MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Project MAC

COMPUTATION STRUCTURES GROUP MEMO No. 33

SUBSYSTEM SHARING IN PARALLEL, ASYNCHRONOUS PROCESSING

Richard Ribak

SUBSYSTEM STAGING IN PARALLEL, ASYNCHRONOUS PROCESSING

by

RICHARD RIBAK

S.B., Massachusetts Institute of Technology
(1967)

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1968

Signature of Author 
Department of Electrical Engineering, May 17, 1968

Certified by _____
Thesis Supervisor

Accepted by _____
Chairman, Departmental Committee on Graduate Students

SUBSYSTEM SHARING IN PARALLEL, ASYNCHRONOUS PROCESSING

by

RICHARD RIBAK

Submitted to the Department of Electrical Engineering
on May 6, 1968 in partial fulfillment of the
requirements for the Degree of Master of Science

ABSTRACT

A parallel asynchronous control structure is developed which is called a pipeline graph. It controls the simultaneous process of N independent jobs. The pipeline graph is developed to be used in the modular design of complex computer systems.

THESIS SUPERVISOR: Jack B. Dennis

TITLE: Associate Professor of Electrical Engineering

ACKNOWLEDGEMENT

The author is indebted to F.L. Luconi and D.R. Slutz for the assistance they gave me in my search for a thesis topic and to Professor J.B. Dennis for being my thesis supervisor. Finally special thanks to Marsha Baker for typing this thesis.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

What will the next generation of computers be like? Will there be more sophisticated serial devices or will there be an emphasis towards more parallelism? IBM[1] in describing its Systems 360/91 provides one answer. They state that order-of-magnitude improvements in computer performance due to system organization alone is unlikely. This assumes the constraint that the machine executes one instruction at a time. They then suggest that if substantial improvement is to be made through system organization the emphasis will have to be on parallel processing.

Parallel processing has been done on computers to a limited extent for sometime. The IBM 709 which was released in 1958 allowed input-output processing while the central processor was executing instructions. The CDC 6600[2] which was first delivered in 1964 has one central processor and ten peripheral processors. The peripheral processors can do a smaller amount of arithmetic, they contain the monitor system and do all the input-output. These peripheral processors operate in parallel with the central processor.

There has also been a trend towards operating two or more central processors in parallel such as will be possible in the Multics[3] system. Here each central processor executes a segment of a program but communications between the central processors must be minimized to

reduce supervisory overhead time. What has not been done except on certain special computers such as the Solomon[4] is to execute individual instructions in parallel.

There is a class of special purpose computers called pipeline[5] systems which can process two or more independent jobs simultaneously. The most important characteristic of these systems is flow rate. Flow rate is defined as the number of jobs which flow from the input to the output of the system per unit time. A pipeline system will tend to minimize the idle time of the processors.

A serious limitation in the above hardware systems is their ability to sequence control when simultaneously processing subsections of the same job or different jobs. This thesis will concentrate on the problems of control in parallel, asynchronous processing. It will develop a pipeline control system which controls the simultaneous processing of N independent jobs. It will investigate the problems involved in distributing jobs between identical processing units. Finally problems in the modular design of control systems will be investigated.

Some theoretical work has been done along these lines. J.E. Rodriquez[6] developed a control system called a program graph. The graph consists of three types of control nodes--operators, selectors, and junctions and two types of links--control and data. These elements along with a set of rules for describing the connection of nodes and links allow one to build a control system to control parallel, asynchronous processing. The program graph has limitations. Only one job can enter a loop at a time. Null

signals are generated which limit the amount of parallel processing that can take place and the program graph cannot distinguish between different jobs.

F. Luconi[7] has developed a schema which allows one to connect a control structure such as the program graph to a processing structure. He also developed a set of tests to determine if a system is output functional. Output functional means that for identical input values to a system the system will produce identical output values. This thesis will not consider the problem of connecting a control structure to a processing structure but will use the output functional tests developed in F. Luconi's thesis[7].

CHAPTER II

A PROCESSING SYSTEM

A digital hardware processing system can be viewed as consisting of two logically distinct, interacting parts called a processing structure and a control structure. The processing structure consists of processing units such as the central processors of computers, adders and multipliers. The processing units are interconnected by means of links. A link is a memory unit which can store the output of a processing unit or act as the input to a processing unit. The flow of data, the processing of data and the storage of data all takes place in the processing structure.

The control structure determines the sequence in which the processing units will process the data in their input links. The control structure consists of control operators which can sendcontrol to other control operators and can activate the processing units. The control operators are interconnected by means of links and each processing unit is connected to a control operator by means of a control link. The control link has two parts, an initiation line and a completion line. The control operator sends a signal through the initiation line to the processing unit. This causes the processing unit to process the values in its input links. After completing the processing, the unit sends a completion signal through the completion line to the control operator.

This thesis will concentrate on the control structure. It will not consider the processing structure or the interconnection of the control and processing structures by means of control links.

REQUIREMENTS FOR A CONTROL STRUCTURE

A control structure should be able to process more than one job simultaneously. Assume that there is a control system with control operators connected to N ($N \gg 1$) processing units. The control is such that only one processing unit is activated at a time. This system is not efficient because N-1 processing units would always be idle. The system can be made more efficient by processing more than one job simultaneously thereby increasing the number of processing units that will be simultaneously processing.

The control structure should be able to distinguish between jobs. Assume that there are two jobs which require processing. 99% of the processing required by the two jobs is identical. The jobs should be able to share the 99% of the processing system they have in common.

Current control systems treat a loop as though it were a feedback path which has one entrance and one exit and which can control the processing of one job at a time. These restrictions should be removed. Their removal would increase the number of processing unitswhich can simultaneously process jobs.

Null signals are signals which do not cause a control operator to activate the processing unit which is connected to it. Null signals are undesirable because they increase the idle time of the processing units.

CONTROL OPERATORS

It is the purpose of this section to define a set of control operators (C-operators). These operators will be used to form a control structure called a pipeline graph (definition 2.2) which will be capable of regulating the sequence of control of N independent jobs simultaneously. The pipeline graph will meet the requirements for a control structure as defined in the previous section.

Before defining the operators some terminology must be explained. An operator has connected to it a finite set of links L. These links are its input links, its output links and special control links. Each link ℓ_i has a value V associated with it ($v(\ell_i) = V$). Each link is restricted to a finite set of values. Each control operator C_i has a domain $D(C_i)$ and a range $R(C_i)$. $D(C_i)$ defines the set of values in the links of the set L which cause C_i to transform. When C_i transforms, the value of each link in the set L is changed to the value defined for that link in the set $R(C_i)$ ($C_i : D(C_i) \rightarrow R(C_i)$).

1. Function Operator: The function operator is illustrated in figure 2.1.

   It is defined as an ordered 4-tuple $F = < L, I, P, J >$ where:

   1) $L = \{\ell_1, \ell_2, \ldots, \ell_z\}$[1] is its finite set of links
   2) $I = \{i_1, i_2, \ldots, i_d\}$ is its finite set of input links
   3) $P = \{p_1, p_2, \ldots, p_r\}$ is its finite set of output links
   4) $J = \{j_1, j_2, \ldots, j_n\}$ is the finite set of values (job numbers) that can appear in the links of the set L.
   5) $L = I \cup P$

[1] See Appendix A for an explanation of the notation.

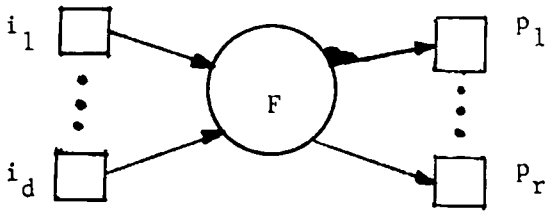


Figure 2.1

TRANSFORMATION

$i_1 \cdots i_d \quad p_1 \cdots p_r \rightarrow i_1 \cdots i_d \quad p_1 \cdots p_r$

$X \cdots X \quad 0 \cdots 0 \rightarrow 0 \cdots 0 \quad X \cdots X$

FUNCTION OPERATOR

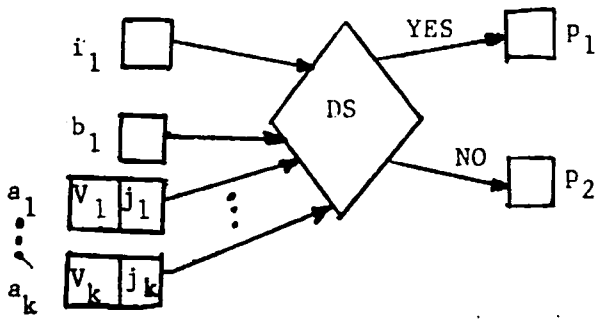


Figure 2.2

TRANSFORMATION

$i_1 \; p_1 \; p_2 \rightarrow i_1 \; p_1 \; p_2$

$X \quad 0 \quad 0 \rightarrow 0 \quad X \quad 0 \quad$ if $[v(b_1) R \; V_i/j_i = v(i_1)]$

$X \quad 0 \quad 0 \rightarrow 0 \quad 0 \quad X \quad$ if $[v(b_1) R \; V_i/j_i = v(i_1)]$

or $[v(i_1) \neq j_1 \; V \; j_2 \; V \; \ldots \; Vj_k]$

DATA SELECTOR

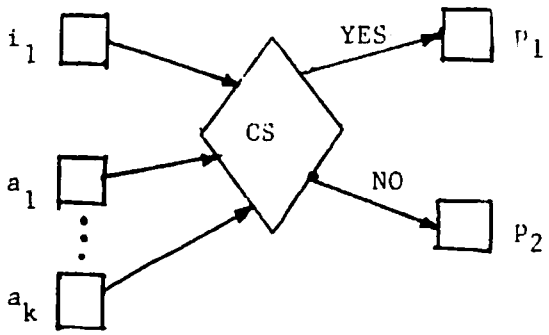


Figure 2.3

TRANSFORMATION

$i_1\ p_1\ p_2 \rightarrow i_1\ p_1\ p_2$

X 0 0 → 0 X 0 if $v(i_1) = v(a_1)\ V \cdots Vv(a_k)$

X 0 0 → 0 0 X if $v(i_1) \neq v(a_1)\ V \cdots Vv(a_k)$

CONTROL SELECTOR

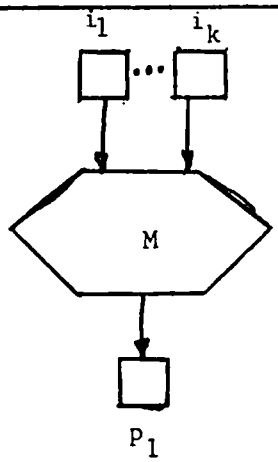


Figure 2.4

TRANSFORMATION

$i_g\ p_1 \rightarrow i_g\ p_1$

X 0 → 0 X for any $i_g (i_g \in I)$

Restriction: No two input links i_k, i_ℓ can ever have the identical value at the same instant.

MERGER

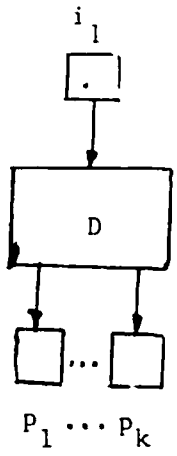


Figure 2.5

TRANSFORMATION

$i_1\ p_g \rightarrow i_1\ p_g$

X 0 → 0 X for any p_g ($p_g \in P$)

Restriction: No two output links p_k, p_ℓ can ever have the identical value at the same instant.

DISTRIBUTOR

The transformation associated with the function operator is of the form:

$$F:D(F) \rightarrow R(F)$$

$$D(F) = \{D_1(F), D_2(F), \cdots, D_n(F)\}$$

$$R(F) = \{R_1(F), R_2(F), \cdots, R_n(F)\}$$

There is one domain $D_i(F)$ and one range $R_i(F)$ associated with each j_i ($j_i \in J$).

$$D_i(F) = \{< v(I) \overset{2}{=} j_i, v(P) = 0> /(j_i \in J)\}$$

$$R_i(F) = \{< v(I) = 0, v(P) = j_i> /(j_i \in J)\}$$

The job number j_i in $D_i(F)$ is identical to the j_i in $R_i(F)$. Every $D_k(F)$ and $R_k(F)$ is of the above form.The only difference is that j_k is used instead of j_i. The actual transformations are as follows

$$F:D_1(F) \rightarrow R_1(F)$$

$$F:D_2(F) \rightarrow R_2(F)$$

$$\vdots$$

$$F:D_n(F) \rightarrow R_n(F)$$

If the value of every input link i_i ($i_i \in I$) is equal to j_i ($j_i \in J$) and the value of every output link p_i($p_i \in P$) is equal to zero then the operator transforms setting the value of every input link equal to zero and the value of every output link equal to j_i.

---

[2] $v(I) = j_i$ means that the value of every link in I is equal to j_i.

2. Data Selector: The data selector (figure 2.2) is used to make a conditional branch based on the value in b_1. It is defined as an ordered 6-tuple $DS = < L, I, P, A, B, J >$ where:

1) $L = \{\ell_1, \ell_2, \ldots, \ell_z\}$ is its finite set of links

2) $I = \{i_1\}$ is its single input link

3) $P = \{p_1, p_2\}$ is its set of two output links

4) $A = \{a_1, a_2, \ldots, a_k\}$ is its finite set of control links

5) $B = \{b_1\}$ is a single control link with a value V_k which is associated with the job number j_k in link i_1 $(i_1 \in I)$

6) $J = \{j_1, j_2, \ldots, j_n\}$ is the finite set of values (job numbers) that can appear in the link in the set I

7) $L = I \cup P \cup A \cup B$

8) $a_i = \{V_i, j_i\}$ is a double link $(a_i \in A)$ which contains a value V_i and a job number j_i

The function associated with each data selector is of the form:

$DS: D(DS) \rightarrow R(DS)$

$D(DS) = \{D_1^1(DS), D_1^2(DS), D_2^1(DS), D_2^2(DS), \cdots, D_n^1(DS), D_n^2(DS)\}$

$R(DS) = \{R_1^1(DS), R_1^2(DS), R_2^1(DS), R_2^2(DS), \cdots, R_n^1(DS), R_n^2(DS)\}$

There are two domains $D_i^1(DS)$ and $D_i^2(DS)$ and two ranges $R_i^1(DS)$ and $R_i^2(DS)$ associated with each j_i.

$$D_i^1(DS) = \{< v(I) = j_i, v(P) = 0> \mid (v(b_1)RV_i^3) \;\&\; (V_i \in a_i) \;\&\; (j_i \in a_i) \;\&\; (a_i \in A)\} \;\&\; (j_i \in J)\}$$

$$D_i^2(DS) = \begin{cases} \{< v(I) = j_i, v(P) = 0> \mid (v(b_1) \not R V_i^4) \;\&\; (j_i \in J) \;\&\; (V_i \in a_i) \;\&\; (j_i \in a_i) \;\&\; (a_i \in A)\} \\ \{v(I) \neq j_1 \vee j_2 \vee \cdots \vee j_n\} \end{cases}$$

---

[3] $v(b_1)RV_i$ means that the value of b_1 is in the range of V_i.

[4] $v(b_1) \not R V_i$ means that the value of b_1 is not in the range V_i.

$$R_i^1(DS) = \{< v(I) = 0,\ v(p_1) = j_i,\ v(p_2) = 0 > \mid (p_1 \in P) \ \&\ (p_2 \in P) \ \&\ (j_i \in J)\}$$

$$R_i^2(DS) = \{< v(I) = 0,\ v(p_1) = 0,\ v(p_2) = j_i > \mid (p_1 \in P) \ \&\ (p_2 \in P) \ \&\ (j_i \in J)\}$$

D_i^1, D_i^2, R_i^1, and R_i^2 refer to the identical j_i. Every D_k^1, D_k^2, R_k^1, and R_k^2 is of the above form. The only difference is that j_k is used instead of j_i. The actual transforms are as follows:

$$DS{:}D_1^1(DS) \rightarrow R_1^1(DS)$$
$$DS{:}D_1^2(DS) \rightarrow R_1^2(DS)$$
$$DS{:}D_2^1(DS) \rightarrow R_2^1(DS)$$
$$DS{:}D_2^2(DS) \rightarrow R_2^2(DS)$$
$$\vdots$$
$$DS{:}D_n^1(DS) \rightarrow R_n^1(DS)$$
$$DS{:}D_n^2(DS) \rightarrow R_n^2(DS)$$

If the value in b_1 is in the range of V_k then control is sent to link p_1 otherwise control is sent to link p_2.

3. Control Selector: The control selector (figure 2.3) is used to make a conditional branch based on the job number in link i_1. It is defined as an ordered 5-tuple CS = < L, I, P, A, J > where:

   1) $L = \{\ell_1, \ell_2, \ldots, \ell_z\}$ is its finite set of links

   2) $I = \{i_1\}$ is its single input link

   3) $P = \{p_1, p_2\}$ is its set of two output links

   4) $A = \{a_1, a_2, \ldots, a_k\}$ is its finite set of control links

5) $J = \{j_1, j_2, \ldots, j_n\}$ is the finite set of values (job numbers) that can appear in the link in the set I

6) $L = I \cup P \cup A$

The function associated with each control selector is of the form:

$CS: D(CS) \rightarrow R(CS)$

$D(CS) = \{D_1^1(CS), D_1^2(CS), D_2^1(CS), D_2^2(CS), \ldots, D_n^1(CS), D_n^2(CS)\}$

$R(CS) = \{R_1^1(CS), R_1^2(CS), R_2^1(CS), R_2^2(CS), \ldots, R_n^1(CS), R_n^2(CS)\}$

There are two domains $D_i^1(CS)$, $D_i^2(CS)$ and two ranges $R_i^1(CS)$, $R_i^2(CS)$ associated with each j_i.

$$D_i^1(CS) = \{< v(I) = j_i, v(P) = 0 > \mid (\exists a_i(v(I) = v(a_i)) \& (a_i \in A)) \& (j_i \in J)\}$$

$$D_i^2(CS) = \{< v(I) = j_i, v(P) = 0 > \mid (\forall a_i)((v(a_i) \neq v(I)) \& (a_i \in A)) \& (j_i \in J)\}$$

$$R_i^1(CS) = \{< v(I) = 0, v(p_1) = j_i, v(p_2) = 0 > \mid (j_i \in J) \& (p_1 \in P) \& (p_2 \in P)\}$$

$$R_i^2(CS) = \{< v(I) = 0, v(p_1) = 0, v(p_2) = j_i > \mid (j_i \in J) \& (p_1 \in P) \& (p_2 \in P)\}$$

D_i^1, D_i^2, R_i^1, and R_i^2 refer to the identical j_i. Every D_k^1, D_k^2, R_k^1 and R_k^2 is of the above form. The only difference is the j_k is used instead of j_i. The actual transformations are as follows:

$CS: D_1^1(CS) \rightarrow R_1^1(CS)$

$CS: D_1^2(CS) \rightarrow R_1^2(CS)$

$CS: D_2^1(CS) \rightarrow R_2^1(CS)$

$CS: D_2^2(CS) \rightarrow R_2^2(CS)$

$\vdots$

$CS: D_n^1(CS) \rightarrow R_n^1(CS)$

$CS: D_n^2(CS) \rightarrow R_n^2(CS)$

If the value in i_1 is equal to the value of any link in the set A then control is sent to link p_1 otherwise control is sent to link p_2.

4. Merger: The merger (figure 2.4) is used for controlling the flow of jobs into a loop and for merging the output links of k sections of a graph into one link (path). It is defined as an ordered 4-tuple $M = < L, I, P, J >$ where:

1) $L = \{\ell_1, \ell_2, \ldots, \ell_z\}$ is its finite set of links

2) $I = \{i_1, i_2, \ldots, i_k\}$ is its finite set of input links

3) $P = \{p_1\}$ is its single output link

4) $J = \{j_1, j_2, \ldots, j_n\}$ is its finite set of values (job numbers) that can appear in the links in the set I.

5) $L = I \cup P$

The function associated with each merger is of the form:

$M:D(M) \rightarrow R(M)$

$D(M) = \{D_1(M), D_2(M), \ldots, D_n(M)\}$

$R(M) = \{R_1(M), R_2(M), \ldots, R_n(M)\}$

There is one domain $D_i(M)$ and one range $R_i(M)$ associated with each j_i.

$D_i(M) = \{< v(i_k) = j_i, v(P) = 0> \mid \exists i_k((v(i_k) = j_i) \& (i_k \in I) \& (j_i \in J))\}$

$R_i(M) = \{< v(i_k) = 0, v(P) = j_i > \mid (i_k \in I) \& (j_i \in J)\}$

D_i and R_i refer to the identical j_i and ℓ_k. Every D_k and R_k is of the above form. The only difference is the j_k is used instead of j_i. The actual transformations are as follows:

$M:D_1(M) \rightarrow R_1(M)$

$M:D_2(M) \rightarrow R_2(M)$

$\vdots$

$M:D_n(M) \rightarrow R_n(M)$

The merger arbitrarily picks one of its input links i_k and transforms it to its output link p_1. There is one restriction on the use of the merger. It must be connected in a graph such that no two of its input links i_k, i_ℓ can every have the identical value (job number) at the same time.

5. Distributor: The distributor (figure 2.5) is the inverse of a merger. It is defined as an ordered 4-tuple $D = \langle L, I, P, J \rangle$ where:

1) $L = \{\ell_1, \ell_2, \ldots, \ell_z\}$ is its finite set of links
2) $I = \{i_1\}$ is its single input link
3) $P = \{p_1, p_2, \ldots, p_k\}$ is its finite set of output links
4) $J = \{j_1, j_2, \ldots, j_n\}$ is its finite set of values (job numbers) that can appear in the link in the set I
5) $L = I \cup P$

The function associated with each distributor is of the form:

$D: D(D) \rightarrow R(D)$

$D(D) = \{D_1(D), D_2(D), \ldots, D_n(D)\}$

$R(D) = \{R_1(D), R_2(D), \ldots, R_n(D)\}$

There is one domain $D_i(D)$ and one range $R_i(D)$ associated with each j_i.

$$D_i(D) = \{\langle v(I) = j_i, v(p_k) = 0 \rangle \mid \exists p_k ((v(p_k) = 0) \,\&\, (p_k \in P) \,\&\, (j_i \in J))\}$$

$$R_i(D) = \{\langle v(I) = 0, v(p_k) = j_i \rangle \mid (p_k \in P) \,\&\, (j_i \in J)\}$$

D_i and R_i refer to the identical j_i and p_k. Every D_k and R_k is of the above form. The only difference is the j_k is used instead of j_i. The actual transformations are as follows:

$D:D_1(D) \rightarrow R_1(D)$

$D:D_2(D) \rightarrow R_2(D)$

$\vdots$

$D:D_n(D) \rightarrow R_n(D)$

The distributor transforms the value in its input link i_1 to anyone of its empty output links p_k. There is one restriction on the use of a distributor. It must be connected to a graph such that no two of its output links p_k, p_ℓ have the identical value at the same time.

DISTRIBUTOR-MERGER STRUCTURE

If the output links of a distributor are connected to different graph structures then the transformation is nondeterministic. That is, since the distributor arbitrarily picks one of its empty output links it does not know ahead of time how the job on its input link will be processed.

Definition 2.1; The distributor-merger structure is illustrated in figure 2.6. It is defined as an ordered 6-tuple DM = < L, I, A, P, B, G > where:

1) $L = \{\ell_1, \ell_2, \ldots, \ell_z\}$ is its finite set of links
2) $I = \{i_1\}$ is its single input link which is the input link of the distributor D.
3) $A = \{a_1, a_2, \ldots a_k\}$ is the finite set of output links of the distributor D whose input link is in the set I.
4) $P = \{p_1\}$ is its single output link which is the output link of the merger M
5) $B = \{b_1, b_2, \ldots b_k\}$ is the finite set of input links of the merger M whose output link is in the set P
6) $G = \{g_1, g_2, \ldots g_k\}$ is a finite set of equivalent graphs
7) $L \supseteq I \cup A \cup P \cup B$

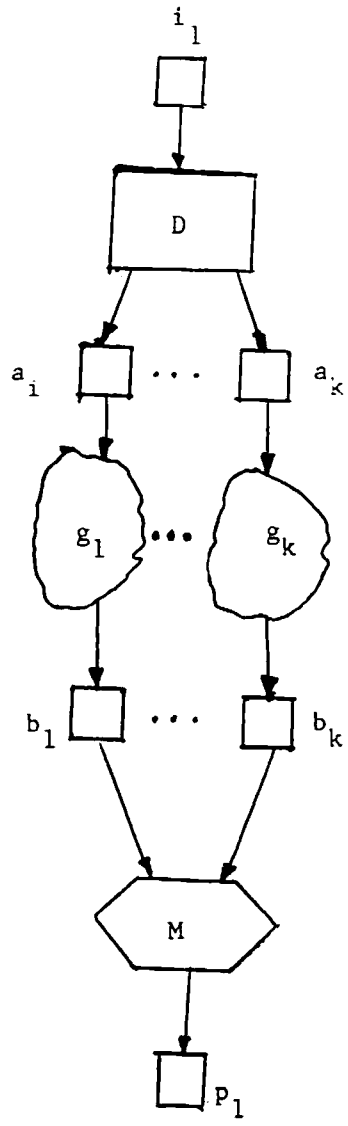


Figure 2.6

DISTRIBUTOR-MERGER STRUCTURE

A, B, and G contain the identical number of elements k. Each g_i has one input link $\ell_i (\ell_i \in A)$ and one output link $\ell_j (\ell_j \in B)$.

A distributor-merger structure does not have the above problem. The transformation from its input link i_1 to its output link t_1 is the same for each job.

PIPELINE GRAPH

A pipeline graph consists of a finite set of C-operators interconnected with links. To start the processing, a value is placed on a subset of the graphs links called the input interface links. When that value appears in a second subset of the graphs links called the output interface links the processing is completed. The value placed in the links is called the job number and is associated with one particular job.

Definition 2.2: A pipeline graph is an ordered 5-tuple PG = < L, C, IT, OT, J > where:

1) $L = \{\ell_1, \ell_2, \ldots, \ell_z\}$ is its finite set of links
2) $C = \{c_1, c_2, \ldots, c_k\}$ is its finite set of operators
3) $IT = \{i_1, i_2, .., i_r\}$ is its finite set of input interface links
4) $OT = \{t_1, t_2, \ldots, t_e\}$ is its finite set of output interface links
5) $J = \{j_1, j_2, \ldots, j_n\}$ is the finite set of values (job numbers) that can appear in the links of the set IT.
6) $L \supseteq IT \cup OT$
7) Every c_i is either a function operator or a data selector or a control selector or a merger or a distributor in a distributor-merger structure.
8) Every link $\ell_i (\ell_i \in L)$ can be included in the input-output sets of at most two c-operators. A link connecting two c-operators must be an input link of one c-operator and an output link of the other.

9) The construction of the graph must be such that it does not violate the restrictions on the c-operators.

10) Every job has a unique set of input interface links $IT(J_i)$ $(IT(J_i) \subseteq IT)$ and a unique set of output interface links $OT(J_i)$ $(OT(J_i) \subseteq OT)$.

11) Every job has a unique job number j_i $(j_i \in J)$.

Definition 2.3: A job is started on a pipeline graph by putting the same value $j_i (j_i \in J)$ into each input interface link $\ell_k (\ell_k \in IT(J_i))$ of that job. At that time the value j_i appears in no other link in the graph. When the graph finishes processing that job, that value appears in each output interface link $\ell_a (\ell_a \in OT(J_i))$ for that job and does not appear in any other link in the graph.

FUNCTION OPERATOR HANGUPS

In order to transform, a function operator must have the same value in each of its input links. If it has different values on its input links there is no way that the values can be changed or removed. The function operator is said to be hung up.

Referring to figure 2.7, M1 and M2 are mergers, F1, F2, and F3 are function operators, and DS is a data selector. ℓ_1 and ℓ_3 are job number "1"'s input interface links and ℓ_{10} its output interface link. ℓ_2 and ℓ_4 are job number "2" 's input interface links and ℓ_{11} its output interface link. The values shown in the links are the initial values. Let M1 transform the value in ℓ_1 to ℓ_5 and M2 transform ℓ_4 to ℓ_6. Next F1 transforms the value in ℓ_5 to ℓ_7 and F2 transforms the value in ℓ_6 to ℓ_8. The input links ℓ_7 and ℓ_8 of F3 now have the values 1 and 2 respectively. F3 is hung up.

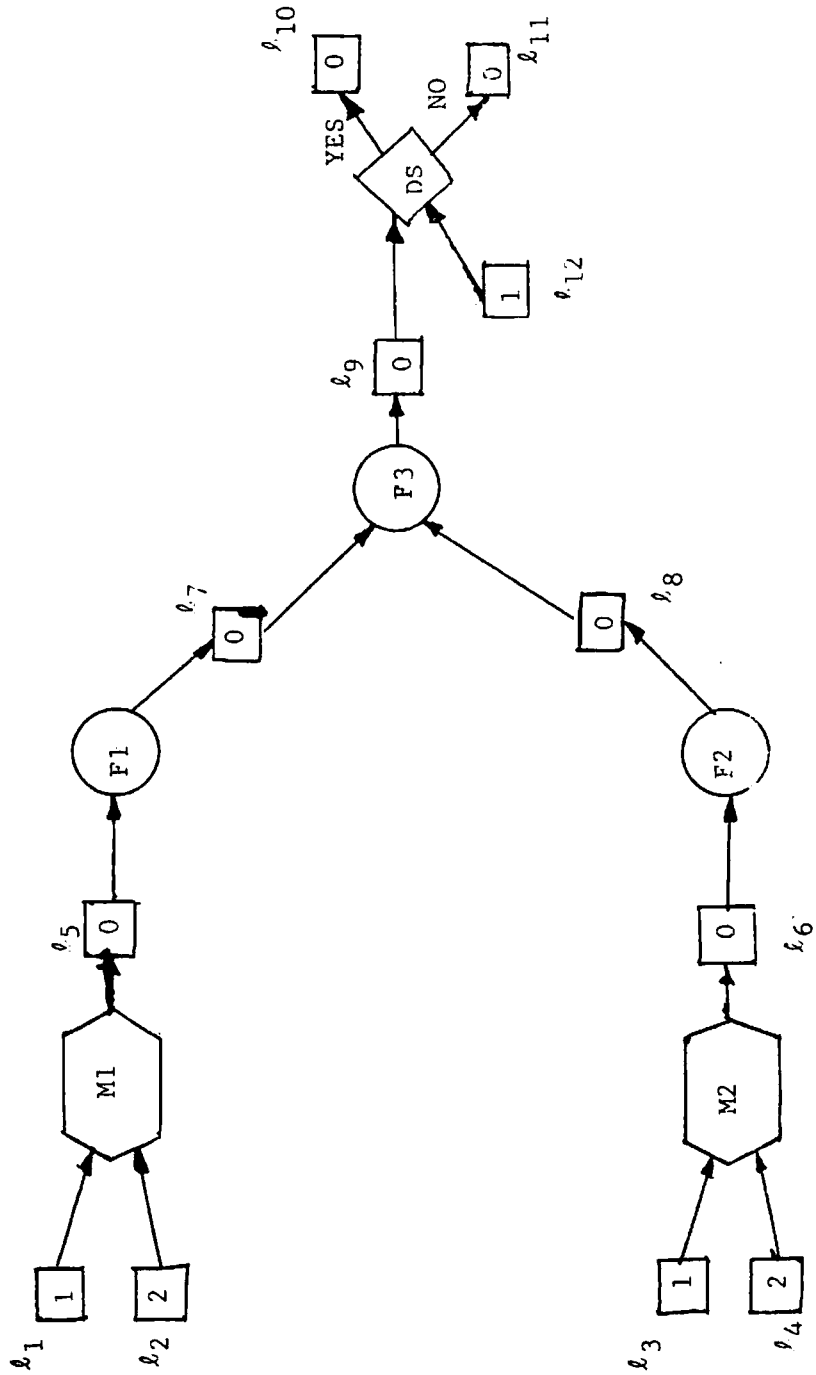

ℓ_1
1
ℓ_2
2
M1
ℓ_5
0
F1
ℓ_7
0
ℓ_3
1
ℓ_4
2
M2
ℓ_6
0
F2
ℓ_8
0
F3
ℓ_9
0
DS
YES
ℓ_{10}
0
NO
ℓ_{11}
0
ℓ_{12}
1

Figure 2.7

Definition 2.4: A function operator in standard form has a set of internal links that store every value that appears on each of its input links. When the identical number has appeared on each of its input links and has been stored in its internal links that number will be transformed to the set of output links of the function operator.

Figure 2.8A illustrates one possible internal structure of a function operator in standard form. It can have at most two values (jobs) appearing on each of its two input links ℓ_1 and ℓ_2. CS1 and CS2 are control selectors, F1, ..., F7 are function operators and M1 is a merger. Figure 2.8B shows the symbol for a function operator in standard form. ℓ_1, ℓ_2, ℓ_3 and ℓ_4 are the corresponding links in figure 2.8A and 2.8B. Referring to figure 2.8A, F5 detects if J1 has entered both links ℓ_1 and ℓ_2. F6 detects if the other job number has entered both ℓ_1 and ℓ_2. If both F5 and F6 transform at the same time then M1 will select the output from one of them and transform it to the input link of F7. F7 is needed to distribute the job number to more than one output link. CS1 and CS2 channel the job numbers to F5 or F6. F1, ..., F4 are needed because CS1 and CS2 cannot transform unless their output links are zero.

A function operator in standard form is used whenever the function operator has more than one input link. The maximum number of internal storage links it needs is equal to the maximum number of job numbers that can appear on anyone of its input links times the total number of input links. The maximum number of job numbers that can appear on an input link is equal to the number of sets of input interface links.

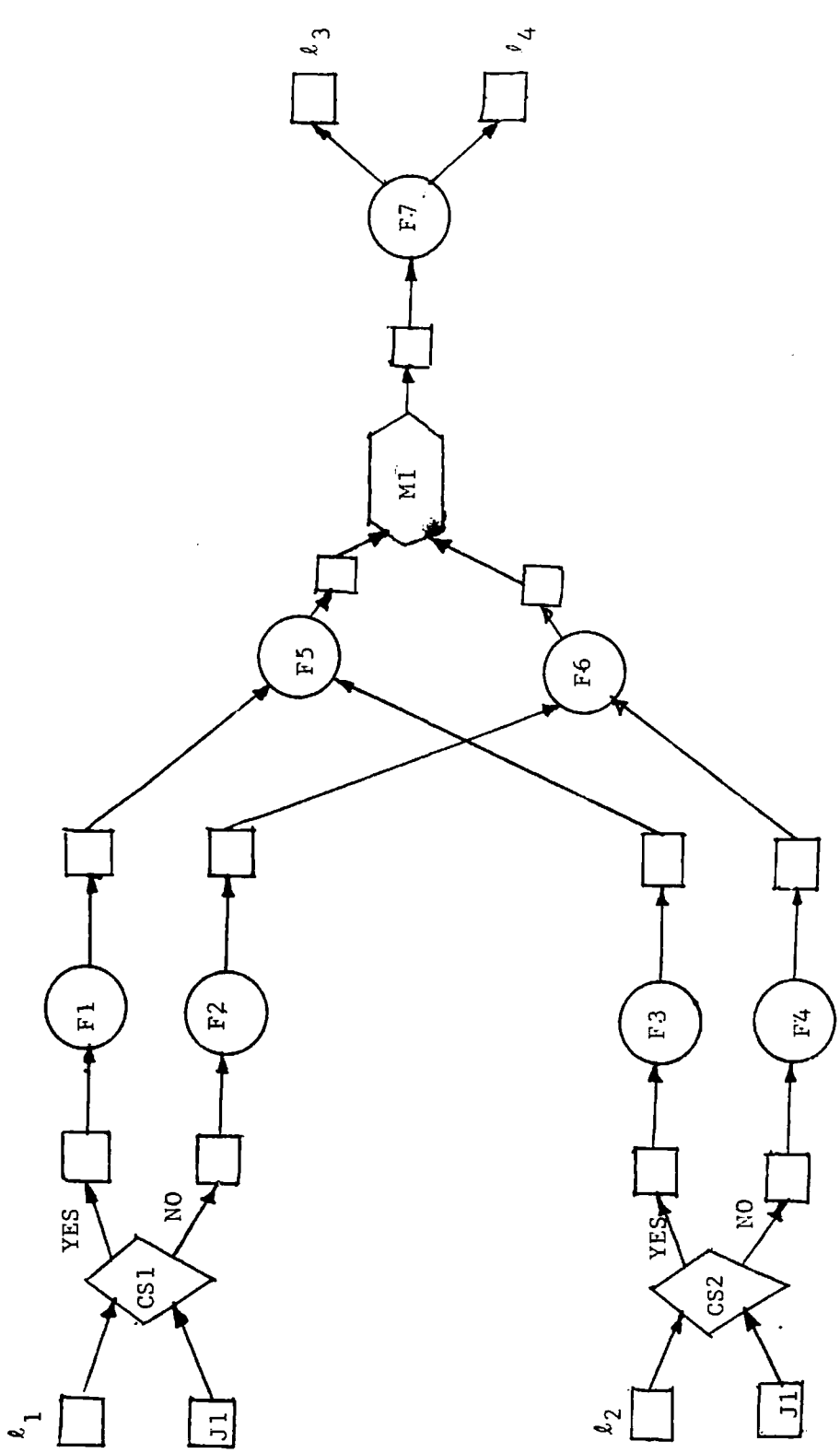


Figure 2.8A

INTERNAL STRUCTURE OF FUNCTION OPERATOR IN STANDARD FORM

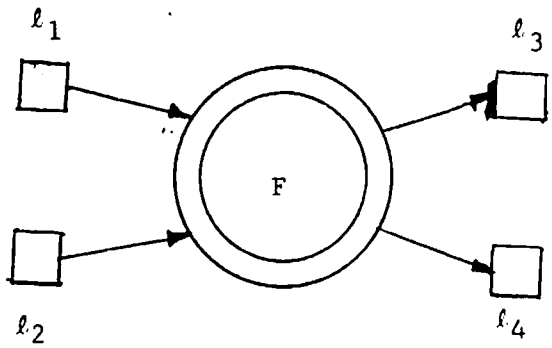


Figure 2.8B

SYMBOL FOR FUNCTION OPERATOR IN STANDARD FORM

EXAMPLES OF A PIPELINE GRAPH

Figure 2.9 illustrates a pipeline graph which can process two jobs (J1, J2) simultaneously and produce the identical transformations on both. CS1 is a control selector, DS1 and DS2 are data selectors, F1, ... F9 are function operators, and M1, ..., M3 are mergers. ℓ_1 and ℓ_3 are the input interface links of job J1 and ℓ_5 is its output interface link. ℓ_2, ℓ_4 and ℓ_6 are the input-output interface links for job J2. The links with letters in them represent the initial condition of the graph. All links with no letters in them contain zeros (empty). The graph contains a loop starting at M1 and the loop has a decisional branch at DS2. It does not matter which job leaves the loop at DS1 first because F9 is in standard form. CS1 places each job in its proper output interface link.

Figure 2.10 illustrates a pipeline graph which can process two jobs (J1, J2) simultaneously but which does not produce identical transformations on each job. CS1 and CS2 are control selectors, M1 and M2 are mergers, and F1, ..., F8 are function operators. ℓ_1, ℓ_3 and ℓ_5 are the input-output interface links of job J1 and ℓ_2, ℓ_4 and ℓ_6 are the input-output interface links of job J2. Control selector CS1 sends job J1 to F6 and job J2 to F7. The rest of the graph is shared by both jobs. Note that F4 and F8 must be in standard form.

OUTPUT FUNCTIONAL PIPELINE GRAPH

A pipeline graph is output functional if for the same set of initial conditions it always produces the same values in its output interface links. A pipeline graph is completely functional if for the same set of initial

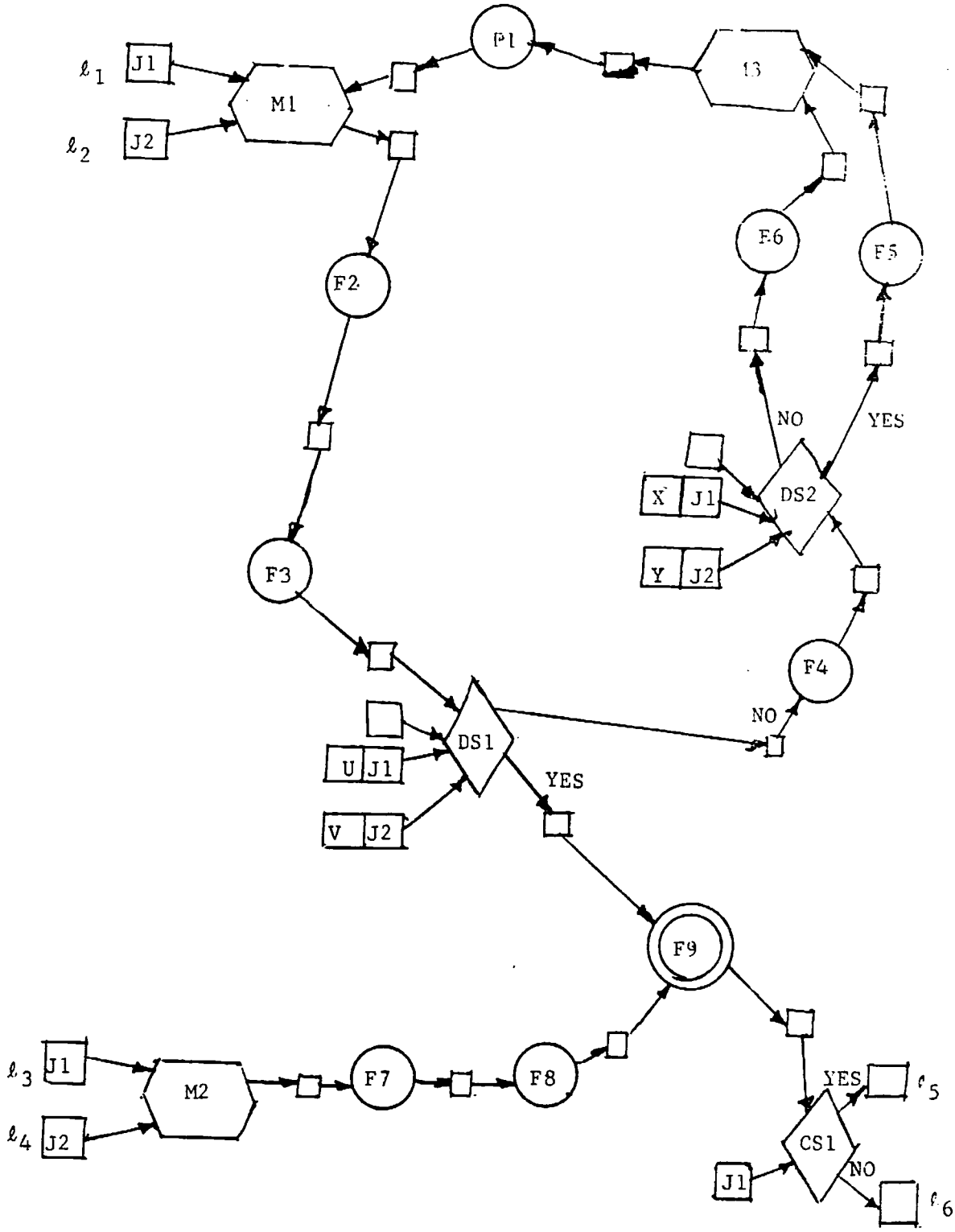


Figure 2.9

IDENTICAL PROCESSING OF TWO JOBS

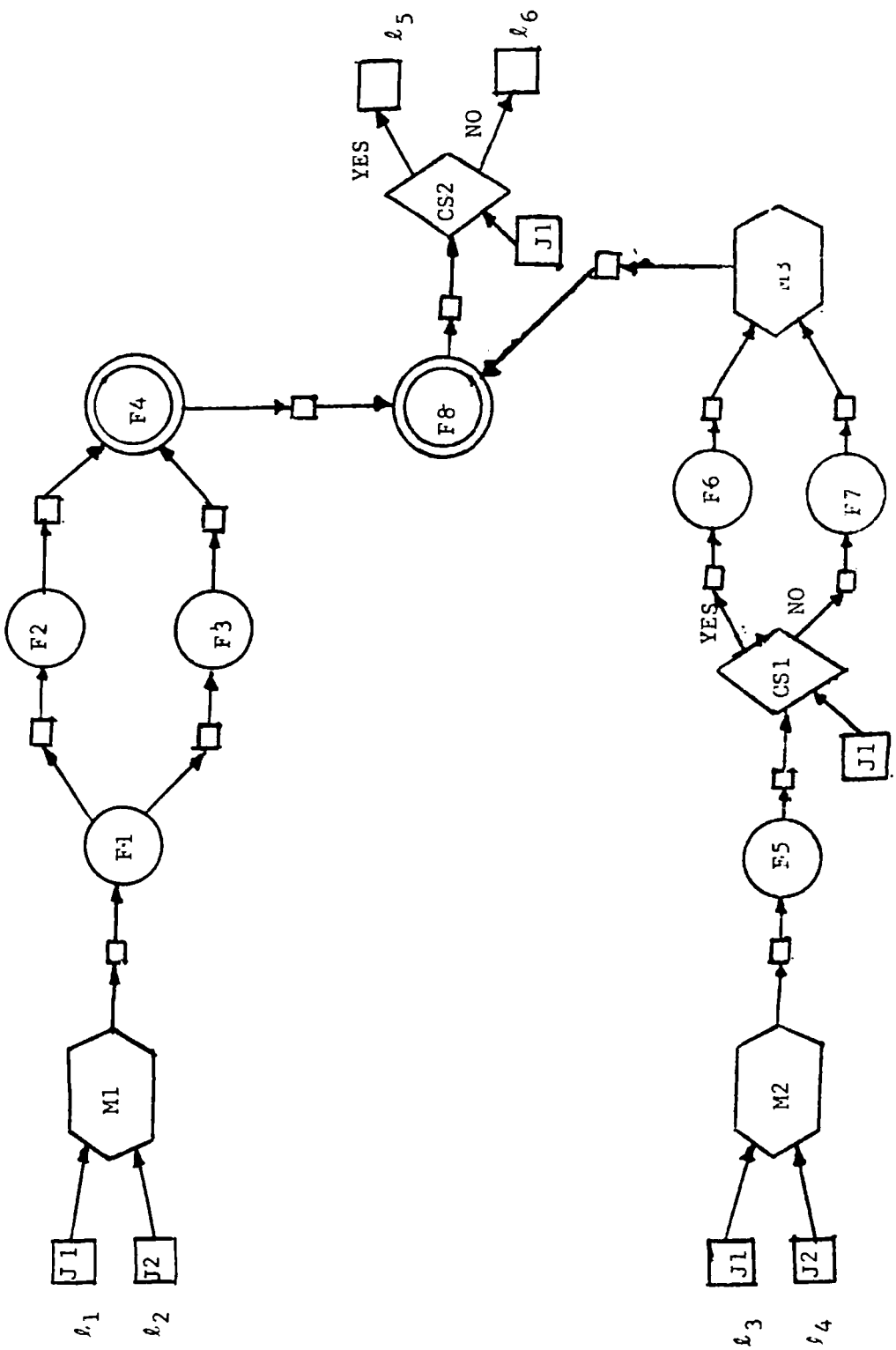


Figure 2.7

SHARING OF A SUBSECTION OF A PIPELINE GRAPH

conditions the same values in the same order appear on every link. Fred Luconi in his Ph.D. thesis[7] has developed a set of tests to determine if a set of c-operators will always form a completely functional graph. These tests (condition 2.1 and 2.2 and theorem 2.1) are taken from his thesis and repeated here. The theorem is repeated without proof.

Condition 2.1: A control structure is conflict-free if for any pair of operators a and b that can become applicable simultaneously either i) a and b have no output links in common, or ii) a or b do not alter the value of a common output link, or iii) both a and b indicate identical value changes to all common output links.

Condition 2.2: A control structure is transformation lossless if for any pair of operators a and b that can become applicable simultaneously either i) a and b do not communicate, or ii) the application of neither can affect the value changes about to be made in the other.

Theorem 2.1: All control structures which meet conditions 2.1 and 2.2 are completely functional.

Theorem 2.1 will be used to prove that a pipeline graph is output functional. The graph cannot be completely functional because for example the merger arbitrarily picks one of its input links to transform to its output link. Therefore the sequence of values in its output link is not repeatable.

Theorem 2.2: A pipeline graph is output functional.

It will be proven that the jobs are non-interacting (do not interfere with each other), that conditions 2.1 and 2.2 apply with respect to each job and that this establishes that the graph is output functional.

Proof:

1. No interaction takes place at the input links of a c-operator. The data selector, control selector, and distributor have only one input link. Therefore no interaction between jobs can take place at the inputs of these c-operators. The same applies to a merger because it transforms the value in only one input link at a time. The function operator will not transform unless all of its input links have the same value (job number). The function operator can therefore hangup (see section on hangups) but will not generate a wrong answer.

2. No interaction takes place at the output links of a c-operator. The function operator, data selector, control selector, and merger will only transform if all of their output links are zero. The distributor is part of the distributor-merger structure. All of the output links of the distributor are connected to equivalent graphs. The distributor transforms the value on its input link to one of its empty output links. Therefore there is no job interaction.

3. 1 and 2 prove that there is no job interaction at a c-operator and definition 2.2 establishes that there is no interaction in the links. Therefore there is no interaction between jobs in a pipeline graph.

4. Conditions 2.1 and 2.2 apply with respect to each job. Consider the function operator, data selector, control selector, and distributor-merger structure but not the merger. Every link appears in the input-output sets of at most two c-operators. The transformations are related

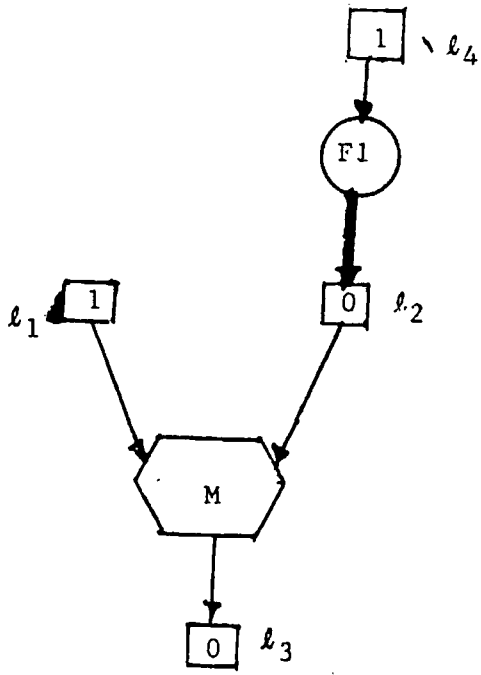


Figure 2.11A

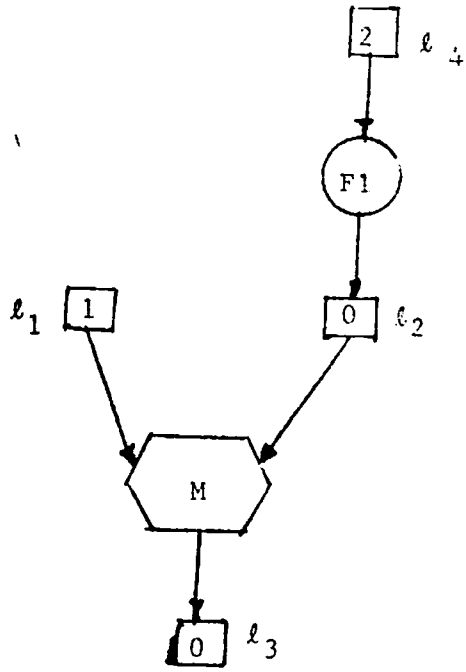


Figure 2.11B

so as to guarantee that at most one of the c-operators will be transformed at any one time. If neighboring operators cannot simultaneously transform, the connection is both conflict-free and transformation lossless.

The merger is conflict-free and transformation lossless with respect to each job. The condition in figure 2.11A cannot occur because this is a violation of the restriction that mergers can never have the same value (job) on any two of its input links. In figure 2.11B if M transforms first ℓ_3 will equal 1. If F1 transforms first ℓ_3 could equal 2. This would not cause any problems because each job is independent and non-interacting. Since the merger and any other c-operator connected to it can not simultaneously transform if they have the same job in their links it is conflict-free and transformation lossless with respect to each job.

5. It has been proven that a pipeline graph is conflict-free and transformation lossless with respect to each job and that the jobs are independent and non-interacting. This means that each job will cause the same sequence of c-operators to transform in the same sequence as if it were the only job in the graph. The pipeline graph is therefore output functional.

HANGUPS IN A LOOP FREE GRAPH

It has already been proven that the pipeline graph is output functional. This means that for a graph with N jobs in it the answers which are generated in the output interface links are repeatable. The question now asked is how long will one have to wait until these answers are generated. If the time it takes to generate an answer is infinity then some set of c-operators

in the graph is hung up. Note that a c-operator takes a finite amount of time to perform a transformation from its input links to its output links and that there are a finite number of c-operators and links in the graph.

In Figure 2.12 ℓ_1 and ℓ_2 are the input interface links and ℓ_6 and ℓ_7 are the output interface links. Job number "1" is placed in ℓ_1 and job number "2" is placed in ℓ_2. All other links are set to zero. The merger M selects ℓ_1 first and transforms the value in ℓ_1 to ℓ_3. Now function operator F transforms the value in ℓ_3 to ℓ_4 and ℓ_5. Next M transforms that value in ℓ_2 to ℓ_3. F will transform the value in ℓ_3 to ℓ_4 and ℓ_5 as soon as both ℓ_4 and ℓ_5 are set to zero. But ℓ_4 is not connected to any c-operator Therefore ℓ_4 can never be set to zero and F can never perform the transformation.F is said to be hung up.

STATE AND TRACE

The objective of this section of the thesis is to prove that under certain constraints a pipeline graph will not hangup. Before this can be done some terminology must be defined.

Definition 2.5: The state of a graph (STATE(i)) is defined as a set whose elements are the values in every link in the graph at a particular instant "i". $L = \{\ell_1, \ell_2, \ldots, \ell_n\}$ is the finite ordered set of all the links of the graph. $STATE(i) = \{v(\ell_1), v(\ell_2), \ldots, v(\ell_n)\}$ is the finite ordered set of the values of all the links in the set L.

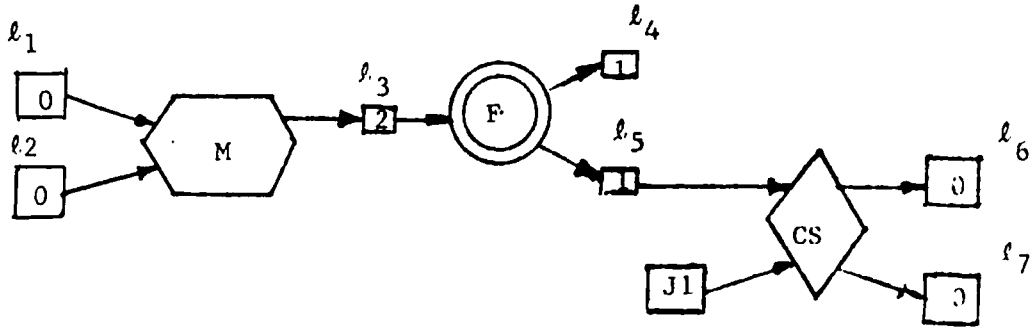


Figure 2.12

HANGUP

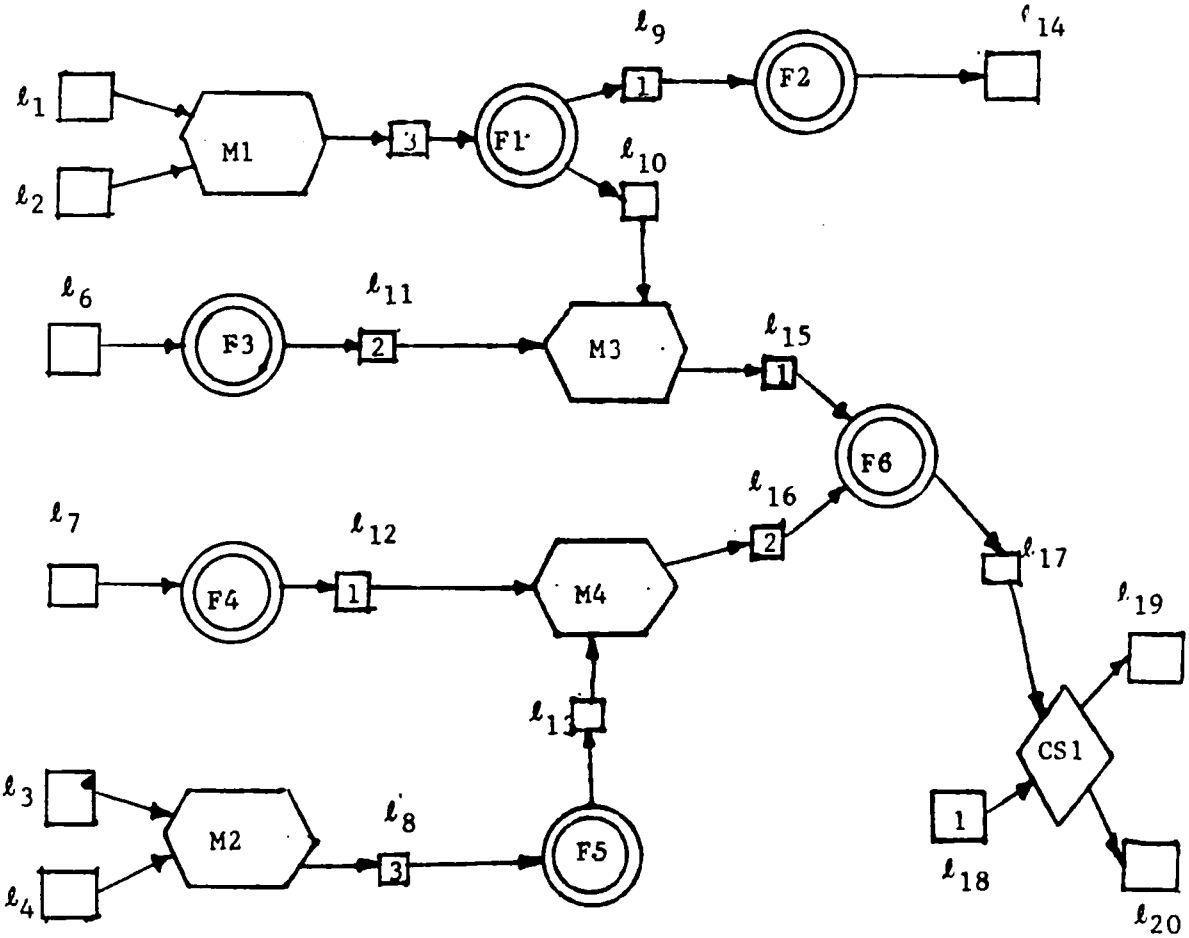


Figure 2.13

GRAPH IN STATE (J)

Definition 2.6: The set of states of a graph (SET.STATES) is defined as the set of all possible STATES(i) of a graph that can be reached from any allowable initial state STATE(j) and includes the initial states. Each state STATE(i) in SET.STATES is unique, that is no two states have elements all of whose values are identical.

SET.STATES = {STATE(1), STATE(2), ..., STATE(Z)}

SET.STATES is a finite set because a pipeline graph is defined as having a finite number of links and will accept only a finite number of jobs.

Definition 2.7: A trace (TRACE(ℓ_i, j) is an ordered set of c-operators which if each were transformed in sequence would cause a link ℓ_i to be set to zero (empty). It is defined for a graph in STATE(j).

TRACE(ℓ_i, j) = {INITIAL.SET (ℓ_i, j), C(J+1), ..., C(K)} is an ordered set of c-operators

INITIAL.SET (ℓ_i, j) = {C(1), ..., C(J)} is a set of c-operators

Each C(j) in INITIAL.SET (ℓ_i, j) is a c-operator which can transform without waiting for any other C(i) in the set TRACE (ℓ_i, j) or INITIAL.SET (ℓ_i, j) to transform.

Working backwards in the definition of TRACE (ℓ_i, j), C(K) could transform and set link ℓ_i to zero if C(K-1) could transform. C(K-1) could transform if ... C(J+1) could transform. C(J+1) could transform if the set of c-operators in INITIAL.SET (ℓ_i, j) could transform. The order of c-operators in TRACE (ℓ_i, j) does not have to be unique. It is possible that two c-operators could transform simultaneously. TRACE (ℓ_i, j) describes onepossible serial sequence of c-operator transformation. INITIAL.SET (ℓ_i, j) will contain c-operators whose output links are empty or c-operators who store the values which are in their input links into their internal links (function operator in standard form).

Referring to Figure 2.13 F1, ..., F6 are function operators in standard

form. M1, M2, M3, and M4 are mergers. CS1 is a control selector. Links ℓ_{14}, ℓ_{19}, and ℓ_{20} are the output interface links. Links ℓ_1, ℓ_2, ℓ_3, ℓ_4, ℓ_6, and ℓ_7 are the input interface links. Links ℓ_9, ℓ_{12}, ℓ_{15} and ℓ_{18} have the value "1". Links ℓ_{11} and ℓ_{16} have the value "2". Links ℓ_5 and ℓ_8 have the value "3". All other links have the value "0".

EXAMPLE 2.1

TRACE (ℓ_5, J) = {INITIAL.SET (ℓ_5, J), F1}

INITIAL.SET (ℓ_5, J) = {F2}

EXAMPLE 2.2

TRACE (ℓ_{11}, J) = {INITIAL.SET (ℓ_{11}, J), M3}

INITIAL.SET (ℓ_{11}, J) = {F6}

In example 2.2, F6 cannot transform because it has different values on each of its input links ℓ_{15} and ℓ_{16}. It is in standard form and therefore the values in ℓ_{15} can be transformed to F6's internal links.

A TRACE (ℓ_i, J) is RUN if the c-operators are made to transform and ℓ_i is therefore set to zero.

RESTRICTIONS FOR PREVENTING HANGUPS

Definition 2.8: A pipeline graph has a unique set of input interface links for each job. When a job is started the same value (job number) is put into each input interface link of that job. That value appears in no other link in the graph. When the graph finishes processing that job, that value appears

in each of the output interface links for the job and does not appear in any other link in the graph (including internal links of function operators in standard form). Every function operator is in standard form.

Experiment 2.1: Each of N jobs (each with its own unique job number, input and output interface links) is processed separately on the graph. This means that all links in the graph are set to zero and then job 1 is started. When job 1 is finished all links are set to zero and job 2 is started, ..., to job N. None of the N jobs hangup and each produces the correct final result according to definition 2.4.

THEOREM ON HANGUPS FOR LOOP-FREE GRAPHS

Theorem 2.3: If the N jobs defined in experiemtn 2.1 are run simultaneously in a loop-free pipeline graph the jobs will not hangup.

Theorem 2.3 is a theorem on processing complexity. A graph constructed of c-operators is not guaranteed to meet the requirements of definition 2.4. Clearly a graph can be constructed which does not work. The issue here is whether a graph which can successfully process N jobs individually, can process N jobs simultaneously. It has already been proven that a graph which meets the requirements of definition 2.8 is output functional. Therefore when values appear on the output interface links it is guaranteed that they are the correct ones. Theorem 2.3 establishes that the values will appear on the output interface in a finite amount of time.

PROOF: The proof will first show that there is an effective algorithm for constructing a TRACE (ℓ_i, j) for every link ℓ_i in every STATE (j) in the set SET.STATES.

1) L = $\{\ell_1, \ell_2, \ldots, \ell_z\}$ is the finite ordered set of links in a graph.

2) $(\forall\, \ell_i)\ (\forall_j)\ [(\ell_i \in L)\ \&\ (STATE(j) \in SET.STATES) \rightarrow TRACE\ (\ell_i, j)]$

TRACE (ℓ_i, j) will then be used to prove that a graph will not hangup.

The graph starts at an initial legal STATE (i). It then goes through a series of legal transitions until it reaches STATE(J) where STATE(J) $\in$ SET.STATES because SET.STATES is defined as containing all possible states that can be reached from a legal initial state. Consider an arbitrary link ℓ_i in state STATE (J).

TRACE (ℓ_x, J) ALGORITHM

$\ell_x = \ell_i$

1. If link ℓ_i is zero go to Done otherwise go to 2.
2. If link ℓ_i is not the input link of a c-operator go to Done otherwise go to 3.
3. If ℓ_i is not the input link of a function operator F(j) in standard form go to 4 otherwise:

   INITIAL.SET (ℓ_x, J) = $\{C(1), \ldots, C(N), F(j)\}$*

   TRACE (ℓ_x, J) = $\{$INITIAL.SET (ℓ_x, J) C(N+1), ..., C(Z)$\}$

   go to Done
4. If ℓ_i is not the input link of a merger M(j) go to 5 otherwise:

   The merger has only one output link ℓ_j.

*C(1), ..., C(N) were formed by N previous iterations of this algorithm.

If ℓ_j is zero then

[INITIAL.SET (ℓ_x, J) = {C(1), ..., C(N), M(j)}

TRACE (ℓ_x, J) = {INITIAL.SET (ℓ_x, J), C(N+1), ..., C(Z)}

go to Done] otherwise

[If ℓ_j is not zero then

TRACE (ℓ_x, J) = {INITIAL.SET (ℓ_x, J), M(j), C(N+1), ... C(Z).

Relabel ℓ_j as ℓ_i.

go to 2.]

5. If ℓ_i is not the input link of a data or control selector S(i) go to 6. otherwise:

S(i) has two output links ℓ_j and ℓ_k.

If ℓ_j and ℓ_k are both zero then

[INITIAL.SET (ℓ_x, J) = {C(1), ..., C(N), S(i)}

TRACE (ℓ_x, J) = {INITIAL.SET (ℓ_x, J), C(N+1), ..., C(Z)}

go to Done] otherwise

If ℓ_j or ℓ_k is not zero (assume ℓ_j) then

[TRACE (ℓ_x, J) = {INITIAL.SET (ℓ_x, J), S(i), C(N+1), ..., C(Z)}

Relabel ℓ_j as ℓ_i.

go to 2]

6. ℓ_i is the input link of a distributor D(i). If any of the output links of D(i) are zero (assume ℓ_j) then {INITIAL.SET (ℓ_x, J) = {C(1), ..., C(N), D(i)}

TRACE (ℓ_x, J) = {INITIAL.SET (ℓ_x, J), C(N+1), ... C(Z)}

go to Done] otherwise

If none of the output links of D(i) are zero choose any one of the output links (assume ℓ_j) then

[TRACE (ℓ_x, J) = INITIAL.SET (ℓ_x, J), D(i), C(N+1), ..., C(Z);

Relabel ℓ_j as ℓ_i

go to 2.]

DONE TRACE (ℓ_x, J) is completed.

The term output functional which is used in the following paragraphs also implies that the jobsin a graph are noninteracting. That is, each job takes the same path as it would if it where the only job in the graph.

In step 1 if ℓ_i is zero then TRACE (ℓ_i, J) is the empty set and there is no problem.

In step 2 this link must be an output interface link.

Proof: It will be assumed that it is not an output interface link and it will be shown that this leads to a contradiction.

The value in ℓ_i must be the same as in experiment 1 for this job because the graph is output functional. When this job is finished in experiment 1 this value will be in ℓ_i because ℓ_i is not the input link of any c-operator. But according to experiment 1 this is not possible because only the output interface links can have nonzero values. Therefore there is a contradiction and ℓ_i is an output interface link.

In step 2 ℓ_i will have the value zero except in the degenerate case TRACE (ℓ_i, J) where it may be nonzero.

Proof: It will be assumed that ℓ_i is nonzero for the non-degenerate case. It will then be shown that this leads to a contradiction.

ℓ_i is the output link of some c-operator C(i) which wishes to transform from its input links to ℓ_i. This is by definition of TRACE (ℓ_x, J). According to definition 2.8 each job has its own unique set of output

interface links. Therefore the same job that put the value in ℓ_i is trying to put a second value into ℓ_i. Since the graph is output functional the same condition happens in experiment 1. But this is a contradiction because the job in experiment 1 did not hangup. Therefore the link ℓ_i is zero.

In step 3 a function operator F(i) in standard form is defined as having a set of internal links which can store every value that appears on its input links. When the proper values appear on its internal links F(i) will transform the value on those links to its output links. F(i) is an element of INITIAL.SET (ℓ_i, J) according to the definition of TRACE (ℓ_i, J).

In step 5 the transformations of a data and control selector are such that only one of its two output links can have a nonzero value.

It has now been established that when TRACE (ℓ_i, J) is completed (go to Done in the ALGORITHM). INITIAL.SET (ℓ_i, J) will contain the proper set of c-operators, that is c-operators whose output links are zero or function operators in standard form.

Now it must be established that the ALGORITHM will terminate for every TRACE (ℓ_i, J)

Proof: Assume the contrary that TRACE (ℓ_i, J) does not terminate and show that this leads to a contradiction.

If TRACE (ℓ_i, J) does not terminate there is an infinite number of c-operators in the graph or some c-operator is repeated in TRACE (ℓ_i, J). The first case is impossible because the graph is finite and the second is impossible because the graph is loop-free. Therefore there is a contradiction and every TRACE (ℓ_i, J) terminates (goes to Done in the ALGORITHM)

Because the definition of TRACE (ℓ_i, J) is completely general every link ℓ_i in every STATE (J) $\in$SET.STATES has a TRACE (ℓ_i, J).

We are now finally ready to prove that a loop-free pipeline graph will not hangup (theorem 2.3). A c-operator will hangup if it has the proper set of values in its input links but its set of output links never have the value zero. A c-operator will hangup if it has the improper values in its input links.

It will now be proven that the values in the output links ($\ell_1, \ldots, \ell_n$) of an arbitrary c-operator C(i) in a loop-free pipeline graph which is in an arbitrary state STATE(J) will be transformed to zero. In the degenerate case where the output links are the output interface links there is no problem because if they have a nonzero value then the graph has finished processing that job.

Choose ℓ_1 and Run TRACE (ℓ_1, J). ℓ_1 now has the value zero. Next choose ℓ_2 and Run TRACE (ℓ_2, K). ℓ_2 now has the value zero. ℓ_1 still has the value zero. Since $\ell_1, \ldots, \ell_n$ are the output links of a common c-operator C(i) the only way that TRACE (ℓ_2, K) could make ℓ_1 nonzero is if there was a loop back to C(i) from ℓ_2. The graph is loop-free therefore ℓ_1 remains zero. This reasoning is the same for the other output links. Therefore the values in the output links of an arbitrary c-operator can be set to zero.

It will now be proven that every c-operator C(i) has the proper values on its input links. The data selector, control selector, and distributor have only one input link. If its value is zero there is no problem. If it is nonzero then the c-operator is ready to transform. The merger selects only one input at a time therefore the same reasoning applies.

A function operator in standard form by definition can store every value that appears on its input links into its internal links. Therefore it can make a transform from its input links to its internal links for every value that appears in its input links. When the proper values are stored in its interral links the function operator will transform them to its output links.

The proper values will appear in a finite amount of time.

Proof: It is assumed that the values will not appear in a finite amount of time and is shown that this leads to a contradiction.

Case 1. Some c-operator is hung up and the necessary value can not reach the function operator. It has already been proven that no c-operator will hangup.. Every c-operator has the proper set of values on its input links and its output links can be set to zero.

Case 2. There is not a directed path from the value to the function operator in question. Since the graph is output functional this means that there would not be a directed path in experiment 1 which is a contradiction.

Theorem 2.3 is now proven. Every c-operator can transform and the graph will not hangup.

HANGUPS IN LOOPS

So far it has been shown that hangups can be caused by function operators. A function operator in standard form has been defined and it has been proven that a loop-free pipeline graph with function operators in standard form will not hangup (theorem 2.3). Now the problems of loop structures will be considered.

Referring to Figure 2.14 assume that the loop is embedded in a pipeline graph. M is a merger, F1, F2, and F3 are function operators in standard form and DS is a data selector. Assume that there are N jobs ($N > 5$) which want to enter the loop through link ℓ_1.

M can arbitrarily pick either input link ℓ_1 or ℓ_7 to transform to its output link ℓ_2. It will be assume that it always picks ℓ_1 when there is a choice. This means that eventually five jobs enter the loop. Assuming that no jobs leave the loop there will be a nonzero value in links ℓ_2, ℓ_3, ℓ_4, ℓ_6 and ℓ_7. M cannot transform again until its output link ℓ_2 is zero. ℓ_2 will not be zero until F2 transforms. F2 cannot transform until DS does which cannot until F3 does which cannot until M does. The loop is hung up.

SIMPLE LOOP IN STANDARD FORM

It is the purpose of this section to define a restricted class of loops that will not hangup.

Definition 2.9: A p-merger (PM) is a merger with a preferred set of input links. Referring to figure 2.15 the dots (...) in the p-merger symbol indicate that this set of input links (b_1, .., b_r) is the preferred set. PM will not transform the values in i_1, .., i_k until b_1,..., b_r all have the value zero.

It is defined as an ordered 5-tuple PM = < L, I, B, P, J > where:

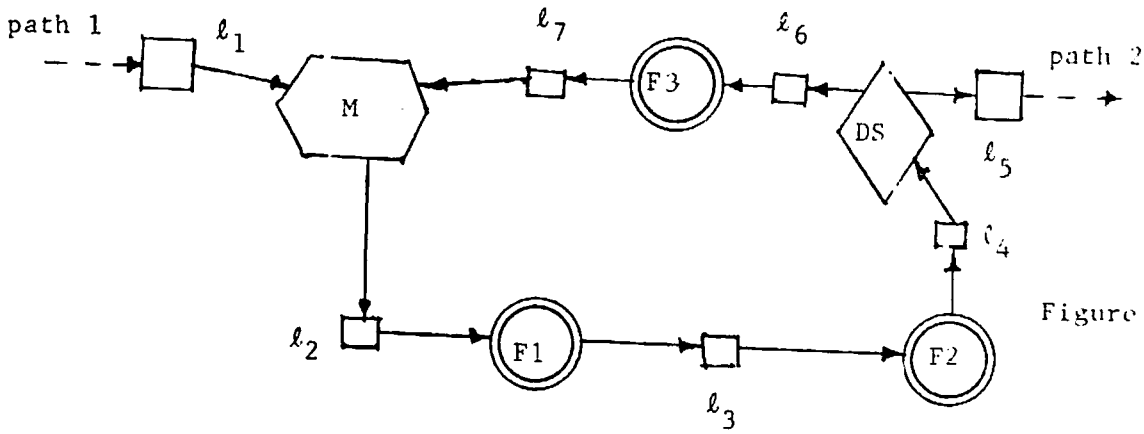


Figure 2.14

LOOP

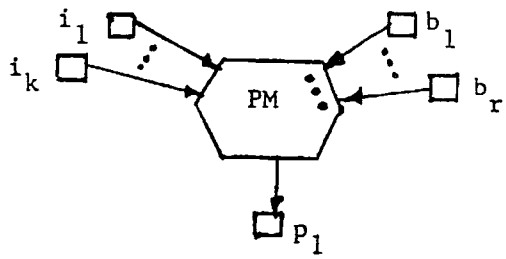


Figure 2.15

P-MERGER

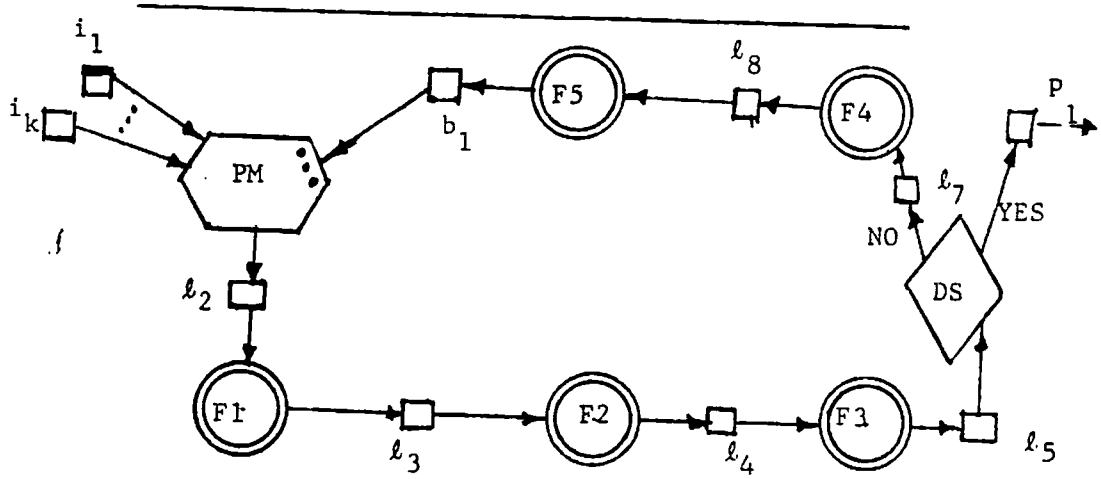


Figure 2.16

SIMPLE LOOP IN STANDARD FORM

1) $L = \{\ell_1, \ell_2, \ldots, \ell_z\}$ is its finite set of links

2) $I = \{i_1, i_2, \ldots, i_k\}$ is its finite set of non-preferred input links

3) $B = \{b_1, b_2, \ldots, b_r\}$ is its finite set of preferred input links

4) $P = \{p_1\}$ is its single output link

5) $J = \{j_1, j_2, \ldots, j_r\}$ is the finite set of values (job numbers) that can appear in the links in the set $I \cup B$

6) $L = I \cup B \cup P$

The function associated with the p-merger is:

$PM{:}D_1(PM) \rightarrow R_1(PM)$

$PM{:}D_2(PM) \rightarrow R_2(PM)$

$\vdots$

$PM{:}D_n(PM) \rightarrow R_n(PM)$

where

$$D_i(PM) = \{< V(\ell_k) = j_i, v(p) = 0 > \mid \pi\, \ell_k\, [(v(\ell_k) = j_i)\ \&\ (j_i \in J)\ \&\ ((\ell_k \in B) \vee (\ell_k \in I)\ \&\ (V(B) = 0))]\}$$

$$R_i(PM) = \{< v(\ell_k) = 0, v(P) = j_i > \mid (j_i \in J)\ \&\ (\ell_k \in I \cup B)\}$$

D_i and R_i refer to the identical j_i and ℓ_k. Every R_k and D_k is of the above form. The only difference is that j_k is used instead of j_i and possibly a different ℓ_k is used.

Definition 2.10 A simple loop in standard form (figure 2.16) is defined as an ordered 5-tuple $SL = < L, I, P, C, J >$ where

1) $L = \{\ell_1, \ell_2, \ldots, \ell_z\}$ is its finite set of links

2) $I = \{i_1, i_2, \ldots, i_d\}$ is its finite set of input links and are the nonpreferred input links of a single merger

3) $P = \{p_1\}$ is its single output link and is the output link of a data selector.

4) $C = \{c_1, c_2, \ldots, c_m\}$ is its finite set of c-operators and consists of one merger, one data selector and M-2 function operators in standard form which have a single input and a single output link. The merger has only one preferred input link b_1.

5) $J = \{j_1, j_2, \ldots, j_n\}$ is the finite set of values (job numbers) that can appear in the links in the set 1

The input to the loop is through the non-preferred links of the merger. The exit to the loop is through the output link of the data selector. The loop originates on the output link of the merger and terminates on the preferred link of the merger.

Theorem 2.4 A simple loop in standard form will not hangup.

Proof: The proof will be shown for the special case in figure 2.16 but it is completely general and will apply to any loop of that form. It will be assumed that all the links (ℓ_2, ℓ_3, ℓ_4, ℓ_5, ℓ_7, ℓ_8 and b_1) in the loop except one has a nonzero value. Each value is unique and represents a job in the loop. It will be shown that one more job cannot enter the loop and that this is sufficient to prevent a hangup.

The only way that another job can enter the loop is for ℓ_2 to have the value zero. PM would not otherwise be able to transform. If ℓ_2 is the link with the zero value then b_1 must have a nonzero value. Only one link in the loop is zero. PM by definition will choose the preferred link b_1 and therefore a new job cannot enter the loop. It has now been shown that a loop of the form of figure 2.16 will always have at least one link with the value zero. A TRACE (ℓ_i, J) which terminates on the c-operator

whose output link is the one with the zero value or whose output link is ℓ_6 can be constructed for every link in the loop. The output link of every c-operator in the loop can therefore be set to zero. No c-operator in the loop will hangup.

COMPLEX LOOP IN STANDARD FORM

The simple loop in standard form which has just been considered is too restrictive. A standard form for complex loops will now be defined.

Definition 2.11; A p-merger - distributor structure consists of one p-merger and one distributor (figure 2.17). It is defined as an order 5-tuple PMD = $< L, C, I, B, J >$ where

1) $L = \{\ell_1, \ell_2, \ldots, \ell_z\}$ is its finite set of links

2) $C = \{PM, D\}$ is its set of two c-operators (one p-merger and one distributor)

3) $I = \{i_1, i_2, .., i_d\}$ is the set of nonpreferred input links of the p-merger

4) $B = \{b_1, b_2, \ldots, b_r\}$ is the set of preferred input links of the p-merger and is the set of output links of the distributor

5) $J = \{j_1, j_2, \ldots, j_n\}$ is the finite set of values (job numbers) that can be in the links of the set I.

6) $L \supset I \cup B$

Every output link of the distributor is a preferred input link of the p-merger and every preferred link of the p-merger is the output link of the distributor.

Definition 2.12: A complex loop in standard form (figure 2.18) is defined as an ordered 4-tuple CL = $< L, C, X, J >$ where

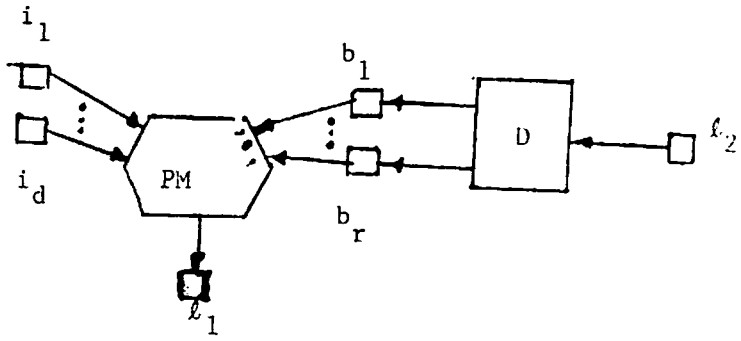


Figure 2.17

P-MERGER - DISTRIBUTOR STRUCTURE

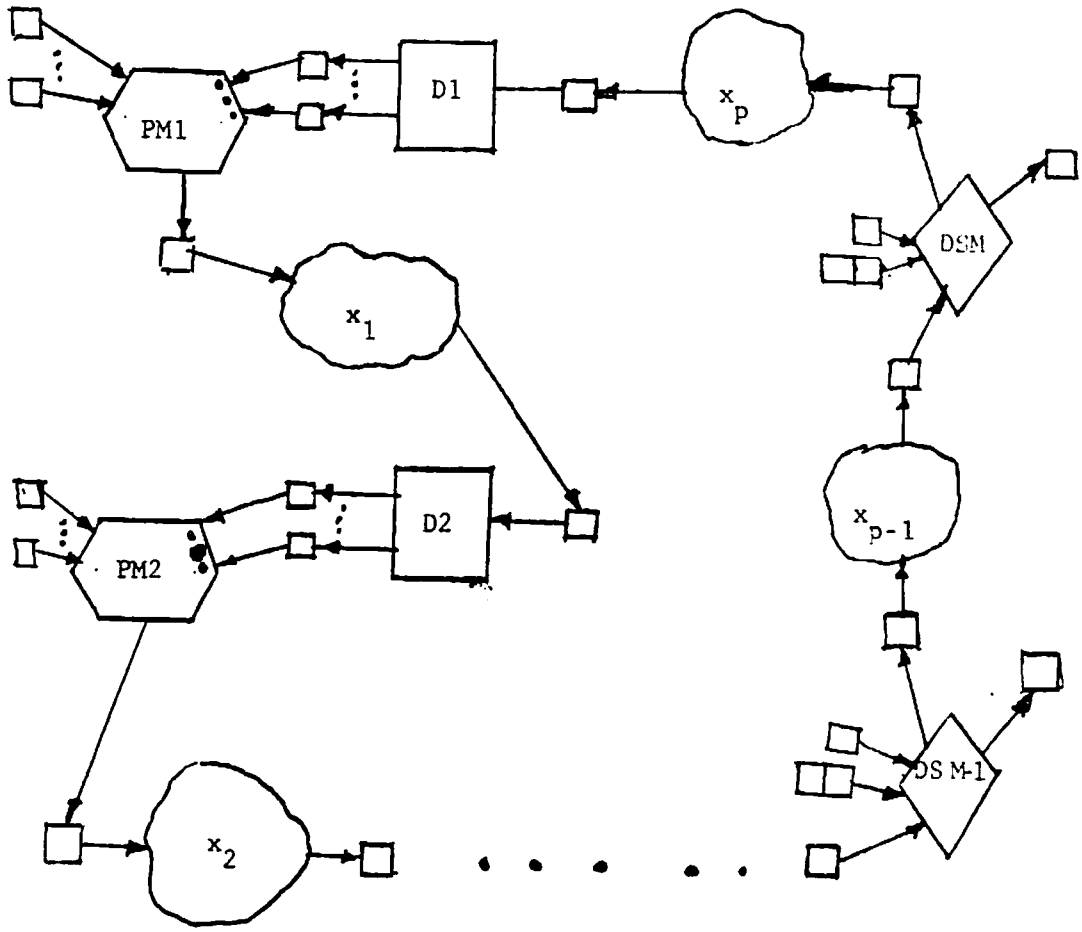


Figure 2.18

COMPLEX LOOP IN STANDARD FORM

1) $L = \{\ell_1, \ell_2, \ldots, \ell_z\}$ is its finite set of links

2) $C = \{c_1, c_2, \ldots, c_k\}$ is its finite set of p-merger-distributor structures and data selectors

3) $X = \{x_1, x_2, \ldots, x_p\}$ is its set of pipeline graphs. Each x_i has one input link, one output link and does not hangup when processing K jobs simultaneously.

4) $J = \{j_1, j_2, \ldots, j_n\}$ is the finite set of values (job numbers) that can be in the links of the set L.

5) The graphs x_i ($x_i \in X$) and the c-operator c_i ($c_i \in C$) are connected to form a single series path in the loop.

6) The only entrances to the loop are through the non-preferred links of the p-merger-distributor structures c_i ($c_i \in C$).

7) The only exits from the loop are through the output link of a data selector c_i($c_i \in C$).

8) Let R equal the sum of the maximum number of jobs that each x_i can process simultaneously.

$$R = \sum_{i=1}^{p} \text{MAX JOBS}(x_i)$$

Every distributor in a p-merger-distributor structure has R+1 output links, where R+1 is the same for every p-merger-distributor c_i($c_i \in C$).

Theorem 2.5: A complex loop in standard form will not hangup.

Proof: It will be proven that a complex loop in standard form will always have R free links and that this is sufficient to prevent a hangup.

1. A complex loop will always have R free links.

   Proof: Referring to figure 2.18 it will be assumed that all jobs enter the loop at PM1. All PM1's R+1 preferred input links must have the value zero if a job is to enter the loop. Jobs are now sent into the loop. In the worst case every x_i is processing its maximum number of jobs. Every other link has a nonzero value (job) except the R+1 preferred links of PM1. Now one more job enters the loop. It must go in one of the preferred links of PM1. Since PM1 has a job on one of its preferred links no new jobs can enter the loop. No new jobs can enter from any other PMi because they have jobs on all of their preferred links. Thus now there are R free links in the complex loop.

2. The R links are sufficient to prevent a hangup.

   Proof: Assume that the maximum number of jobs are in the loop. There are R empty links. In the worst case each x_i is such that no new jobs can enter x_i until all the jobs in x_i have left. If this happens to all the x_i's then the equivalent of R new jobs have been sent into the loop. There is a TRACE (ℓ_i, J) from the output link of each x_i to the distributor with R empty output links. The R jobs from the x_i's can therefore be stored in these links. Each of the x_i's is now empty and will accept at least one new job. The proof is now the same as for a simple loop in standard form. The complex loop in standard form will therefore not hangup.

It was assumed that each x_i was hangup free. This means that it could be a loop-free structure in standard form, a simple loop in standard form, or a complex loop in standard form. Complex loops are therefore very general in nature.

CHAPTER III

HIERARCHICAL DESIGN

The thesis has so far concentrated on the problems involved in designing a control system (pipeline graph) which can control the processing of N independent jobs simultaneously. In software terminology each pipeline graph when properly connected to a processing structure [7,8] is equivalent to a subroutine. The problem that will now be considered is that of structuring these subroutines together to form a larger program. The main problem that will be investigated is the communications between pipeline graphs.

This chapter will later consider a pipeline graph which computes the tangent of a number. In the process of computing the tangent, the graph sends control to two other pipeline graphs which compute the sine and cosine. There are a number of problems involved. It is possible that there will be a number of simultaneous requests for the use of a pipeline graph. If the requested graph can process more than one job simultaneously, then there is the problem of distributing requests and returning the answers to the proper requesters. To solve the above problems two new c-operators will be defined. It will then be shown that interacting pipeline graphs can be connected together.

This is a modular design approach. The basic building blocks are the pipeline graphs considered in the last chapter. These are interconnected to form complex processing structures. The advantage to this approach is that the pipeline graphs can be built and debugged without considering the larger structure in which they will be embedded.

EXTENDED FUNCTION OPERATOR

The extended function operator (EF-operator) is used to place the proper job into the set of input interface links of a pipeline graph. When that job number appears on the graph's set of output interface links, the EF-operator sends control to the c-operator which is connected to its output links.

The EF-operator (figure 3.1) is defined as an ordered 7-tuple $EF = < L, I, P, A, B, S, J >$ where:

1) $L = \{\ell_1, \ell_2, \ldots, \ell_z\}$ is its finite set of links

2) $I = \{i_1\}$ is its single input link

3) $P = \{p_1\}$ is its single output link

4) $A = \{a_1, a_2, \ldots, a_k\}$ is its finite set of initiation links

5) $B = \{b_1, b_2, \ldots, b_m\}$ is its finite set of completion links

6) $S = \{J1\}$ is the single job number which is put into the initiation links and detected in the completion links

7) $J = \{j_1, j_2, \ldots, j_n\}$ is its finite set of job numbers that can appear in the links of the set I

8) $L = I \cup P \cup A \cup B$

The function associated with the EF-operator is:

$EF: D_1^1 (EF) \rightarrow R_1^1(EF)$

$EF: D_1^2 (EF) \rightarrow R_1^2(EF)$

$EF: D_2^1 (EF) \rightarrow R_2^1(EF)$

$EF: D_2^2 (EF) \rightarrow R_2^2(EF)$

$\vdots$

$EF: D_n^1 (EF) \rightarrow R_n^1(EF)$

$EF: D_n^2 (EF) \rightarrow R_n^2(EF)$

$D_i^1(EF) = \{< v(I) = j_i, v(A) = 0, v(B) = 0, v(P) = 0 > \mid j_i \in J\}$

$R_i^1(EF) = \{< v(I) = j_i, v(A) = J1, v(B) = 0, v(P) = 0 > \mid (j_i \in J) \& (J1 \in S)\}$

$D_i^2(EF) = \{< v(I) = j_i, v(A) = 0, v(B) = J1, v(P) = 0 > \mid j_i \in J\}$

$R_i^2(EF) = \{< v(I) = 0, v(A) = 0, v(B) = 0, v(P) = j_i > \mid j_i \in J\}$

There are two domains D_i^1 and D_i^2 and two ranges R_i^1 and R_i^2 associated with each job number j_i ($j_i \in J$). D_i^1, D_i^2, R_i^1, and R_i^2 refer to the identical job number j_i. Every D_k^1, D_k^2, R_k^1, and R_k^2 is of the above form. The only difference is that j_k is used instead of j_i.

The value J1 (figure 3.1) is the job number which is placed in the initiation links and detected in the completion links. When the pipeline graph is connected to a processing structure, the initiation links start the processing and the completion links detect completion of the processing.

Figure 3.2 illustrates the connection of EF operators to a pipeline graph P1 which can process two jobs simultaneously. Note that once a job enters a set of input interface links a second job cannot enter those same links until the first job has left the pipeline graph.

EXTENDED CONTROL SELECTOR

The extended control selector (EC-operator) is used to send jobs to the proper c-operator. If a pipeline graph is processing a number of jobs, the EC-selector can be used to channel the output of the graph for each job to the c-operators which had requested that processing.

The EC-operator (figure 3.3) is defined as an ordered 4-tuple $EC = < L, I, P, J >$ where:

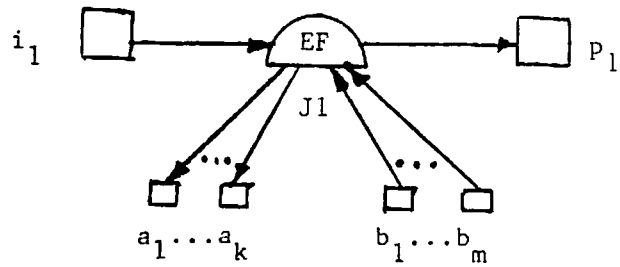


Figure 3.1

TRANSFORMATION

i_1	a_1	...	a_k	b_1	...	b_m	p_1	→	i_1	a_1	...	a_k	b_1	...	b_m	p_1
X	0	...	0	0	...	0	0	→	X	J1	...	J1	0	...	0	0
X	0	...	0	J1	...	J1	0	→	0	0	...	0	0	...	0	X

EXTENDED FUNCTION OPERATOR

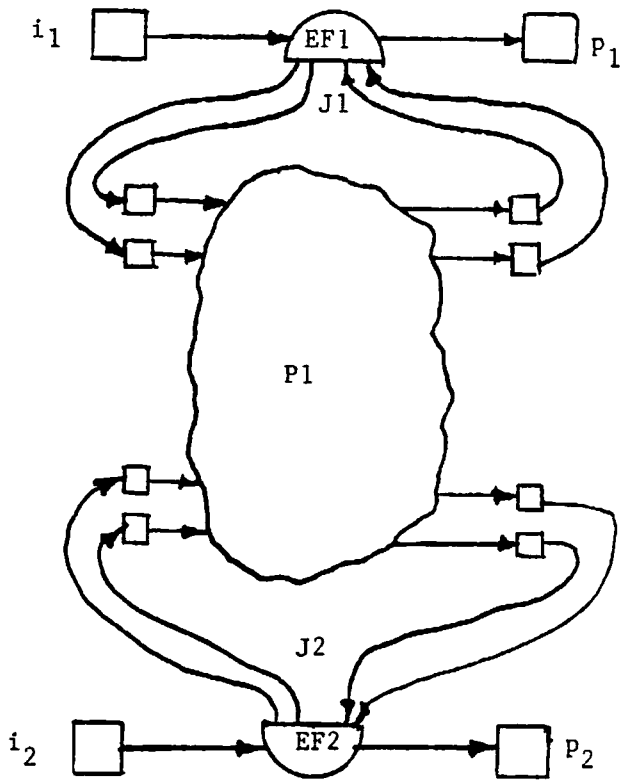


Figure 3.2

1) $L = \{\ell_1, \ell_2, \ldots, \ell_n\}$ is its finite set of links

2) $I = \{i_1\}$ is its single input link

3) $P = \{p_1, p_2, \ldots, p_n\}$ is its ordered finite set of output links

4) $J = \{J1, J2, \ldots, JN\}$ is an ordered finite set where

$$J1 = \{j_1, \ldots, j_k\}$$

$$\vdots$$

$JN = \{j_a, \ldots, j_b\}$ is a unique finite set of job numbers

5) $L = I \cup P$

The number of elements in P is identical to the number of elements in J. Each JK is a finite set of job numbers. J1 is associated with output link p_1. J2 is associated with output link p_2. $\cdots$ JN is associated with output link p_n.

The function associated with the EC-operator is:

$$EC{:}D_1(EC) \rightarrow R_1(EC)$$

$$EC{:}D_2(EC) \rightarrow R_2(EC)$$

$$\vdots$$

$$EC{:}D_N(EC) \rightarrow R_N(EC)$$

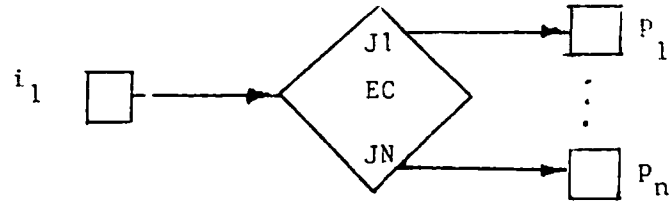
where

$$D_i(EC) = \{< \ v(I) = j_k, v(P) = 0 > \mid j_k \in J\}$$

$$R_i(EC) = \{< v(I) = 0, v(p_i) = j_k > \mid \exists\, p_i(p_i \in P) \ \& \ (j_k \in JI)\}$$

There is one domain D_i and one range R_i associated with each output link $p_i(p_i \in P)$. D_i and R_i refer to the identical job number j_k. Every D_k and R_k is of the above form. The only difference is the p_k is used instead of p_i.

The EC-operator transforms the value (job number) in its input link to the output link that has that job number associated with it.



EXTENDED CONTROL SELECTOR TRANSFORMATION

$$i_1\ p_1 \cdots p_{i-1}\ p_i\ p_{i+1} \cdots p_n \rightarrow i_1\ p_1 \cdots p_{i-1}\ p_i\ p_{i+1} \cdots p_n$$

$$X\ 0 \ldots 0 \quad 0\ 0 \quad \ldots 0 \rightarrow 0\ 0 \ldots 0 \quad X\ 0 \quad \ldots 0$$

Where X matches the job number associated with p_i.

RESTRICTIONS:

1. Each output link has one or more unique job numbers associated with it.
2. The set of job numbers associated with the output links contains every job number that could appear in the input link.

Figure 3.3

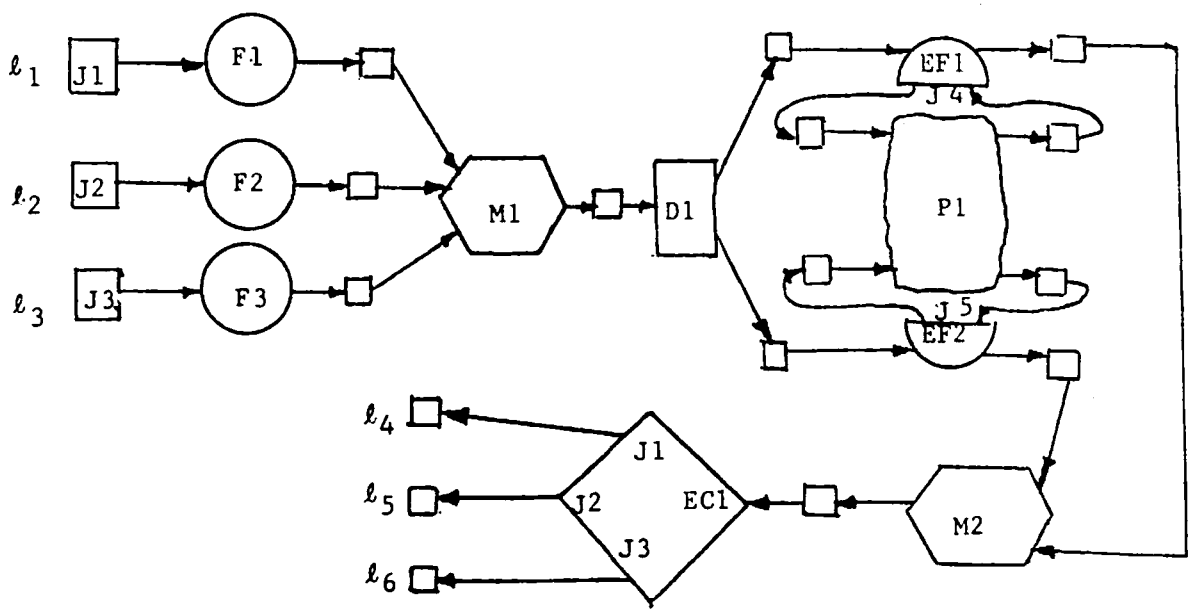


Figure 3.4

Figure 3.4 illustrates the use of an EC-selector. F1, F2, and F3 are function operators, M1 and M2 are mergers, D1 is a distributor, EC1 is an EC-selector and P1 is a pipeline graph. ℓ_1 and ℓ_4, ℓ_2 and ℓ_5 and ℓ_3 and ℓ_6 are the input-output interface links for jobs J1, J2, and J3 respectively. Each of the jobs is first processed by its corresponding function operator. Next the three jobs are processed by P1 which can process two jobs simultaneously. Finally EC1 sends each job to its proper output interface link.

PIPELINE GRAPH IN STANDARD FORM

A pipeline graph in standard form (figure 3.5A) is defined as an ordered 7-tuple SF = $< L, C, I, P, X, D, J >$ where:

1) $L = \{\ell_1, \ell_2, \ldots, \ell_z\}$ is its finite set of links.

2) $C = \{c_1, c_2, \ldots, c_m\}$ is a finite set of c-operators consisting of two mergers (M1, M2), one distributor (D1), one EC-operator (EC1) and M-4 EF-operators.

3) $I = \{i_1, i_2, \ldots, i_b\}$ is its finite set of input links

4) $P = \{p_1, p_2, \ldots, p_b\}$ is its finite set of output links

5) $X = \{x_1\}$ is the single pipeline graph which can process k jobs simultaneously.

6) $D = \{d_1, d_2, \ldots, d_g\}$ is the set of dynamic input-output interface links for the graph x_1 ($x_1 \in X$)

7) $J = \{j_1, j_2, \ldots, j_n\}$ is its finite set of job numbers that can appear in the links of the set I.

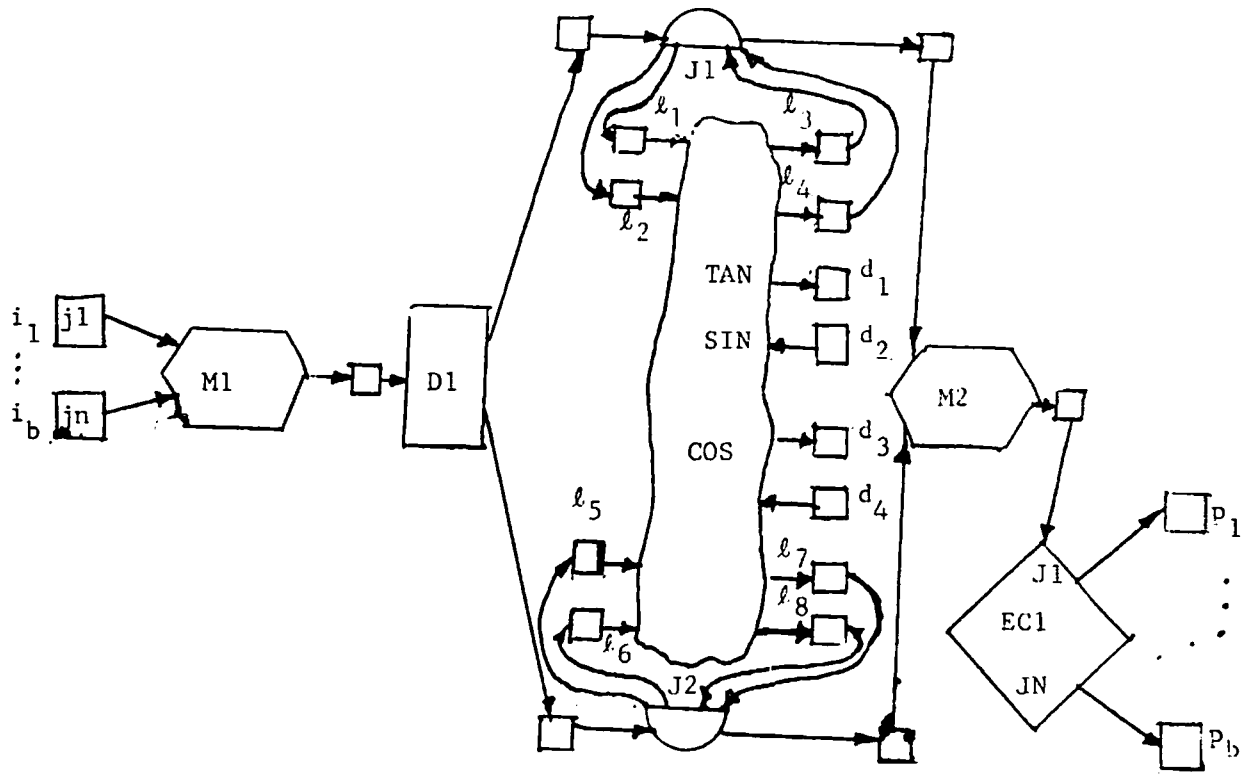


Figure 3.5A

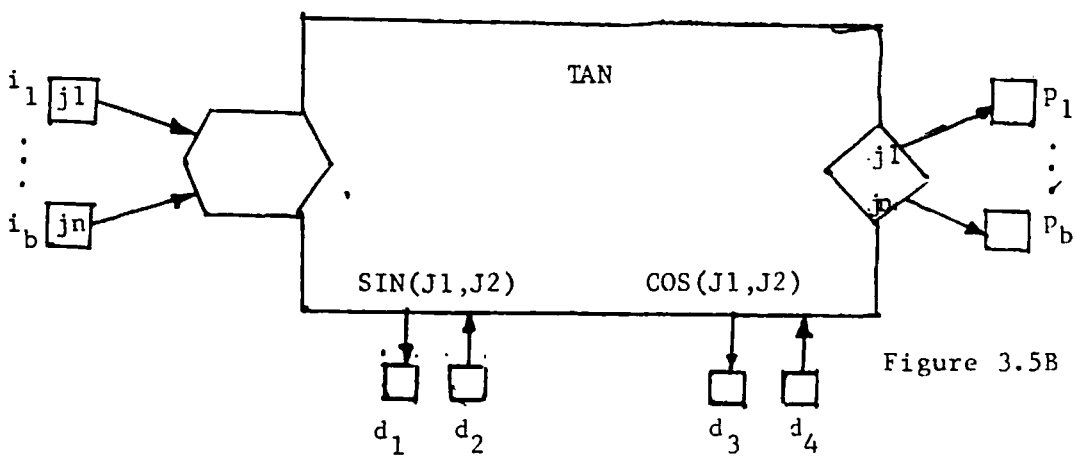


Figure 3.5B

PIPELINE GRAPH IN STANDARD FORM

I is the set of input links of a single merger M1 (M1 $\in$ C). The output link of M1 is the input link of distributor D1 (D1 $\in$ C). The output links of the distributor are the input links of the M-2 EF-operators (EF $\in$ C). The output links of the EF-operators are connected to a single merger M2(M2 $\in$ C). M2 in turn is connected to an EC-operator EC1(EC1 $\in$ C). The output links of EC1 are the set P. Each set of input-output interface links for each job that the graph x_1 can process simultaneously is connected to an EF-operator. The dynamic input-output interface links are the set of links on the graph x_1 which are used to request processing by other pipeline graphs.

Figure 3.5A illustrates a pipeline graph in standard form. TAN is a pipeline graph constructed according to the rules discussed in the previous chapter. It can process two jobs simultaneously, where ℓ_1, ℓ_2, ℓ_3 and ℓ_4 are the input-output interface of job J1 and ℓ_5, ℓ_6, ℓ_7 and ℓ_8 for job J2. Links d_1, d_2, d_3, and d_4 are the dynamic input-output interface links. They request processing other pipeline graphs.

The symbol for a pipeline graph in standard form is illustrated in figure 3.5B. SIN(J1, J2) means that the set of dynamic input-output interface links (d_1, d_2) request processing from the pipeline graph labeled SIN and will transmit either the job number J1 or J2.

A SPECIAL PURPOSE COMPUTER

This section will illustrate the design of a special purpose computer which can calculate the sin, cos, tan, cot, sec, and csc trigonometric functions. The basic building blocks are pipeline graphs to compute

each of the functions. The tan, cot, sec, and csc can be defined in terms of the sin and cos. Their pipeline graphs will send control to the sin and cos graphs in the process of computing their respective functions. Each of the pipeline graphs is put into standard form. They are interconnected as shown in figure 3.6. To simplify the diagram a central channel is shown. Each number is the channel appears twice and represents a connection between those two points. The job numbers are not shown in the diagram.

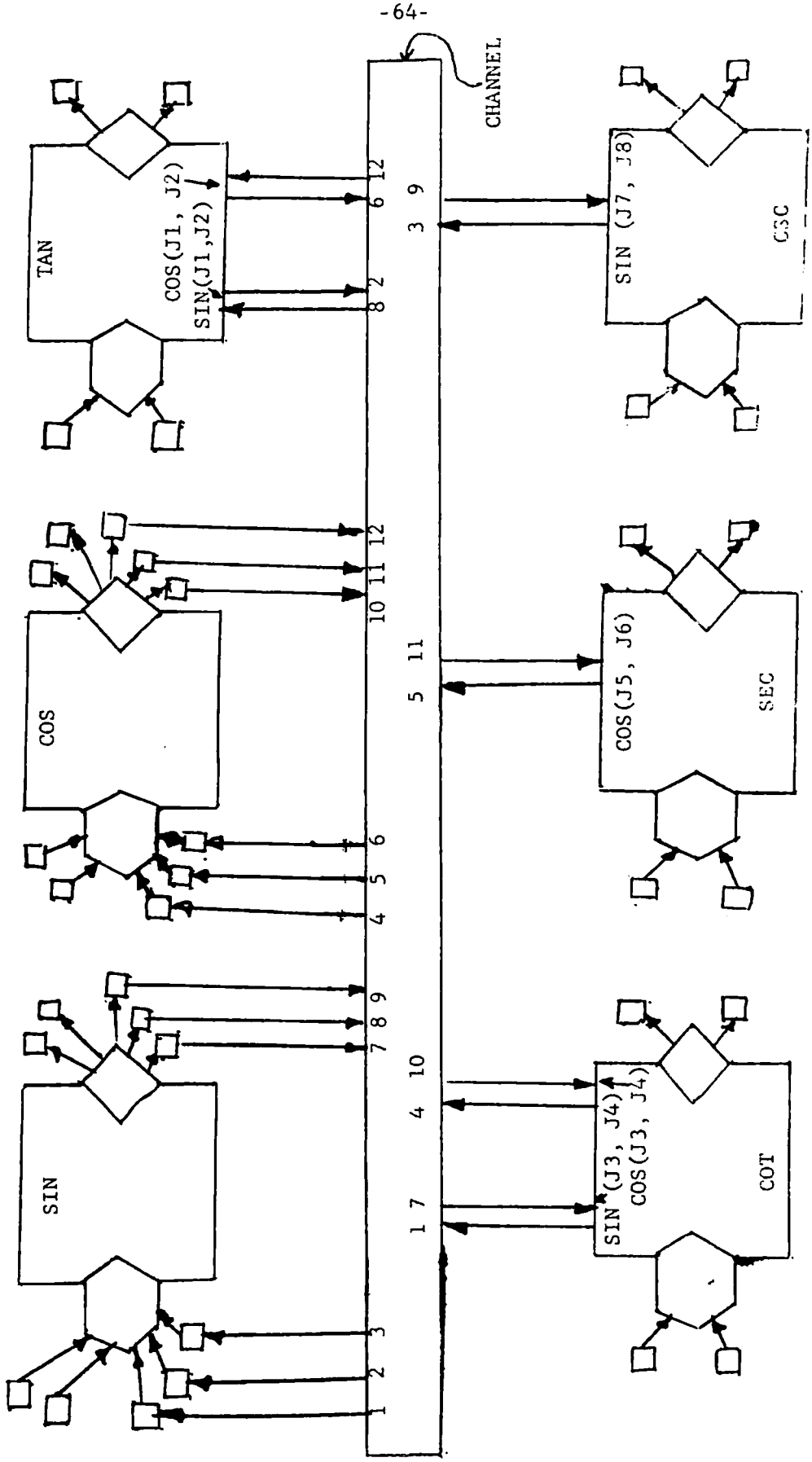

CHANNEL
TAN
COS(J1, J2)
SIN(J1,J2)
SIN (J7, J8)
CSC
COS
COS(J5, J6)
SEC
SIN
SIN (J3, J4)
COS(J3, J4)
COT
1
2
3
4
5
6
7
8
9
10
11
12

Figure 3.6

CHAPTER IV

CONCLUSION

There are a number of applications for a pipeline graph. One is in the design of special purpose computers such as considered in the last chapter. An important application which will be considered here is their use in the central processor of a general purpose computer.

The processing units in a central processor are adders, subtractors, multipliers, dividers, etc. All computer languages must be written in terms of these operations either directly or indirectly. Assembly languages are written directly in terms of these operations and high level languages such as FORTRAN IV must be compiled, that is translated into an assembly language. Since most programming is done in terms of high level languages the compiler has become an important factor in limiting the power of computer languages. It is inefficient to build up hierarchies of high level languages.

With a system such as the pipeline graph it is possible to wire-in (hardware) a set of processing instructions which are common to most high level languages. This would reduce the importance of the compiler, increase the speed of processing and would make it easier to develop complex programming languages.

There any many problems which need to be investigated for central processors which are controlled by pipeline graphs. The memory-hardware interface needs to be investigated and so does the dynamic sharing of the system by many users.

This thesis did not consider the connection of the pipeline graph to a processing structure. The connection must be made in such a manner as to insure that the theorems on output functionality and hangups still apply.

APPENDIX A

SUMMARY OF NOTATION

$A \in B$	is true if and only if A belong to B
$A \& B$	is true if and only if both A and B are true
$A \vee B$	is true if and only if A or B or both are true
$A \Rightarrow B$	means if A is true then B is true
$\forall x$	means for all x
$\exists x\ A$	means that there exists an x such that A is true
$v(I) = x$	means that the value of every element in the set I is identically equal to the value x
$v(\ell_i) = x$	means that the value of ℓ_i is identically equal to the value x
$D(F)$	means the domain of F
$R(F)$	means the range of F
$F:D(F) \rightarrow R(F)$	defines the transform F
$v(b_1)\ R\ V_i$	means that the value of b_1 is in the range of the value V_i
$v(b_1)\ \not{R}\ V_i$	means that the value of b_1 is not in the range of the value V_i

REFERENCES

1. IBM Journal of Research and Development, Vol. 11, No. 1, January 1967.

2. Hassitt, A., Computer Programming and Computer Systems, Academic Press, 1967

3. Corbato, F.J. and Vyssotsky, V.A., Introduction and Overview of the Multics System, AFIPS Conference Proceedings, (FJCC) Vol. 27, Spartan Books, 1965, pp 185-196.

4. Slotnick, D.L. and Borck, W.C., The Solomon Computer, AFIPS Conference Proceedings, (FJCC) Vol. 22, Spartan Books, 1962, pp 97-107.

5. Cotten, L.W., Circuit Implementation of High-Speed Pipeline Systems, AFIPS Conference Proceedings, (FJCC) Vol. 27, Spartan Books, 1965, pp 489-504.

6. Rodriguez, J.E., A Graph Model for Parallel Computations, Doctoral Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering, September 1967.

7. Luconi, F.L., Asynchronous Computational Structures, Doctoral Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering, January 1968.

8. Clark, W.A., Macromodular Computer Systems, AFIPS Conference Proceedings, (SJCC) Vol. 30, Thompson Books, 1967, pp 337-349.