. The SAL automatically induces this precedence by the order in which statements become ready for execution, as determined by the availability of their operands.

```
1. t1:=F(a,c)
2. t2:=G(t1,b)
3. t3:=F(t2,c)
4. t4:=F(t3,c)
5. t5:=F(d,c)
6. t6:=G(t5,e)
7. t7:=G(t6,t4)
```

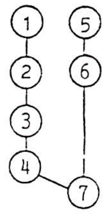

Figure 1. A SAL Program.

The choice of an architecture based on a SAL has been made because, as is shown in [U], SALs yield programs that are maximally parallel. A program has this property if the order in which statements are made available for execution is determined solely by the necessary data dependencies. A property of maximal parallelism is that the parallelism of a program may be unbounded. An example of this would be a SAL procedure to compute the following vector function:

$$H(N) = \langle A(1),A(2),\ldots,A(N)\rangle$$

where the computations for A(j) and A(k) are independent if $j\neq k$. Each of the components A(i) is computed concurrently, thus the degree of parallelism is proportional to N, a parameter of the procedure. This type of dynamic parallelism is, as we shall see below, produced by SALs, but is not present in systems based on more popular

This work was supported in part by the National Science Foundation under Grant. No. MCS 77-03310.

static models of parallelism such as Petri nets [Pe] (for an example of such systems see [Pa]), or parallel program schemata [Ka] (for an example of such systems see [R]).

By basing our architecture on a SAL we can be assured of maximal parallelism in the data flow of any target system. However, this should not be confused with maximum functional parallelism. To illustrate the distinction, consider the program of Figure 1 with F(x,y) interpreted as x*y and G(x,y) as x+y. A functionally equivalent program is shown in Figure 2. Its associated precedence graph shows a greater degree of parallelism. This increase in parallelism is possible because the programmer can make use of properties about addition and multiplication (such as associativity, commutativity, distributivity, etc.) to restructure the program of Figure 1 to yield one having a greater degree of parallelism. In general, automatic detection of functional parallelism is undecidable [B], but in many cases of practical interest it can be achieved. (Considerable research in this area has been done by Kuck - for a survey see [Ku]).

Paradoxically then, detecting the maximum amount of parallelism possible in a program in which the functions are uninterpreted (maximal parallelism) is a decidable question and architectures to achieve this end are a possibility, but detecting the maximum amount of parallelism in the case of interpreted functions is undecidable. Thus we have limited our architecture to the automatic detection of maximal parallelism. Any further speed-up made possible by functional properties is more conveniently delegated to a compiler optimizer.

```
1. t1:=F(a,c)
2. t2:=G(t1,b)
3. t3:=F(c,c)
4. t4:=F(t2,t3)
5. t5:=F(c,d)
6. t6:=G(t5,e)
7. t7:=G(t4,t6)
```

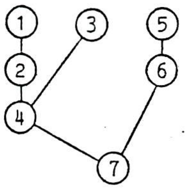

Figure 2. A Functionally Equivalent Program.

Although an architecture based on a SAL gives maximal parallelism, several problems arise as a consequence of using a SAL. The first problem is how to implement iterative procedures without violating the single assignment rule. A method for solving this, using recursion, is presented in Section 2.1. It is an adaption of an algorithm first presented in [U]. The second problem (first recognized in [C]), is how to cope with the possibly unbounded memory requirements of a SAL during its execution. A method for alleviating this problem by dynamically deallocating no longer needed memory space, during program execution, is also presented in Section 2.1. This method results from our use of recursion. The third problem (also first recognized in [C]) is how to detect and select those statements that are ready for execution. A method for handling this problem, based on a result-forwarding scheme between multiple instruction set processors, described in [Ke], is presented in Section 2.2

Achieving performance gains by taking advantage of the parallelism at the instruction set processor (ISP) level has been questioned by Amdahl in [Am]. However, experimental results presented in [Pl] for the LAU system are encouraging, even though this architecture does not take advantage of all parallelism that occurs dynamically during program execution. Furthermore, there is some evidence that proving correctness of SAL programs is easier than conventional programs (at least in the case of determinate programs - see [As]). Finally, the term "data driven" derives from the fact that SALs are sequenced solely by the flow of data - the operands of the instructions

## 2. PRELIMINARY OUTLINE OF THE ARCHITECTURE

### 2.1 The Base Language

Figure 3 illustrate how an iterative program in the single assignment base language might look. The one shown forms a vector of three elements from the unspecified element-by-element dyadic operation "o" on two other vectors of three elements. At the bottom of the figure is a similar FORTRAN program, which uses a DO loop. The syntax of the SAL is not included in this paper, as it is by no means finalized. We shall explain, as they occur, those concepts that are important to this discussion.

```
∇F(&A:A&,&B,&C|&I≤&N)
  (&A):=(&B) o (&C)
  &AA:=&A + 1
  &BB:=&B + 1
  &CC:=&C + 1
  &J :=&I + 1
  F(&AA:A&,&BB,&CC|&J≤&N)∇
□I:= 1
  N:= 3
  AO:= 200
  BO:= 300
  CO:= 400
  F(AO:AF,BO,CO|I≤N) □

                    --------

    DO 10 I=1,3
 10 A[I]=B[I]oC[I]
```

Figure 3. A Program in the Base Language.

In the example of Figure 3 the main body of the program is enclosed between two □s. It is a six statement program. The first five statements assign constants to variables I, N, AO, BO and CO. The sixth statement calls a procedure F with input variable AO, BO, CO, I and N, and an output variable AF. Notice that no data type declarations have been included - for the moment it is sufficient to assume that variable names refer to integers. The output variable AF is paired with AO to indicate that its value is returned by F, and that it is logically related to AO in the following way. During the execution of a conventional program a variable, say V, will take on a sequence of values. This cannot occur in a SAL as each variable may be assigned at most once. In a SAL, V would correspond to a sequence of linked variables whose values correspond to the past values of V. Such a list is created by the recursive calling of F in the example of Figure 3. AO corresponds to the first item in the list and AF, after F has been applied, corresponds to one after the last. (Similar output variables could have been included for BO, CO, I and N). The binary relation (I≤N) to the right of the vertical line, used in the calling specification for F, is interpreted as a predicate. A procedure call may be invoked as soon as all the input variables are assigned. If the predicate is true the call proceeds, if not, the values of the output variables are assigned the values of their corresponding input variables and the the call returns.

The definition of F is given before the main program. It is enclosed between two ∇s. The definition is made in terms of dummy variables (identified by their ampersand prefix). Variables enclosed in parentheses are pointers (i.e, one level of indirection is allowed), and the format of the assignment statements is three-address. It can be seen that the language is essentially at the assembly language level. Only two types of statement occur: assignment statements and procedure calls. Both types are considered ready for execution as soon as their operand/input variables have been computed. When F is called, the variables used in the call are substituted for the corresponding dummy ones in the definition of F. Those dummy variables that are not defined in the call (such as &AA, &BB, &CC and &J in Figure 3) are assigned memory locations (addresses) from a list of available memory locations (called avail hereafter). In the case of F, since it calls itself (last statement in the definition of F), new copies of all the statements in F are repeatedly made with new variable names until the predicate is no longer true.

The result of the procedure call F(AO:AF,BO,CO|I≤N) is shown in Figure 4. The statement shown immediately after the header for F in Figure 3 is similar to the FORTRAN statement A[I]=B[I]oC[I]. (Since the SAL has no indexing capability pointers are used.) Three copies of this are created by the procedure call (see Figure 4), each with a new set of pointers. Before each copy is created the new pointer variables (a,b,c for the first copy, e,f,g for the second, etc.) must be obtained from avail and assigned values. Thus the sequence of variables <(AO),(a),(e)> represents a pointer moving along the list of three elements in the result vector. AO is the start address of the vector and AF (=i=e+1) is the address of the next location after the last element of the vector. Similarly <(BO),(b),(f)> and <(CO),(c),(g)> represent the pointer values for the two vectors from which the result is formed.

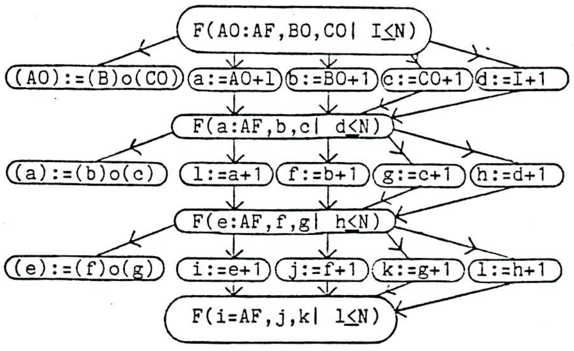

Figure 4. The Effect of Calling F.

The number of copies of F is controlled by the sequence <I,d,h,l>. The value of l is 4, for which the predicate is false, thus three copies of F are made and AF is assigned a value (=AO+3) by the last one. The sequence <I,d,h,l> corresponds to the successive values taken on by I, the loop counter, in the corresponding FORTRAN program. Since all the elements of this sequence and the elements of those sequences corresponding to the pointers are concurrently available in the SAL program (at least, after they have all been formed), the three major computation statements, viz:

```
(AO):=(BO)o(CO)
(a):=(b)o(c)
(e):=(f)o(g)
```

can be performed concurrently (see Figure 4), unlike their counterparts that compute A[1],A[2] and A[3] in the FORTRAN program. Of course the concurrency is not ideal, as copies of the major computation statements are not made available simultaneously - pointer and loop counter variables need to be created and assigned first. If this housekeeping can be accomplished efficiently and the unspecified operation "o" is time consuming, the whole process has a high degree of parallelism approximating that of a DO TOGETHER type of command. If the major computations in a recursive procedure call are data dependent on their predecessors in the recursive calling sequence (consider a program to compute $\sqrt{X}$ using Newton's method) little parallelism can be expected. This is consistent with maximal parallelism. Speeding up such programs requires knowledge about the functions being computed (functional parallelism). Between these two extremes of data dependency in iterative programs - from programs in which each iteration is independent of its predecessor, to those in which each iteration depends on its predecessor - lies a whole spectrum of dependencies between iterations. By treating iteration recursively, as illustrated above, in connection with a SAL, these varying degrees of parallelism can be exposed automatically. Nevertheless, some inefficiencies do result from using a SAL in the way proposed here. For example, modifying one element in a vector requires renaming all the elements.

If a program is of the DO FOREVER type (e.g. an operating system) successful operation requires an infinite avail list. This is clearly impossible. To overcome this, no longer needed memory locations are recognized and placed on avail by a garbage collector. This is accomplished as follows. A search through a SAL program text, as a preprocessing step, allows a count to be assigned to each variable and dummy variable. The count corresponds to the number of times the variable appears on the right hand side of assignment statements plus the number of times it occurs as an input variable to a procedure call (i.e. the count indicates how many times the variable is read). When the program is run each read of a variable decrements its corresponding count by one. When that count reaches zero the variable is no longer needed, and the garbage collector may return it to avail. Note that A and (A) are treated as distinct variable names, and that the preprocessing step required to determine the value of the count for each variable is an algorithm whose complexity increases only linearly as the number of statements in the program. Whenever dummy variables are assigned a variable name their associated count is added to the count associated with that variable. The count associated with each dummy variable never changes.

To assure that the single assignment rule is obeyed it is necessary to check a SAL program for the following two things. First, check to see that each variable that is the result of an assignment statement or the output variable of a procedure call occurs at most once. Second, define a binary relation d, the dependence relation, between two program variables such that AdB if A:=F(B,X) or A:=G(Y,B) for all F,G,X and Y; or if A is the output variable of a function call to which B is an input variable. Represent d as a directed graph and check to see that it is circuit-free. Recall that A and (A) are to be treated as distinct variables. The problem of detecting whether two pointers (A) and (B) point to the same location, and thus permit the possibility of violating the single assignment rule by both being the destination of an assignment statement, is left open. (Detection at execution time hardly seems satisfactory.)

## 2.2 The Related Machine Organization

Figure 5 shows a block diagram of the machine organization to support the SAL outlined in the previous section. Program text is stored in IM, the instruction memory, along with the read counts of the dummy variables in the procedure definitions. Instructions are read in the order in which they occur in the program text by IMM, the instruction memory manager. These are sent on to IQ, the instruction queue. In the case of procedure calls IMM replaces the dummy variables by the variables in the procedure call. Those dummy variables not specified by the call are replaced by addresses from avail. Avail is managed by GC, the garbage collector. Avail is reprsented as a linked list in CM, the count memory. GC contains a pointer to the top of this list. Each word in CM has a countfield and a linkfield. If the countfield=0, the word should be in avail. The linkfield is used in this case to link the word to the list avail. The words in CM are in one-to-one correspondence with those in DM, the data memory. The count at location m corresponds to the number of reads left for the data item at location m in DM. When a variable is read by an ISP it signals GC which decrements the countfield of the corresponding word in CM. If the countfield becomes zero, GC places the pointer to avail into the linkfield and updates the pointer to avail with the address of the variable. In the case of variables created or used in procedure calls, certain countfields must be added to. Information about this comes from IMM and is used to select the appropriate word from CM and add to its countfield using the adder/subtracter associated with CM.

Instructions are issued from the IQ to the idle ISPs. When an instruction reaches an ISP, the ISP looks for the two operands. First it checks RFT, the result-forwarding table. Failing there it checks DM. Unless both operands are found the instruction is recycled through the merge module back into IQ. This recycling is possible because in a SAL the order in which statements are presented for execution is unimportant - availability of the operands guarantees a correct order of execution. If an ISP can execute an instruction, i.e., if it can find both operands, it does several things. It indicates to GC those variables whose countfields must be decremented. Concurrently, it initiates the instruction execution. Also concurrently, it enters into an empty location in RFT the name (or address in DM) of its result variable together with its number. When an instruction first gets issued to an ISP, say ISP k, that ISP checks for its operands in the RFT as follows. It does an associative search, through RFT, based on the address of each operand. If it fails it goes to DM (see above). If it succeeds it retrieves the ISP number stored with the operand address. This number denotes the ISP that is working on computing that variable. The number k is then sent to that

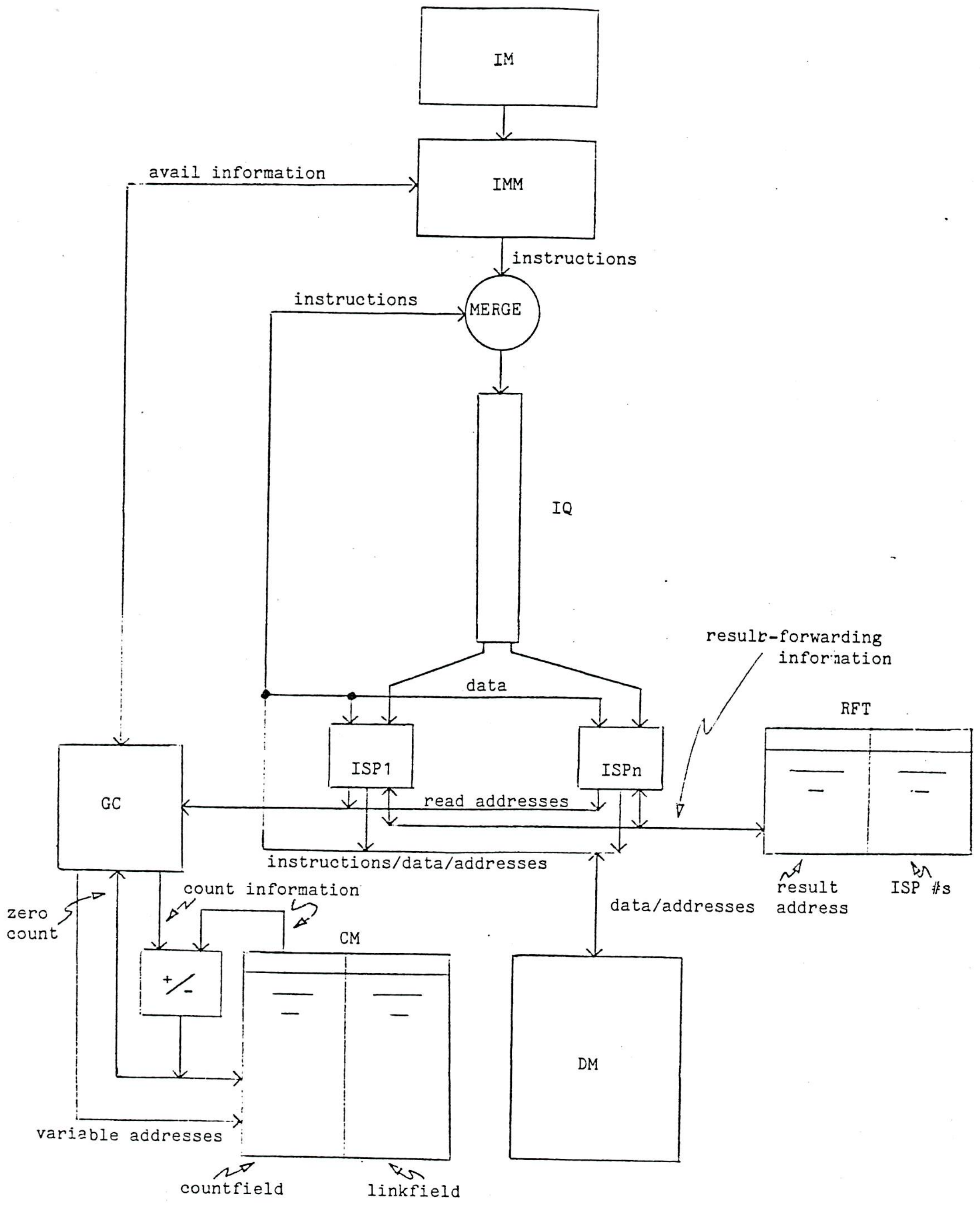

Figure 5. Machine Organization.

ISP where it is held in a result-forwarding list. When the ISP has computed the variable it sends the result to all those ISPs whose numbers are in its result forwarding list. It also writes through to DM and sets to "empty" the location in RFT corresponding to the computed variable.

The central idea of result-forwarding is to reduce the bottleneck at DM, by reducing the need to fetch operands from DM. Using multiple DMs is another, more costly solution. For result-forwarding to work effectively it requires a certain amount of locality in the program. A subset of the ISPs may be virtual, i.e. just reservation stations. Also a deadlock can occur in which instructions continue to recycle through IQ without any of them being serviced. The main design parameters are l, the length of IQ, n, the number of ISPs and m, the subset that is virtual.

## 3. CONCLUDING REMARKS

The above architecture would be ideal for supporting a non-procedural language such as LUCID (see[As]). However, for architectures based on SALs there is, as yet, no method for expressing programs that are intentionally non-determinate (consider the problem of an airline reservation system). To have a complete system this deficiency needs to be removed.

I would like to thank Professor Keki Irani for his helpful comments and suggestions during the writing of this paper.

## 4. REFERENCES

[Am] Amdahl, G. M., "Validity of a Single Processor Approach to Achieving Large Scale Computing Capabilities", AFIPS Conf. Proc., Vol. 30, pp. 483-485, SJCC, 1967.

[As] Ashcroft, E. A., and W. W. Wadge, "Lucid, a Nonprocedural Language with Iteration", CACM, Vol. 20, No. 7, pp. 519-526, July 1977.

[B] Bernstein, A. J., "Analysis of Programs for Parallel Processing", IEEE TC, EC-15, No. 5, pp. 757-763, October 1966.

[C] Chamberlin, D. D., "Parallel Implementation of a Single Assignment Language", Ph.D. Thesis, Stanford University, 1971.

[I] Irani, K. B., "A Proposal for a Programming Language for Parallel Processing Environments", Rept. to Rome Air Development Center, Contract No. F30602-73-C-0001, January 1975.

[Ka] Karp, R. M., and R. E. Miller, "Parallel Program Schemata", J. of Comp. and System Sciences, Vol. 13, No. 2, pp. 147-195, May 1969.

[Ke] Keller, R. M., "Look-Ahead Processor", Computing Surveys, Vol. 7, No. 4, pp. 177-195, December 1975.

[Kl] Klinkhammer, J. F., "A Definitional Language", Phillips Research Laboratories, Eindhoven, the Netherlands.

[Ku] Kuck, D. J., "A Survey of Parallel Machine Organization and Programming", Computing Surveys, Vol. 9, No. 1, pp. 29-59, March 1977.

[M] Mudge, T. N., "A Data Driven Computer Architecture", SEL Report No. 117, University of Michigan, January 1978.

[Pa] Patil, S. S., and J. B. Dennis, "The Description and Realization of Digital Systems", COMPCON 72, Proc. IEEE Comp. Conf., pp. 313-316, September 1972.

[Pe] Petri, C. A., Communication with Automata, Suppl. 1 to Tech. Report RADC-TR-65-377, Vol. 1, Rome Air Development Center, New York, 1966.

[Pl] Plas, A., et al., "LAU System Architecture: A Parallel Data-Driven Processor Based on Single Assignment", Proc. 1976 International Conf. on Parallel Processing, pp. 293-302, August 1976.

[R] Rumbaugh, J., "A Data Flow Multiprocessor", IEEE TC, C-26, No. 2, pp. 138-146, February 1977.

[S] Syre, J. C. et al., "Pipelining, Parallelism and Asynchronism in the LAU System", Proc. 1977 International Conf. on Parallel Processing, pp. 87-92, August 1977.

[T] Tesler, L. G., and H. J. Enea, "A Language Design for Concurrent Processes", AFIPS Conf. Proc., Vol. 32, pp. 403-408, SJCC 1968.

[U] Urschler, G., "The Transformation of Flow Diagrams into Maximally Parallel Form", Sagamore Comp. Conf. on Parallel Processing, pp. 38-46, March 1973.