TOWARD A HARDWARE IMPLEMENTATION OF THE CONTOUR MODEL

By

PU-KOUNG PHILIP TU

Bachelor of Engineering
Chung Yuan Christian College of Science
and Engineering
Taipei, Taiwan, Republic of China
1973

Submitted to the Faculty of the Graduate College
of the Oklahoma State University
in partial fulfillment of the requirements
for the Degree of
MASTER OF SCIENCE
July, 1978

TOWARD A HARDWARE IMPLEMENTATION

OF THE CONTOUR MODEL

Thesis Approved:

G. E. Hedrick
Thesis Adviser

J. R. Phillips

E. L. Shreve

Norman N Durham
Dean of the Graduate College

1012644

ACKNOWLEDGMENTS

I am deeply grateful to my principal adviser, Dr. George Hedrick, for suggesting this problem and for his guidance, concern, and encouragement through this study.

Special thanks to Dr. J. B. Johnston, whose paper became a guidance to this study.

I would like to thank Drs. G. E. Hedrick, John R. Phillips, and Edward L. Shreve, for serving on my advisory committee and for their assistance in correcting the final copy of this thesis.

I also wish to express my thanks to Mrs. Carol Taylor for the excellent typing of this thesis.

I would like to express my gratitude to my wife, Elizabeth, for her encouragement and her confidence in my ability. Finally, I would like to express my gratitude to my parents, Mr. and Mrs. Chi-Lian Tu, for their encouragement and support throughout my study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

In the recent past, use of the stack mechanism in the implementation of programming languages has become popular. A stack is a set of storage locations into which one stores the data into the top pushing down the data already in it, and from which one deletes the top element popping up the elements below. For example, all operations have their implied operands as either the top element of the stack or the top two elements of the stack, the result is returned to the stack and becomes the top element after the operation. The LIFO (last in first out) push down stack is utilized in many types of computers to generate a machine equivalent program. Machines utilizing the stack mechanism in hardware exe-
Stack makes the compilers simpler. Reduce the semantic gap.
cute faster than a software implementation (1). However, Johnston (10) pointed out that stack-related implementations have at least two disadvantages. The first and more serious dis$\overset{ad}{\wedge}$vantage concerns the strictly LIFO nature universally attributed to stacks. Certain unnecessary restrictions are associated with block exit, these restrictions falsely appear to be inherent properties of algorithm execution. The second disadvantage involves the identification of a stack as a locus of control. The identification causes difficulties in the execution of multiple activity processes.

The contour model of block structured processes was first introduced by Johnston in 1971. He first described how the contour model can execute an ALGOL 60 program by using virtual processors. Then he presented the cellular structure, the basic data items, the virtual processor structure, and the access path designators incorporated in the contour model. Finally, he defined the basic set of instructions required to implement the contour model. He also pointed out that the contour model is directly applicable to ALGOL 60, but not to those programming languages which do not have contour retention properties, ALGOL 68 is an example. Hedrick (5) modified the contour model in 1976 by implementing the "heap". He defined the heap as a set of memory locations which is used to reserve space for the allocation of arrays during runtime.

Review of Literatures

First, three examples are used to introduce Johnston's contour model, then to review briefly Gries' stack model. so a comparison between these two models can be made. Finally, the modification made by Hedrick (5) to the contour model is presented.

Introduction to the Contour Model

Johnston (10) defined the contour model: it is a cell-based model of the semantics of algorithm execution in a nested block environment. He also defined a process of the contour model as a sequence of snapshots. Each snapshot has two components: a time-invariant algorithm (static) and a time-varying record of that algorithm (dynamic). Also, each snapshot results from the preceding one by the execution of a single instruction of that algorithm. In the contour model, both the algorithm contour and the record of execution being modeled are block structured processes. As shown in Figure 1, each "begin-end" pair in Figure 1.a corresponds to a contour in Figure 1.b.

Since ALGOL 60 has the necessary contour retention properties, the contour model of block structured processes is directly applicable to it (10). Therefore, it is best to use some ALGOL 60 like programs to describe the contour model. Before doing this, some basic features of the contour model are presented first.

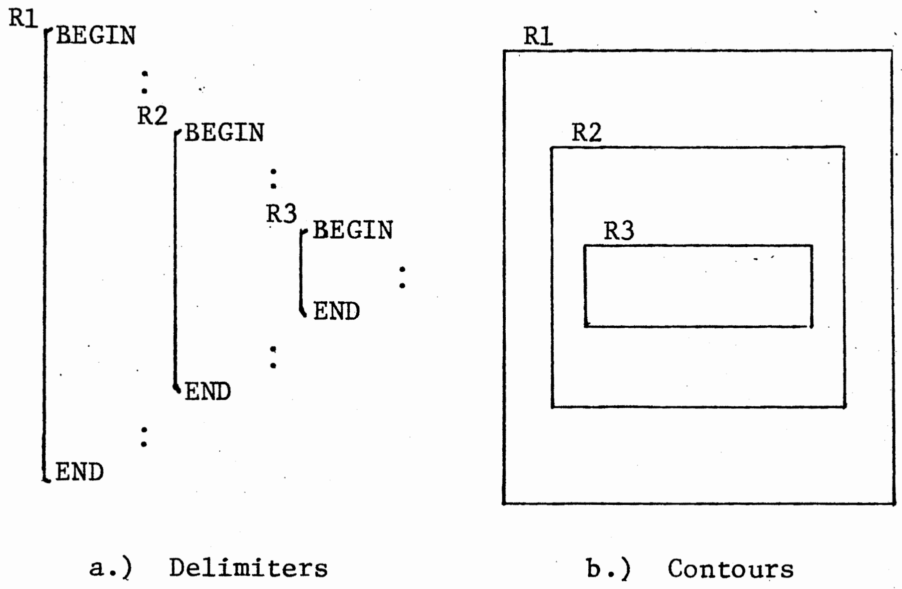


Figure 1. Nested Algorithm Contour

Storage Unit and Data Type. The contour model is a cell-based model of semantics of algorithm contours (10). A cell is a contiguous sequence of memory locations. The general cell format used in this paper is shown in Figure 2. However, it will be modified from data type to data type. The cell is composed of an organization portion and a residence portion. The organization portion of a cell consists of type, size, reference count, inhibit boxes, item present bit, and valid bit. Type is used to specify what kind of data is contained in the cell. Size specifies the number of subcells in the residence of a cell. Reference count is maintianed equal to the number of pointers to that cell for deallocation purposes (10). Inhibit boxes specify what operations cannot happen to that cell, inhibit access and inhibit write are examples. Item present bit and valid bit specify the information stored in that cell is complete and valid respectively. The residence of a cell consists of a sequence of a non-negative integer subcells. Each subcell contains 16 bits. Several subcells together hold one basic datum: an integer, a pointer, or a label. If the number of subcells in a cell is more than two, then it is a compound cell, otherwise it is a single cell. Processors, contours, and instructions are three other special data items appear in this paper.

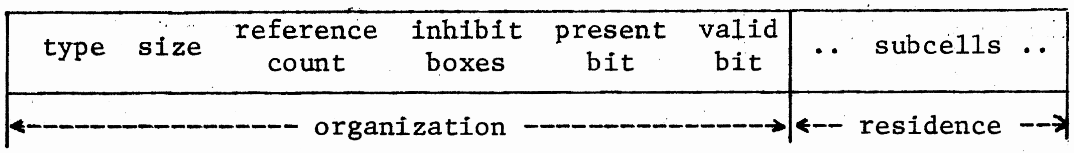


Figure 2. General Cell Format

Further information about the data formation of each different type of data is shown in Figure 3, where PB, VB, and SB are abbreviations of item present bit, valid bit, and sign bit respectively. Also, the number of bits assigned to each subfield is specified on the top of each

16

3	7	4	1	1		16
type (000)	--------	inhibit boxes	P B	V B	S B	value

a.) Integer

3	7	4	1	1	16
type (010)	--------	inhibit boxes	P B	V B	pointer

b.) Pointer

3	7	4	1	1	16	16
type (011)	--------	inhibit boxes	P B	V B	environment pointer	instruction pointer

c.) Label

3	3	3	7	16	16
type (001)	format	tag	--------	static link	successor link

7	9	16	16
instruction height	operation code	(1st) operand if any	(2nd.) operand if any

d.) Instruction

Figure 3. Storage Organization

3	7	5	1	7	9	16	16	
type (110)	reference count	-----	V B	height	size	static link	antecedent link	subcell(s)

e.) Record Contour

3	7	5	1	7	9	16	16	
type (111)	reference count	-----	V B	height	size	static link	antecedent link	subcell(s)

f.) Algorithm Contour

3	7	2	4	16	16	16	16
type (100)	reference count	state	----	P.soa.ep	P.soa.ip	P.lab.ep	P.lab.ip

7	9	16	16	16
instruction height(P.ih)	----------	pointer register (P.ptr)	P.gpr.ptr	P.dsp.ptr

P.gpr.ptr: points to the beginning loc. of gen. purp. regs.

P.dsp.ptr: points to the beginning location of display regs.

P.soa.ep:	environment pointer of site of activity
P.soa.ip:	instruction pointer of site of activity
P.lab.ep:	environment pointer of label register
P.lab.ip:	instruction pointer of label register
P.gpr.ptr:	general purpose register pointer
P.dsp.ptr:	display register pointer

g.) Virtual Processor

Figure 3. (Continued)

subfield on the basis of a 64 K main store is used. In Figure 3, some subfields have not been defined yet, but they will be discussed in detail in the remaining chapters.

Algorithm and Record of Execution. A process is a sequence of snapshots, each contains a static algorithm and a dynamic current state of the record of execution of that algorithm (10). The contour structure of an algorithm functions as a template for the formation of the record contour. During runtime, whenever an ENTER instruction is being executed, a copy of some specific algorithm contour is made. As shown in Figure 4, B1' and B2' are copies of B1 and B2 respectively, and if B2' is immediately enclosed by B1', then B1 should immediately enclose B2. When record contour B2' is copied from algorithm contour B2, the same amount of storage locations occupied by B2 are allocated for record contour B2'. But the contents of these memory locations allocated for B2' may not necessarily be the same as that of B2.

Height and Successor Link. One of the most important features of the contour model is that it stresses quite explicitly the nested contour structure of both the algorithm and records of execution of block structured processes (10). It is the reason that each contour and each instruction must have a height to specify its level. As shown in Figure 4, the height of each contour or each instruction is specified in parentheses. Under normal conditions, the instruction to be executed next is always specified by the successor link of the instruction being executed.

Processor and Virtual Processor. Usually, a processor may be defined as a device which fetches and executes instructions. In the

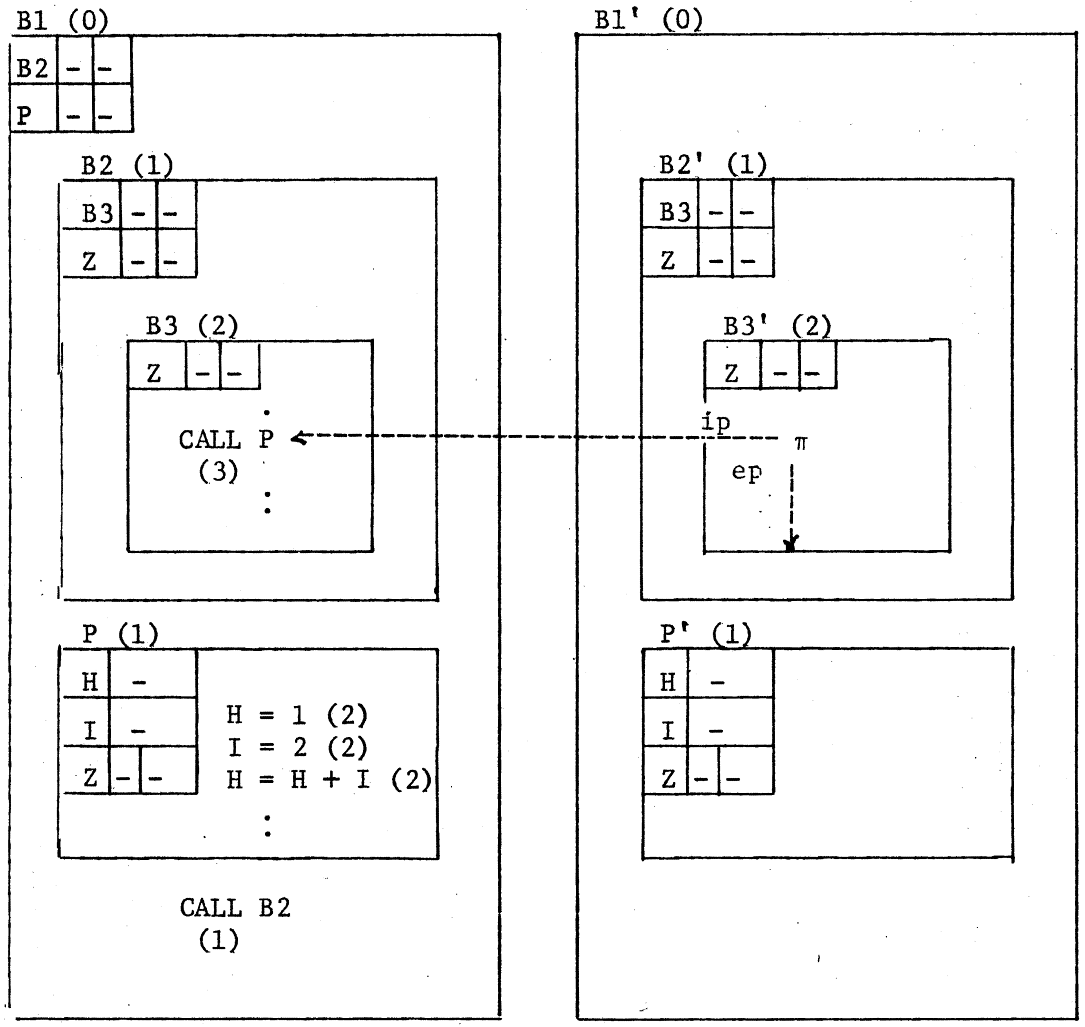


Figure 4. Example of a Snapshot

present context, a processor is a pointer-pair datum, and is denoted by P.soa (site of activity). A P.soa consists of an environment pointer (P.soa.ep) and an instruction pointer (P.soa.ip). The environment pointer and the instruction pointer are two major elements of a processor. They are related to each other as follows: if P.soa.ep is null, then P.soa.ip must point to an instruction which is enclosed by no contour; if P.soa.ep points to a record contour P', and P' is a copy of the algorithm contour P, then P.soa.ip must point to an instruction which is immediately enclosed by P (10). In both cases, P.soa.ip points to the instruction being executed or about to be executed. In the contour model, a processor must be in one of the four states: invalid, awake, asleep, or terminated. As a matter of fact, the processor mentioned here are virtual processors, but they will be treated as processors in the normal sense for convenience in these examples.

Display and Environment. At any point of execution of a program, it is possible for the processor to reference subcells of different active record contours. For example, the ip in Figure 4 points to an instruction immediately enclosed by B3, the processor must be able to access subcells Z in B3', B3 and Z in B2', and B2 plus P in B1'. One way to solve this problem is to collect the addresses of these reference record contours in the processor's display registers. In the contour model, the value of processor height (P.ih), which is a copy of the height of the instruction being executed, is used to update the processor's display registers (P.dsp) if necessary. If the contents of P.ih is 0, then all display registers are empty. If P.ih contains an integer n+1 with n>0, then (1) for i>n, P.dsp.i is empty, (2) for i=n, the contents of P.dsp.i is a copy of P.soa.ep, and (3) for 0≤i<n, P.dsp.i points to a record contour which immediately encloses the record contour pointed by P.dsp.(i+1). Use Figure 4 as an example, the current contents of P.ih is 3, so P.dsp.2 points to B3', P.dsp.1 points to B2', and P.dsp.0 points to B1'. In this example, 2 is the highest available index of the display registers, and B1' is defined to be the outmost environment while B3' is the innermost (top) environment of the current active processor.

The access environments of the processor consist of a set of record contours pointed to by active display registers. In other words, if ep points to the record contour B3', then the environments are composed of B3' and all those record contours enclosing B3'. The access environments make a processor access subcells of contours by name possible. One thing must be pointed out here, if two or more record contours have a subcell with the same name, then the subcell to be accessed is the one in the innermost record contour.

Antecedent Link and Static Link. The algorithm contour is invariant during the life of a program, but no record contour is invariant. When a record contour P' is allocated, which is a copy of the algorithm contour P, the antecedent link of P' must point to P. P is defined to be the antecedent of P', and P' is a decedent of P. A record contour has a unique antecedent, but an algorithm contour may have several decedents (10). The antecedent links of record contours are used for run-time validity checks.

A contour has a zero height if and only if it has a null static link (10). If a contour, A, has a positive height i+1, then the static link of A must point to a contour, B, having a height equal to i. B is said to immediately enclose A, and A is said to be immediately enclosed or nested within B (10). By using both of the height and the static links if recird contours, the processor's display registers can be built correctly.

Label and Flow Control. Labels are the basic type of data item used in flow of control mechanism. A valid label consists of two pointers: an environment pointer and an instruction pointer. In order to make a new label, a label register as part of a processor is necessary and is denoted by P.lab. A label register is used to keep the return address temporarily when a procedure call instruction is being executed.

ENTER and EXIT. ENTER and EXIT are two important instructions for block structured processes. In the contour model, the effect of executing of an ENTER instruction consists of three steps: copying the algorithm contour into which the processor is entering; adjoining this new record contour to the top of the currently active processor's environments; and updating the processor to an an instruction immediately enclosed by the algorithm contour being entered. The effect of executing an EXIT instruction includes removing the innermost record contour from the active processor's environments and updating the processor to an instruction immediately outside the algorithm contour being exited.

Simplification and Abbreviation. Since the algorithm is static during the runtime of a program, it needs to be shown only once and remains invariant for all remaining snapshots (10). The processor is abbreviated to be π, of which its two legs represent the two pointers, ep and ip, of the processor respectively. Also, it is common for ep to point only the innermost contour of its environments (10).

Built-in Functions and AHPL. In order to describe the contour model concisely and to demonstrate how the contour model handles a program, some built-in functions together with A Hardware Programming Language (AHPL) are used. The definition of each built-in function followed by some examples is described in TABLE I.

The following three examples are used to show how the contour model handles the problems of passing procedure names as actual parameters to the called procedure, parallelism, and recursion respectively.

TABLE I

BUILT-IN FUNCTIONS

BUILT-IN FUNCTION	DEFINITION	EXAMPLE	COMMENT
ADDR(name of contour)	returns the address of the contour in the parentheses	ADDR(P')	similar to ADDR in PL/1
SUB(name of contour(name of subcell))	returns the contents of the subcell in the 2-nd parentheses of the contour in the first parentheses	SUB(P'(L)).ep	return the contents of the ep of the subcell L in record contour P'
		SUB(C(Z)).ip	return the contents of the ip of the subcell Z in algorithm contour C
ANTE(name of contour)	find the antecedent of the record contour in the parentheses	ANTE(P')	return the address of the algorithm contour from which P' is copied
STAT(name of contour)	returns the address of the contour which immediately encloses the argument	STAT(C')	return the address of the record contour which immediately encloses C'
INC(name of register,increment)	increases the contents of the first argument by the second argument	INC(MAR,2)	increase the contents of memory address register by 2

Example 1. In the first example, procedure names are passed as actual parameters to the called procedure. The program to be executed is EXAMPLE_1, as shown in Figure 5.a. EX_1 is the program re-expressed in terms of the contour model, as shown in Figure 5.b. For convenience, the instructions embedded in the algorithm contours are numbered from top to bottom. The number on the right side of each instruction represents the address of that instruction.

As mentioned earlier, a process is a sequence of snapshots, each snapshot results from the preceding one by executing a single instruction of that algorithm.

In snapshot 0, the contour model is prepared to execute EX_1. That is, the processor is initialized to be:

P.soa.ep $\leftarrow$ null;

P.soa.ip $\leftarrow$ 1.

While the rest of subfields of the processor are left uninitialized.

In snapshot 1, instruction 1 is to be executed. Instruction 1 is an ENTER instruction. When an ENTER instruction is being executed, a service routine GETAREA (3) is called to allocate storage locations for record contour M' in this case, and return the address of the first memory location of record contour M' to the processor's pointer register, P.ptr. The AHPL description of snapshot 1 is:

P.ih $\leftarrow$ 0;

P.ptr $\leftarrow$ ADDR(M');

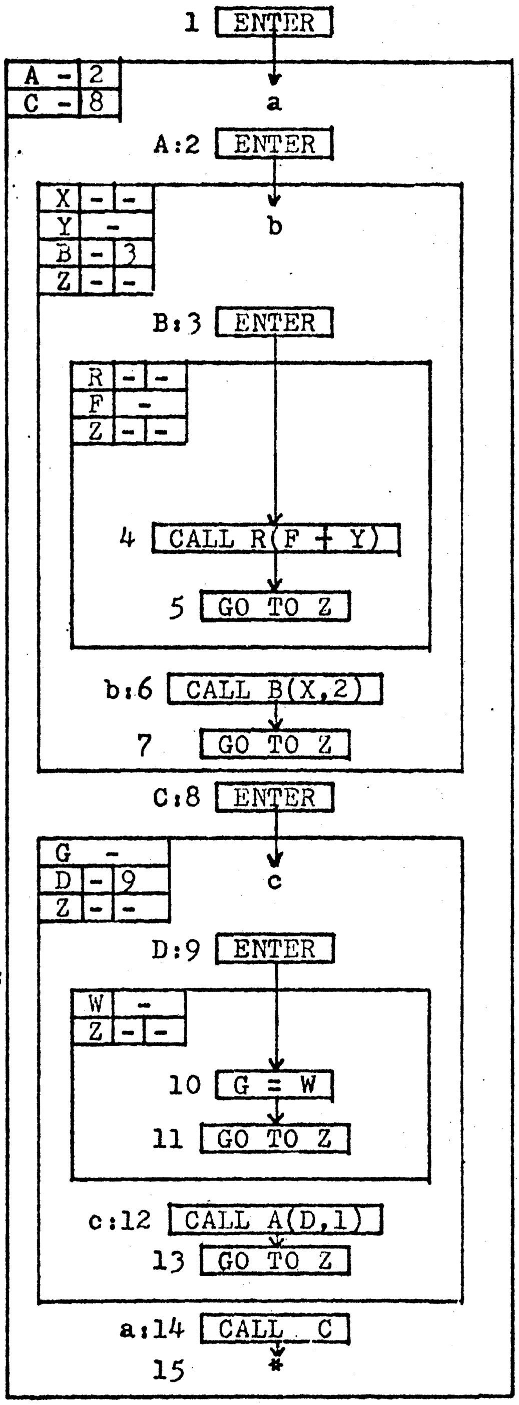
P.dsp.0 $\leftarrow$ P.ptr;

P.soa.ep $\leftarrow$ P.ptr;

P.soa.ip $\leftarrow$ 14.

```
M
BEGIN

 A
 PROCEDURE A(X,Y);
 PROCEDURE X; INTEGER Y;
 BEGIN
  B
  PROCEDURE B(R,F);
  PROCEDURE R; INTEGER F;
  BEGIN

       R(F + Y);

  END;

  B(X,2);
 END;
 C
 PROCEDURE C;

 BEGIN INTEGER G;

  D
  PROCEDURE D(W); INTEGER W;
  BEGIN

       G = W;

  END;

  A(D,1);
 END;

C;
END;
```

a. EXAMPLE_1



b. EX_1

Figure 5. Example 1

In the contour model, the environment pointer of any label residing in an algorithm contour is not initialized completely at assembly time. That is, any algorithm contour contains only incomplete (invalid) labels. As shown in Figure 5.b, the environment pointers of A and C are not initialized. So, to initialize the environment pointers of all labels of a newly allocated record contour is necessary. The way to convert those incomplete labels into valid labels is to let their environment pointer point to the newly allocated record contour if its height is equal to 0 or it is associated with a block. As shown in SNAPSHOT_1 of Figure 6, the environment pointer of A or C points to M'.

In snapshot 2, instruction 14 is to be executed. Instruction 14 is a procedure call instruction with no parameters. The subcell C can be accessed through P.dsp.0, and its value is stored into the label register (P.lab). Since P.soa.ip is updated to point to the instruction to be executed next under normal conditions (the details see Chapter IV), P.soa.ip points to instruction 15. However, instruction 14 is a jump to a subroutine instruction, so the processor exchanges the site of activity register P.soa with the label register P.lab before the next instruction to be executed. After this exchange, P.soa.ip points to instruction 8 instead of 15. The AHPL description of this snapshot is:

```
P.ih ←—— 1;
P.soa.ip ←—— 15;
P.lab.ep ←—— SUB(M'(C)).ep;
P.lab.ip ←—— 8;
TEMP ←—— P.soa.ep;
P.soa.ep ←—— P.lab.ep;
P.lab.ep ←—— TEMP;
```

TEMP $\leftarrow$ P.soa.ip;

P.soa.ip $\leftarrow$ P.lab.ip;

P.lab.ip $\leftarrow$ TEMP.

Where, TEMP may be a general purpose register. At the end of the execution on instruction 14, P.lab contains the return address from procedure C.

In snapshot 3, instruction 8 is to be executed. Instruction 8 is an ENTER instruction. Record contour C' is allocated and its first address is stored into P.ptr. During the allocation, the environment pointer of D points to C', and the return label Z is a copy of P.lab. The AHPL description of snapshot 3 is:

P.ptr $\leftarrow$ ADDR(C');

SUB(C'(D)).ep $\leftarrow$ P.ptr;

SUB(C'(Z)).ep $\leftarrow$ P.lab.ep;

SUB(C'(Z)).ip $\leftarrow$ P.lab.ip;

P.soa.ep $\leftarrow$ P.ptr;

P.soa.ip $\leftarrow$ 12.

In snapshot 4, instruction 12 is to be executed. Since instruction 12 has its height equal to 2, record contour C' becomes the top environment. Instruction 12 is a procedure call on A with parameters D and 1. Procedure A and D can be accessed through P.dsp.0 and P.dsp.1 respectively, and the contents of A is copied into P.lab. Then exchange P.soa with P.lab. The AHPL description of snapshot 4 is:

P.ih $\leftarrow$ 2;

P.dsp.1 $\leftarrow$ ADDR(C');

P.soa.ip $\leftarrow$ 13;

P.lab.ep $\leftarrow$ SUB(M'(A)).ep (or ADDR(M'));

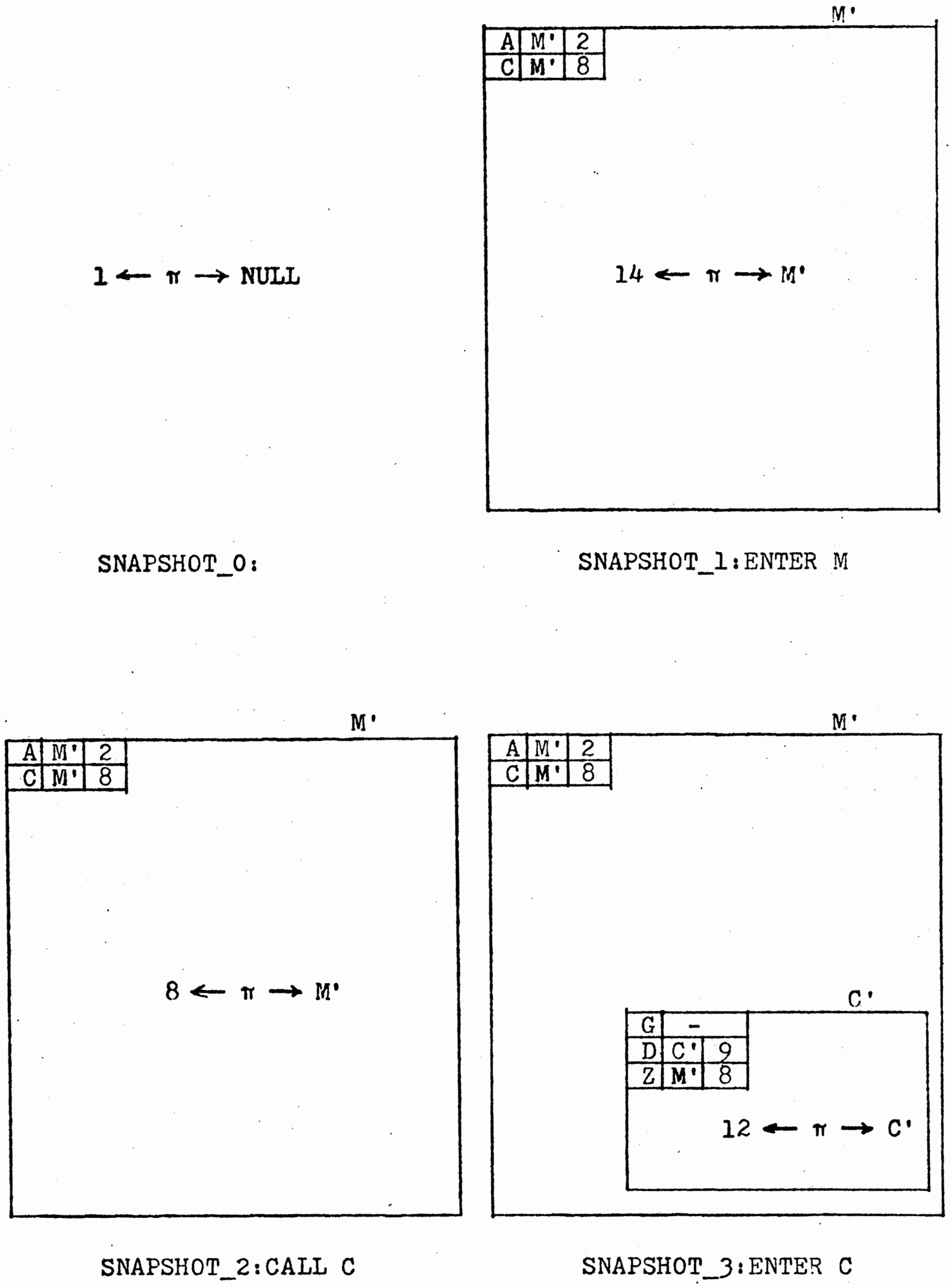


Figure 6. SNAPSHOT_0 to SNAPSHOT_3 of Example 1

P.lab.ip $\leftarrow$ SUB(M'(A)).ip (or 2);

TEMP $\leftarrow$ P.soa.ep;

P.soa.ep $\leftarrow$ P.lab.ep;

P.lab.ep $\leftarrow$ TEMP;

TEMP $\leftarrow$ P.soa.ip;

P.soa.ip $\leftarrow$ P.lab.ip;

P.lab.ip $\leftarrow$ TEMP.

In snapshot 5, instruction 2 is to be executed. The height of instruction 2 is 1, record contour M' becomes the top environment again. Instruction 2 is entering contour A, a copy of algorithm contour A is made with subcells X, Y, B, and Z. The formal parameter X corresponds to actual parameter D, Y is an integer, B is an internal sub-procedure to procedure A, and Z is the return label. After the allocation of record contour A', X is a copy of D, Y is equal to 1, the ep of B points to A', and Z is a copy of P.lab. The AHPL description of this snapshot is:

P.ih $\leftarrow$ 1;

P.dsp.1 $\leftarrow$ null;

P.ptr $\leftarrow$ ADDR(A');

P.soa.ip $\leftarrow$ 6;

P.soa.ep $\leftarrow$ P.ptr;

SUB(A'(X)).ep $\leftarrow$ SUB(C'(D)).ep;

SUB(A'(X)).ip $\leftarrow$ SUB(C'(D)).ip;

SUB(A'(Y)) $\leftarrow$ 1;

SUB(A'(B)).ep $\leftarrow$ P.ptr (or ADDR(A'));

SUB(A'(Z)).ep $\leftarrow$ P.lab.ep;

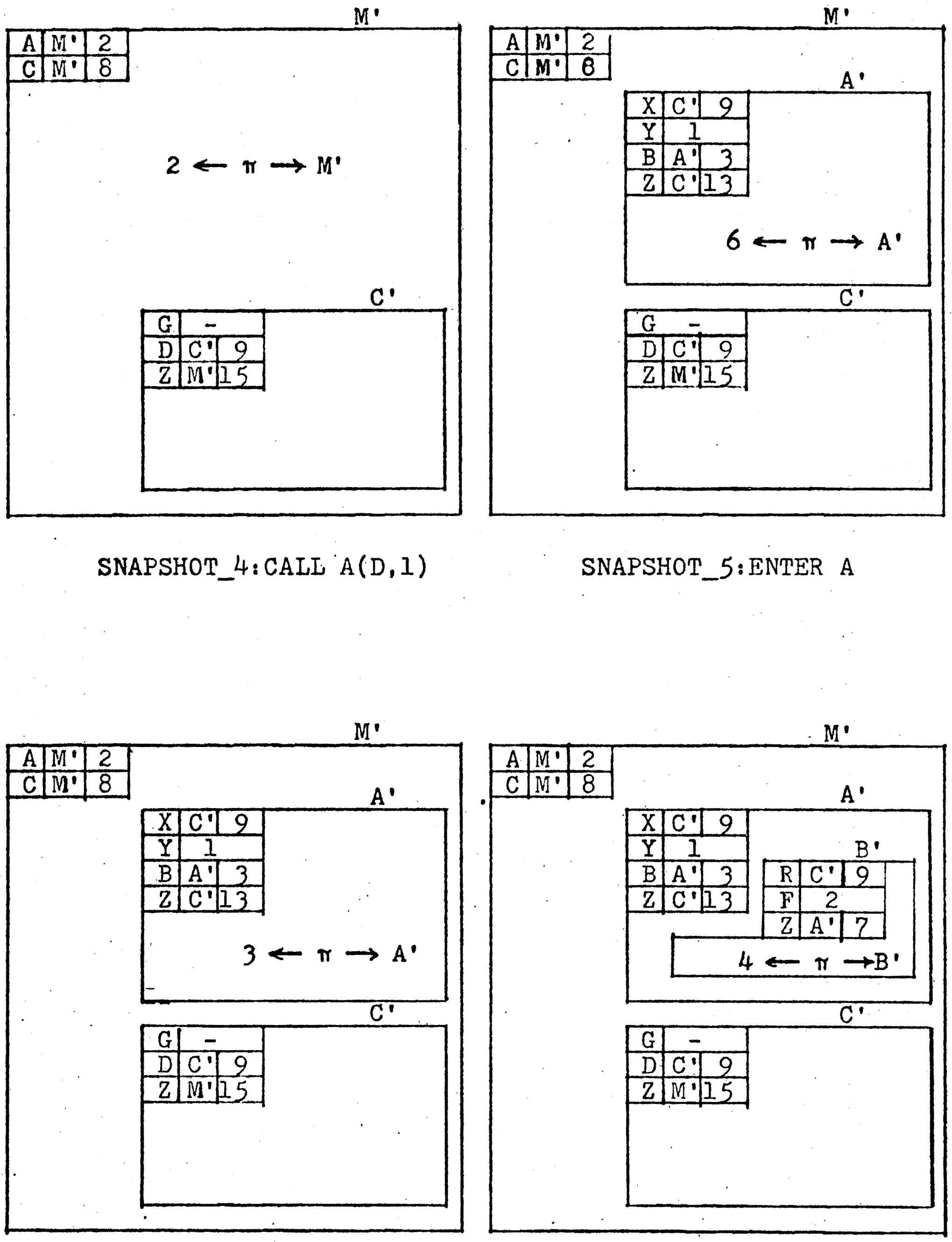
SUB(A'(Z)).ip $\leftarrow$ P.lab.ip.

In snapshot 6, instruction 6 is to be executed. The height of instruction 6 is 2, so record contour A' becomes the top environment: that is, P.dsp.1 points to A'. Instruction 6 is a procedure call on B with parameters X and 2. Subcells B an X can be accessed through P.dsp.1 and P.dsp.0 respectively. Again, the return address is stored into P.lab before the transfer occurs. The AHPL description of this snapshot is:

```
P.ih ←—— 2;
P.dsp.1 ←—— P.ptr  (or ADDR(A'));
P.soa.ip ←—— 7;
P.lab.ep ←—— SUB(A'(B)).ep  (or ADDR(A'));
P.lab.ip ←—— SUB(A'(B)).ip  (or 3);
TEMP ←—— P.spa.ep;
P.soa.ep ←—— P.lab.ep;
P.lab.ep ←—— TEMP;
TEMP ←—— P.soa.ip;
P.lab.ip ←—— P.lab.ip;
P.lab.ip ←—— TEMP.
```

In snapshot 7, instruction 3 is to be executed. Instruction 3 is an ENTER instruction into contour B, so a record contour B' is allocated immediately inside record contour A': that is, the static link of B' points to A'. The AHPL description is :

```
P.ptr ←—— ADDR(B');
P.soa.ip ←—— 4;
P.soa.ep ←—— P.ptr;
SUB(B'(R)).ep ←—— SUB(A'(X)).ep;
SUB(B'(R)).ip ←—— SUB(A'(X)).ip;
```



SNAPSHOT_4:CALL A(D,1) SNAPSHOT_5:ENTER A

SNAPSHOT_6:CALL B(X) SNAPSHOT_7:ENTER B

Figure 7. SNAPSHOT_4 to SNAPSHOT_7 of Example 1

SUB(B'(F)) $\longleftarrow$ 2;

SUB(B'(Z)).ep $\longleftarrow$ P.lab.ep;

SUB(B'(Z)).ip $\longleftarrow$ P.lab.ip.

In snapshot 8, instruction 4 is to be executed. Instruction 4 is a procedure call on R with parameter F+Y. Since the height of instruction 4 is 3, P.dsp must be initialized to point to B', through which R and F can be accessed, while Y can be accessed through P.dsp.1. The AHPL description of this snapshot is:

P.ih $\longleftarrow$ 3;

P.dsp.2 $\longleftarrow$ P.ptr;

P.soa.ip $\longleftarrow$ 5;

P.lab.ep $\longleftarrow$ SUB(B'(R)).ep (or ADDR(C'));

P.lab.ip $\longleftarrow$ SUB(B'(R)).ip (or 9);

TEMP $\longleftarrow$ P.soa.ep;

P.soa.ep $\longleftarrow$ P.lab.ep;

P.lab.ep $\longleftarrow$ TEMP;

TEMP $\longleftarrow$ P.soa.ip;

P.soa.ip $\longleftarrow$ P.lab.ip;

P.lab.ip $\longleftarrow$ TEMP.

In snapshot 9, instruction 9 is to be executed. Instruction 9 has its height equal to 2, and it is immediately enclosed by algorithm contour C, so the display registers must be reconstructed through the static link of record contour C': that is, P.dsp.0 points to M', and P.dsp.1 points to C'. Instruction 9 is an ENTER instruction into D, so a copy of D is made with subcells W and Z. The corresponding AHPL description is:

P.ih $\longleftarrow$ 2;

P.dsp.1 $\longleftarrow$ ADDR(C');

P.dsp.0 $\longleftarrow$ STAT(C');

P.ptr $\longleftarrow$ ADDR(D');

P.soa.ep $\longleftarrow$ P.ptr;

P.soa.ip $\longleftarrow$ 10;

SUB(D'(W)) $\longleftarrow$ F+Y (or 3);

SUB(D'(Z)).ep $\longleftarrow$ P.lab.ep;

SUB(D'(Z)).ip $\longleftarrow$ P.lab.ip.

In snapshot 10, instruction 10 is to be executed. The height of instruction 10 is 3, so P.dsp.2 must be initialized to point to D'. Instruction 10 is to deposit the value of W into G, which resides in record contour C'. The AHPL description of this snapshot is:

P.ih $\longleftarrow$ 3;

P.dsp.2 $\longleftarrow$ P.ptr;

P.soa.ip $\longleftarrow$ 11;

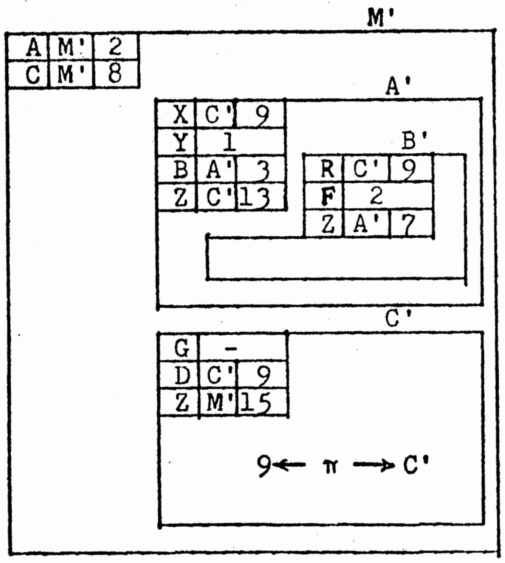
SUB(C'(G)) $\longleftarrow$ SUB(D'(W)).

In snapshot 11, instruction 11 is to be executed. Instruction 11 is a return instruction to the calling procedure, which is specified by the return label Z. After executing this instruction, the contents of Z becomes the new site of activity, and the valid bit of record contour D' is set off: that is, D' is no longer able to be accessed. The AHPL description is:

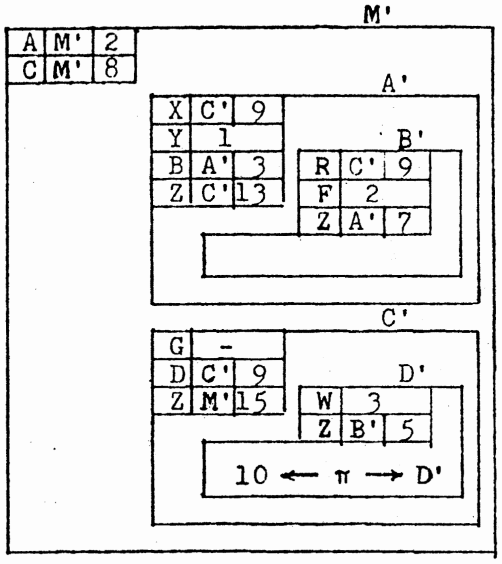
P.soa.ep $\longleftarrow$ SUB(D'(Z)).ep;

P.soa.ip $\longleftarrow$ SUB(D'(Z)).ip.

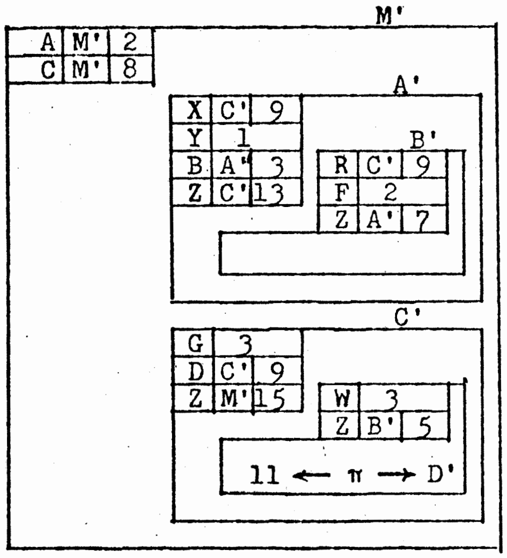
In snapshot 12, instruction 5 is to be executed. Since the last instruction is a jump instruction, the display registers must be reconstrcted by using the static link of B' and the height of instruction 5.



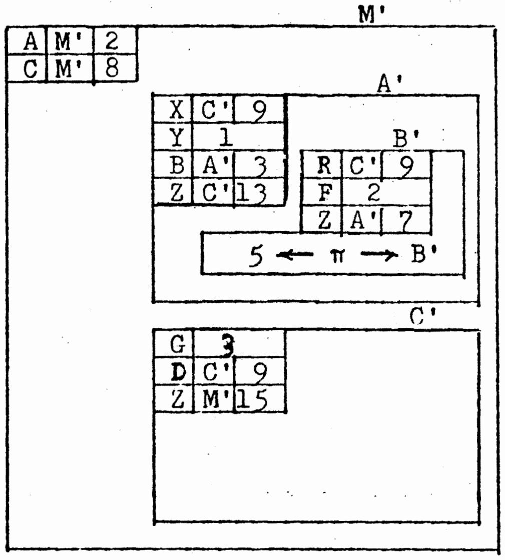
SNAPSHOT_8:CALL R(F+Y)



SNAPSHOT_9:ENTER D



SNAPSHOT_10:G=W



SNAPSHOT 11:EXIT D

Figure 8. SNAPSHOT_8 to SNAPSHOT_11 of Example 1

Instruction 5 is a return instruction to the calling procedure specified by the return label Z. After executing this instruction, the contents of Z, which resides in record contour B', becomes the new site of activity, and record contour B' can no longer able be accessed. The AHPL description of this snapshot is:

P.ih $\leftarrow$ 3;

P.dsp.2 $\leftarrow$ P.soa.ep (or ADDR(B'));

P.dsp.1 $\leftarrow$ STAT(B') (or ADDR(A'));

P.dsp.0 $\leftarrow$ STAT(A') (or ADDR(M'));

P.soa.ep $\leftarrow$ SUB(B'(Z)).ep (or ADDR(A'));

P.soa.ip $\leftarrow$ SUB(B'(Z)).ip (or 7).

In snapshot 13, instruction 7 is to be executed. Instruction 7 is also a return instruction to the calling procedure specified by the return label Z. The same procedure as in snapshot 12 is followed, and the AHPL description is:

P.ih $\leftarrow$ 2;

P.dsp.1 $\leftarrow$ P.soa.ep (or ADDR(A'));

P.dsp.0 $\leftarrow$ STAT(A') (or ADDR(M'));

P.soa.ep $\leftarrow$ SUB(A'(Z)).ep (or ADDR(C'));

P.soa.ip $\leftarrow$ SUB(A'(Z)).ip (or 13).

In snapshot 14, instruction 13 is to be executed. Instruction 13 is another return instruction to the calling procedure specified by the return label Z. Its AHPL description is:

P.ih $\leftarrow$ 2;

P.dsp.1 $\leftarrow$ P.soa.ep (or ADDR(C'));

P.dsp.0 $\leftarrow$ STAT(C') (or ADDR(M'));

P.soa.ep $\leftarrow$ SUB(C'(Z)).ep (or ADDR(M'));

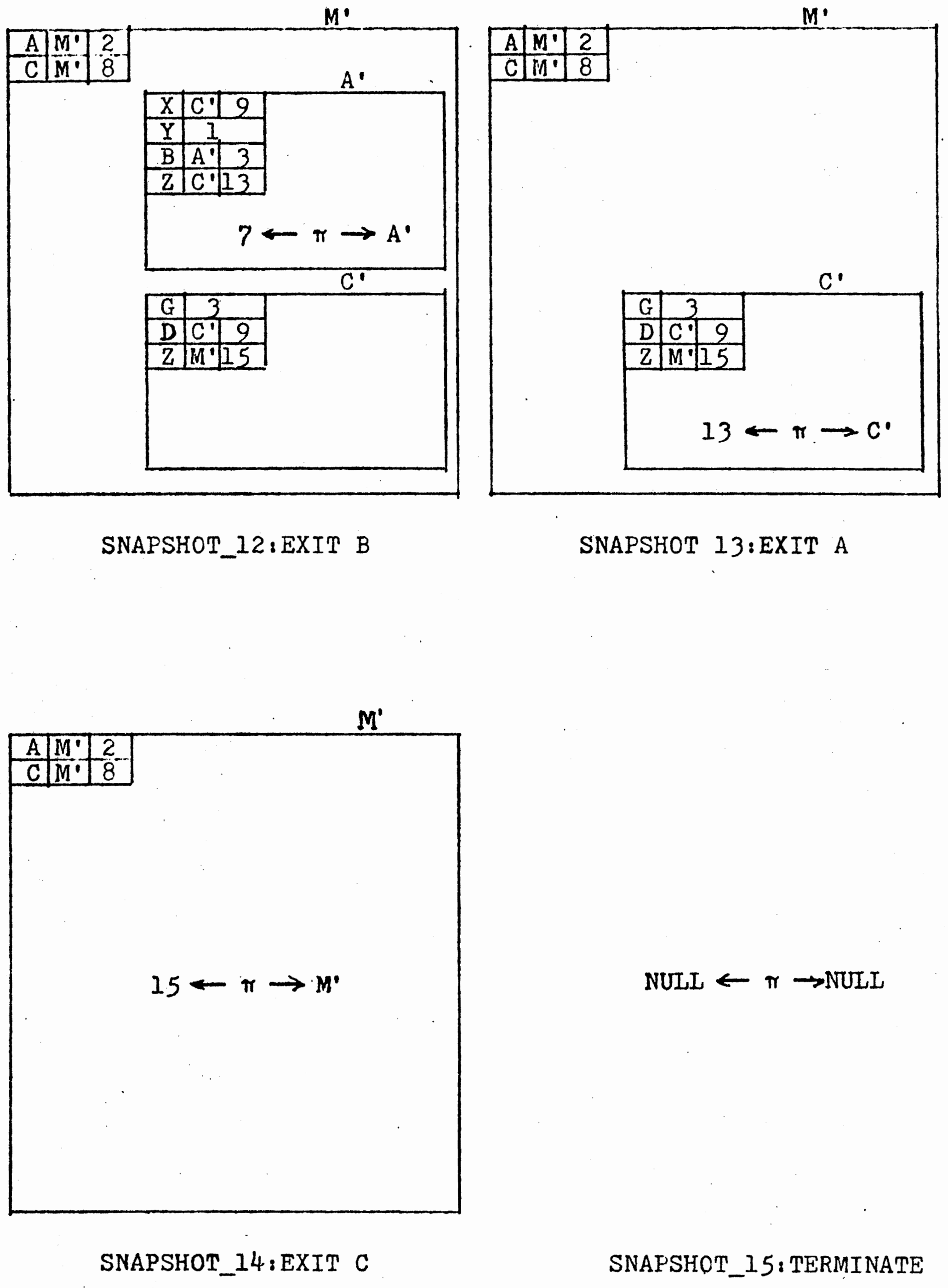


Figure 9. SNAPSHOT_12 to SNAPSHOT_15 of Example 1

P.soa.ip $\longleftarrow$ SUB(C'(Z)).ip (or 15).

In snapshot 15, instruction 15 is to be executed. Instruction 15 is to terminate EX_1, so the executing processor's state will become terminated after the execution of this instruction: that is, its two pointers will point nowhere. Also, record contour M' is no longer able to be accessed. The AHPL description of this snapshot is:

P.ih $\longleftarrow$ 1;

P.soa.ep $\longleftarrow$ null;

P.soa.ip $\longleftarrow$ null;

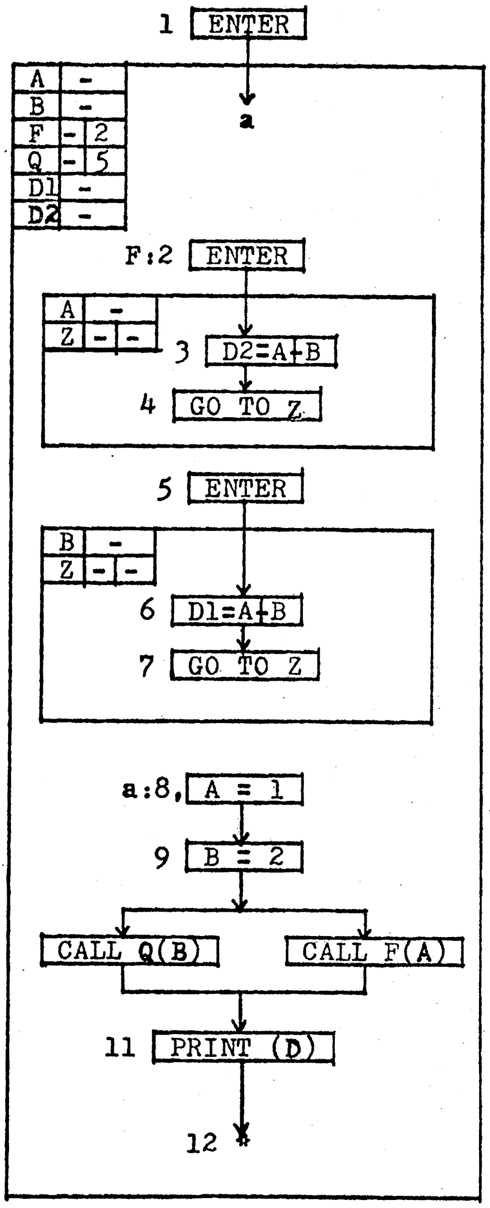
P.dsp.Q $\longleftarrow$ null.

In this example, procedure C is called in snapshot 2 first. Inside C, procedure A is called in snapshot 4, with procedure D as an actual paremater. Inside A, procedure B is called in snapshot 6, with its formal parameter X, a procedure, as the actual parameter. Finally, the formal parameter R is called in snapshot 8. Since B's formal parameter R is X which in tern is D, procedure D is involked. Each time a procedure is called with a procedure as the actual parameter, the contour model can just treat it as a label and pass it to the called procedure, and no extra procedure overhead is required.

Example 2. In this example, we will see how the contour model can execute two procedures in parallel. The program to be executed is EXAMPLE_2, as shown in Figure 10.a. EX_2, as shown in Figure 10.b, is the algorithm re-expressed in terms of the data structure of the contour model. For convenience, the array D declared in the main algorithm contour M is treated as two distinct elements D1 and D2 at this moment, though this is not the true situation.

```
M
BEGIN INTEGER A,B;
      [1:2] INTEGER D;


F
 PROCEDURE F=(REF, INT A)

      INT: A + B;


Q
 PROCEDURE Q=(REF, INT B)


      INT: A + B;



A:=1;

B:=2;

D:=[Q(B), F(A)];

PRINT (D);

END;
```

a. EXAMPLE_2



b. EX_2

Figure 10. Example 2

Again, a sequence of snapshots is implemented to depict the execution of EX_2 by the contour model. However, no literal explanation except the AHPL description is presented in each snapshot unless something new (which does not happen to EX_1) is encountered. Since more than one processor will be used in this example, indexes are used to separate different processors. For example, P_0.soa.ep denotes the environment pointer of the processor with index 0.

The AHPL descriptions of snapshot 0 through snapshot 3 are:

P_0.soa.ep $\longleftarrow$ null (snapshot 0 begins);

P_0.soa.ip $\longleftarrow$ 1 (snapshot 0 ends);

P_0.ptr $\longleftarrow$ ADDR(M') (snapshot 1 begins);

P_0.soa.ep $\longleftarrow$ P_0.ptr;

P_0.soa.ip $\longleftarrow$ 8;

SUB(M'(F)).ep $\longleftarrow$ P_0.ptr (or ADDR(M'));

SUB(M'(F)).ip $\longleftarrow$ SUB(M(F)).ip (or 2);

SUB(M'(Q)).ep $\longleftarrow$ P_0.ptr (or ADDR(M'));

SUB(M'(Q)).ip $\longleftarrow$ SUB(M(Q)).ip (or 5);

P_0.ih $\longleftarrow$ 1 (snapshot 2 begins);

P_0.dsp.0 $\longleftarrow$ P_0.ptr;

P_0.soa.ip $\longleftarrow$ 9;

SUB(M'(A)) $\longleftarrow$ 1;

P_0.soa.ip $\longleftarrow$ 10 (snapshot 3 begins);

SUB(M'(B)) $\longleftarrow$ 2.

In snapshot 4, instruction 10 is to be executed. In instruction 10, two procedures are called simultaneously, so two new processors P_1 and P_2 are allocated and initialized by P_0 (for details see CHAPTER III and IV). F, Q, A, and B can be accessed through P_0.dsp.0 and the offset

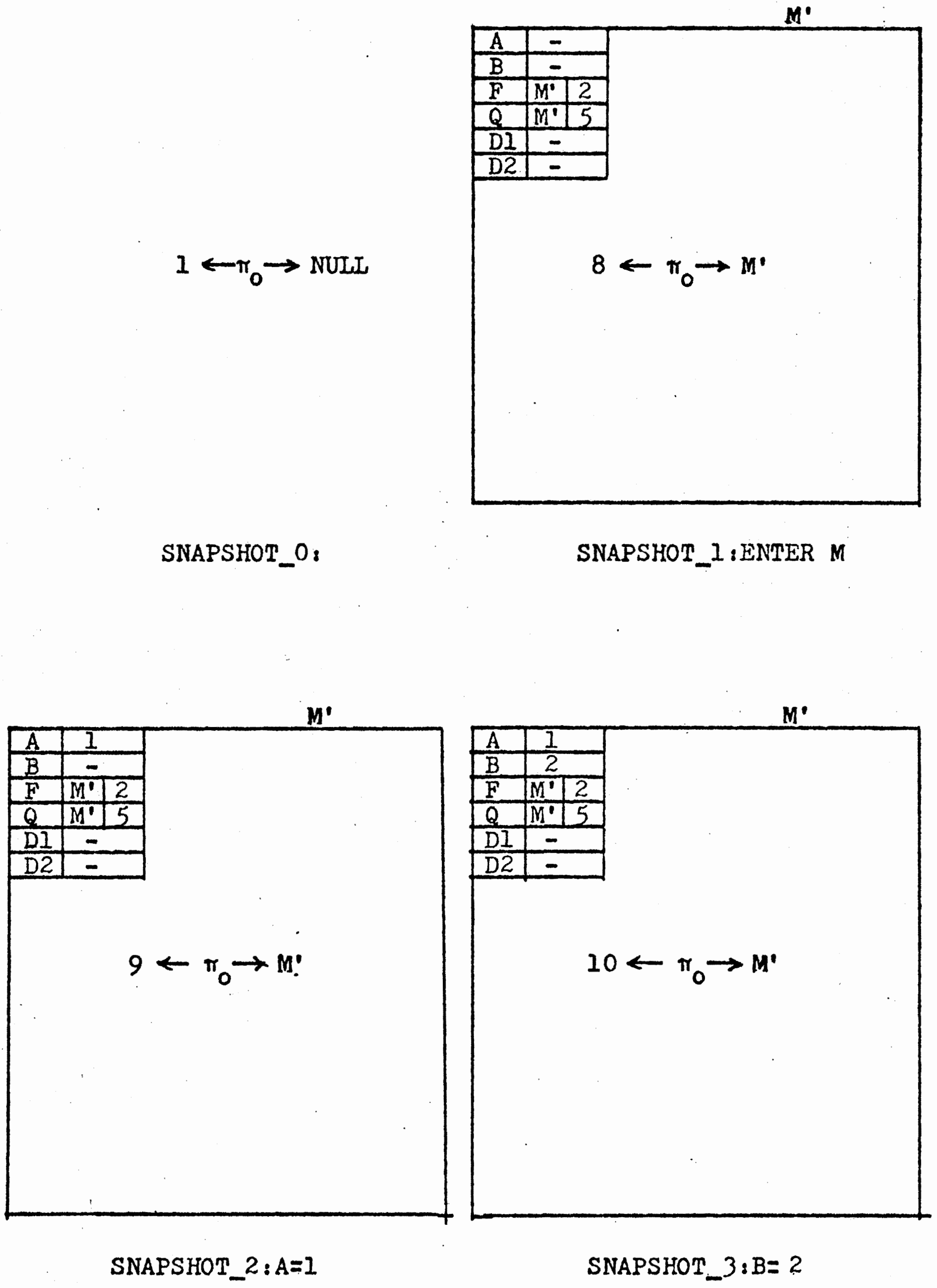


Figure 11. SNAPSHOT_0 to SNAPSHOT_3 of Example 2

of each corresponding subcell in M'. The AHPL description of snapshot 4 is:

P_0.soa.ip $\leftarrow$ 11;

P_1.soa.ep $\leftarrow$ SUB(M'(F)).ep;

P_1.soa.ip $\leftarrow$ SUB(M'(F)).ip;

P_1.lab.ep $\leftarrow$ P_0.soa.ep;

P_1.lab.ip $\leftarrow$ P_0.soa.ip;

P_2.soa.ep $\leftarrow$ SUB(M'(Q)).ep;

P_2.soa.ip $\leftarrow$ SUB(M'(Q)).ip;

P_2.lab.ep $\leftarrow$ P_0.soa.ep;

P_2.lab.ip $\leftarrow$ P_0.soa.ip.

In snapshot 5, two instructions 2 and 5 are executed parallely by processors P_1 and P_2 respectively. Since both instructions 2 and 5 are ENTER instructions, two record contours F' and Q' are allocated and initialized. The AHPL description of processor P_1 executing instruction 2 is:

P_1.ptr $\leftarrow$ ADDR(F');

P_1.soa.ep $\leftarrow$ P_1.ptr;

P_1.soa.ip $\leftarrow$ 3;

P_1.ih $\leftarrow$ 1;

P_1.dsp.0 $\leftarrow$ ADDR(M');

SUB(F'(A)) $\leftarrow$ SUB(M'(A));

SUB(F'(Z)).ep $\leftarrow$ P_1.lab.ep;

SUB(F'(Z)).ip $\leftarrow$ P_1.lab.ip.

The AHPL description of P_2 executing instruction 5 is:

P_2.ptr $\leftarrow$ ADDR(Q');

P_2.soa.ep $\leftarrow$ P_2.ptr;

P_2.soa.ip $\longleftarrow$ 6;

P_2.ih $\longleftarrow$ 1;

P_2.dsp.0 $\longleftarrow$ ADDR(M');

SUB(Q'(B)) $\longleftarrow$ SUB(M'(B));

SUB(Q'(Z)).ep $\longleftarrow$ P_2.lab.ep;

SUB(Q'(Z)).ip $\longleftarrow$ P_2.lab.ip.

In snapshot 6, instruction 3 and 6 are to be executed by P_1 and P_2 respectively. Since the heights of both instructions 3 and 6 are 2, so P_1.dsp.1 and P_2.dsp.1 must be initialized to point to F' and Q' separately. Both instructions 3 and 6 are to find the sum of A and B. For P_1, A is a local variable, while B is a global variable. For P_2, B is a local variable, while A is a global variable. Global variables B and A can be accessed through P_1.dsp.0 and P_2.dsp.0 respectively. Finally, deposit the sum of A and B into D1 and D2. The AHPL description of P_1 executing instruction 3 is:

P_1.ih $\longleftarrow$ 2;

P_1.dsp.1 $\longleftarrow$ P_1.ptr;

P_1.soa.ip $\longleftarrow$ 4;

SUB(M'(D1)) $\longleftarrow$ SUB(M'(B)) + SUB(F'(A)).

The AHPL description of P_2 executing instruction 6 is:

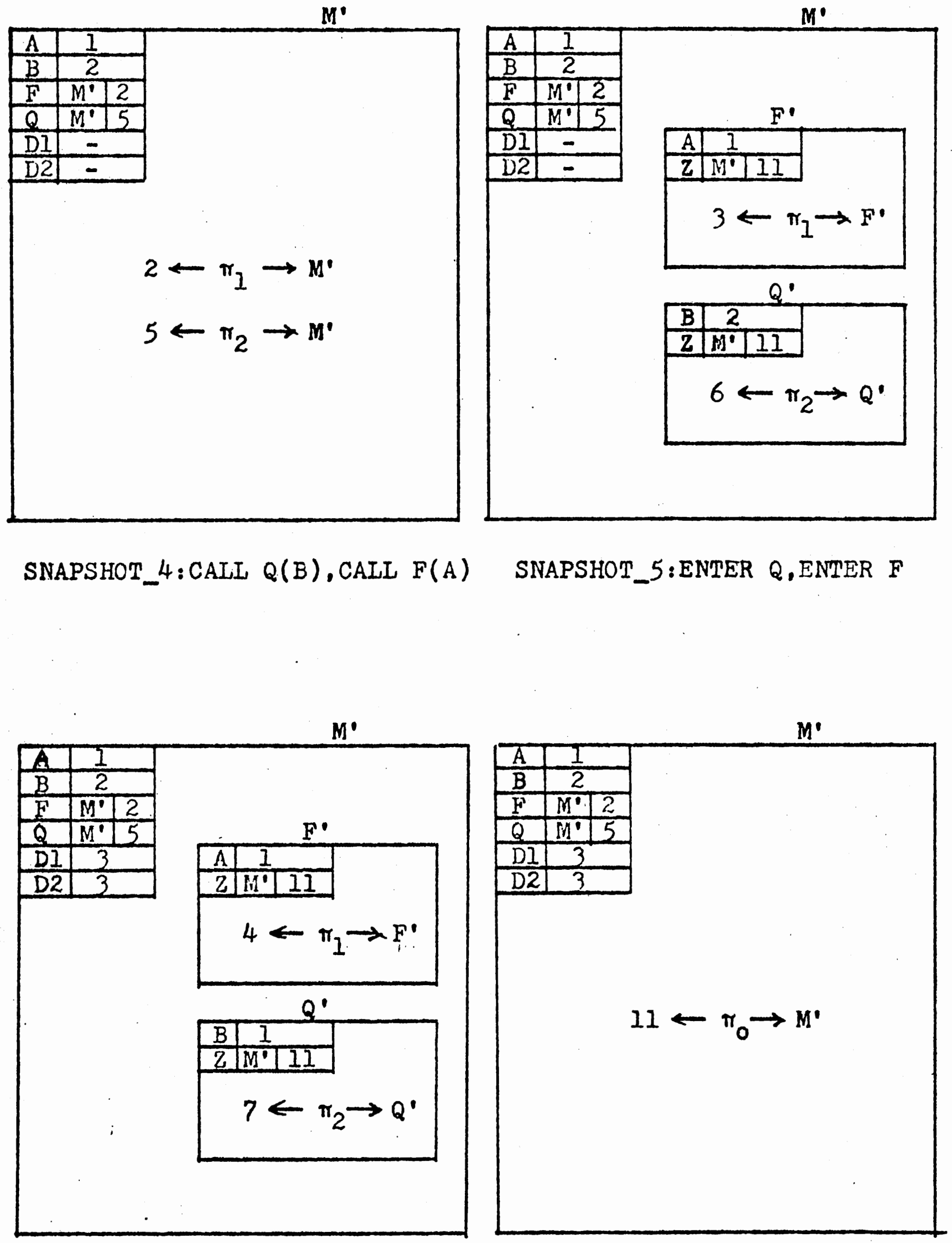
P_2.ih $\longleftarrow$ 2;

P_2.dsp.1 $\longleftarrow$ P_2.ptr;

P_2.soa.ip $\longleftarrow$ 7;

SUB(M'(D2)) $\longleftarrow$ SUB(M'(A)) + SUB(Q'(B)).

In snapshot 7, instruction 4 and instruction 7 are to be exected by P_1 and P_2 separately. Since both instructions 4 and 7 are return instructions to the same procedure: that is, P_1 and P_2 will rejoin to a single


M'
A 1
B 2
F M' 2
Q M' 5
D1 -
D2 -
$2 \leftarrow \pi_1 \rightarrow M'$
$5 \leftarrow \pi_2 \rightarrow M'$
M'
A 1
B 2
F M' 2
Q M' 5
D1 -
D2 -
F'
A 1
Z M' 11
$3 \leftarrow \pi_1 \rightarrow F'$
Q'
B 2
Z M' 11
$6 \leftarrow \pi_2 \rightarrow Q'$
M'
A 1
B 2
F M' 2
Q M' 5
D1 3
D2 3
F'
A 1
Z M' 11
$4 \leftarrow \pi_1 \rightarrow F'$
Q'
B 1
Z M' 11
$7 \leftarrow \pi_2 \rightarrow Q'$
M'
A 1
B 2
F M' 2
Q M' 5
D1 3
D2 3
$11 \leftarrow \pi_0 \rightarrow M'$

SNAPSHOT_4:CALL Q(B),CALL F(A)

SNAPSHOT_5:ENTER Q,ENTER F

SNAPSHOT_6:D_1=A+B,D_2=A+B

SNAPSHOT_7:EXIT F,EXIT Q

Figure 12. SNAPSHOT_4 to SNAPSHOT_7 of Example 2

site of activity. So, P_0 will come back to execute EX_2. Recall when P_0 initializes P_1 and P_2 in snapshot 4, the state of P_0 should become asleep, however, P_0.soa.ip still points to instruction 11, which is the rejoining point after the processes F and Q are finished.

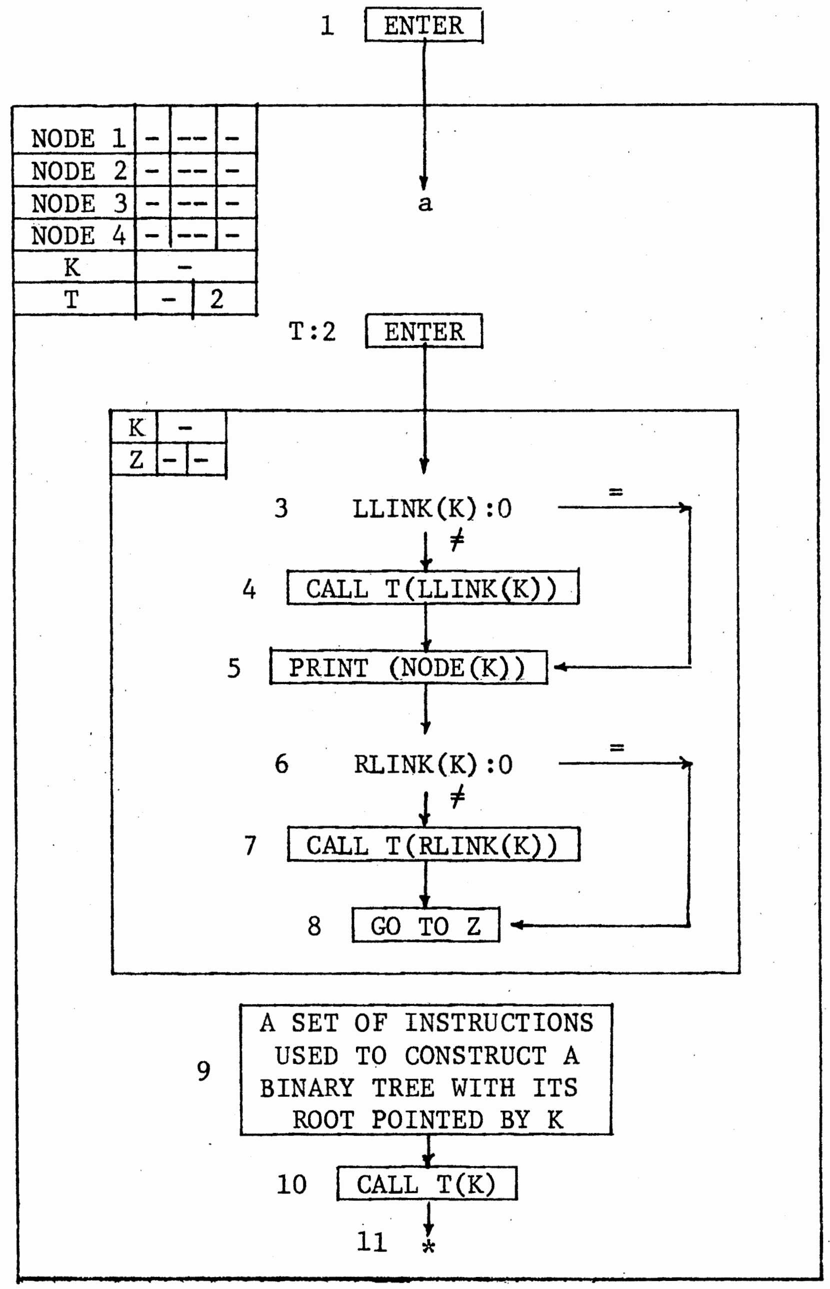
In snapshot 8, instruction 11 is to be executed. Instruction 11 is used to print out the values of array D, so the values of D1 and D2 will be shown on the output list.

In snapshot 9, instruction 12 is to be executed. Instruction 12 terminates EX_2, so P_0.soa.ep and P_0.soa.ip will point to null after execting instruction 12.

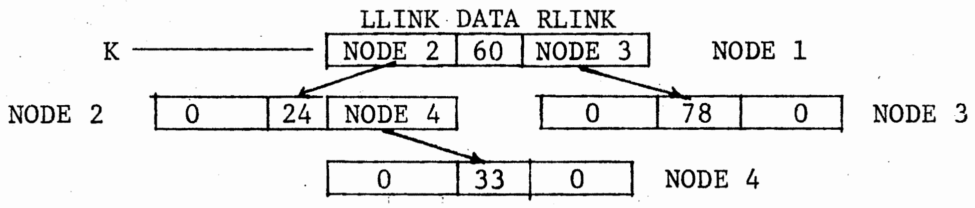
For clarity, three processors P_0, P_1, and P_2 are used in this example. Of course, there are some alternatives. For example, P_0 is used as the "master" processor in the execution of EX_2, so only one processor need be initialized in snapshot 4 to execute F and Q parallely by P_0 and the processor just initialized respectively. After process Q is finished, its executing processor will "commit suicide" and rejoin back to P_0.

Example 3. In this example, the algorithm EX_3 to be executed is the inorder traversal of a binary tree (BT). In Figure 13.a, EX_3 is the algorithm to be executed by the contour model. The binary tree to be traversed is BT, as shown in Figure 13.b. The root of BT is pointed by a pointer K, each node of BT has three subfields: LLINK (left link), RLINK (right link), and DATA. If the LLINK (or RLINK) of a node contains a 0, then that node has no left (or right) child.

An array NODE, declared in the main procedure M, is used to store the nodes of a binary tree. T is the recursive procedure to traverse that binary tree. Again, the array NODE is treated as several (4 in



a). EX_3



b). Binary Tree

Figure 13. Example 3

this example) different variables. Each of which contains LLINK, DATA, and RLINK. In order to put more attention on how the contour model executes EX_3 recursively, we skip discussing instruction 1 and instruction 9. The subcells of record contour M' before the execution of instruction 10 is shown in SNAPSHOT_2 in Figure 14.

In snapshot 3, instruction 10 is to be executed. Procedure T is called with an actual parameter K. Both T and K can be accessed through P.dsp.0, which points to record contour M' currently. Its AHPL description is:

P.soa.ip $\longleftarrow$ 11;

P.lab.ep $\longleftarrow$ SUB(M'(T)).ep;

P.lab.ip $\longleftarrow$ SUB(M'(T)).ip;

TEMP $\longleftarrow$ P.soa.ep;

P.soa.ep $\longleftarrow$ P.lab.ep;

P.lab.ep $\longleftarrow$ TEMP;

TEMP $\longleftarrow$ P.soa.ip;

P.soa.ip $\longleftarrow$ P.lab.ip;

P.lab.ip $\longleftarrow$ TEMP.

In snapshot 4, instruction 2 is to be executed. Instruction 2 is an ENTER instruction into procedure T. The first record contour T' is allocated and initialized. Its AHPL description is:

P.ptr $\longleftarrow$ ADDR(T');

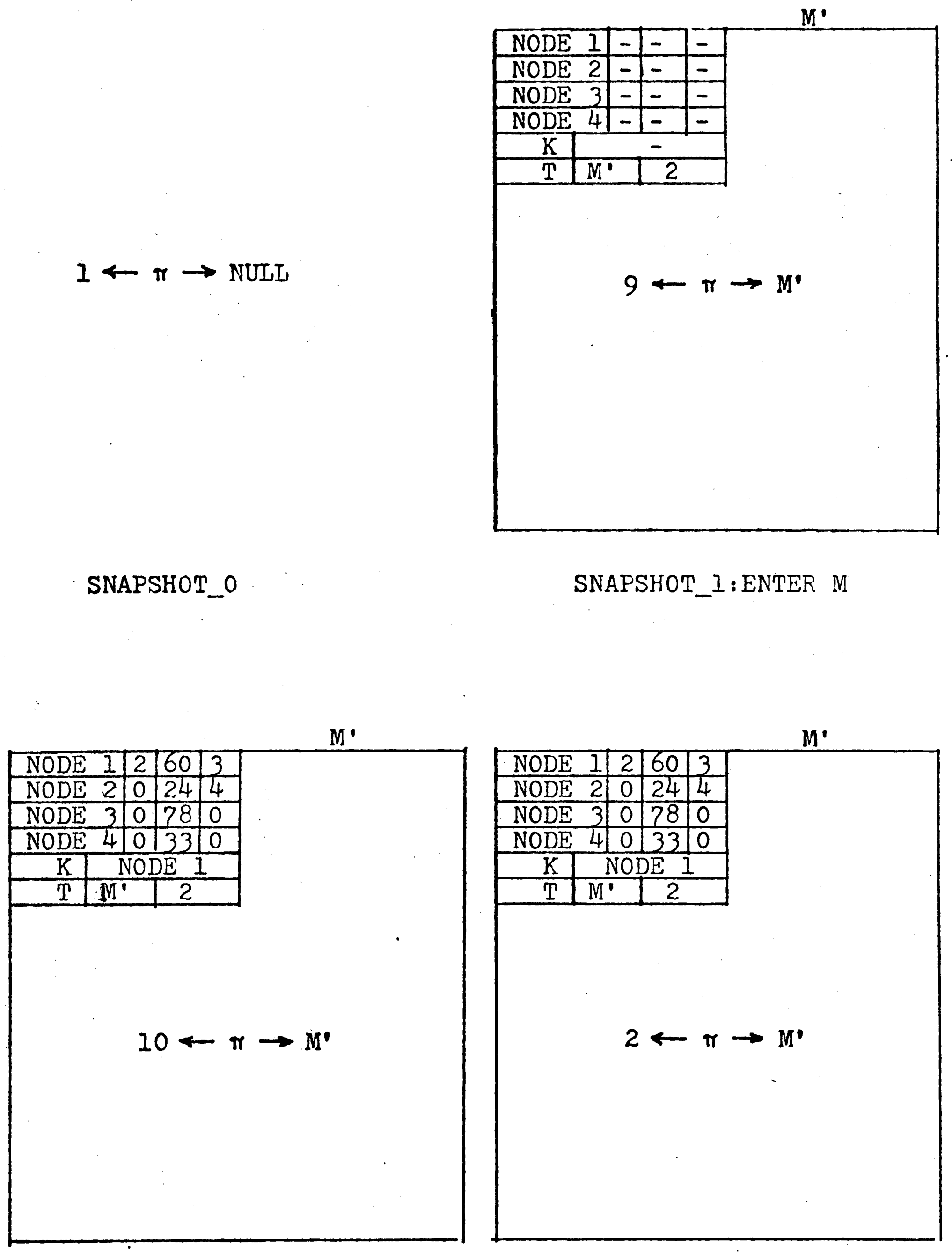
P.soa.ep $\longleftarrow$ P.ptr;

P.soa.ip $\longleftarrow$ 3;

SUB(T'(K)) $\longleftarrow$ SUB(M'(K));

SUB(T'(Z)).ep $\longleftarrow$ P.lab.ep;

SUB(T'(Z)).ip $\longleftarrow$ P.lab.ip.



NODE 1	-	-	-
NODE 2	-	-	-
NODE 3	-	-	-
NODE 4	-	-	-
K	-		
T	M'	2	

NODE 1	2	60	3
NODE 2	0	24	4
NODE 3	0	78	0
NODE 4	0	33	0
K	NODE 1		
T	M'	2	

NODE 1	2	60	3
NODE 2	0	24	4
NODE 3	0	78	0
NODE 4	0	33	0
K	NODE 1		
T	M'	2	

SNAPSHOT_0 SNAPSHOT_1:ENTER M

SNAPSHOT_2:CONSTRUCT BT SNAPSHOT_3:CALL T(NODE_1)

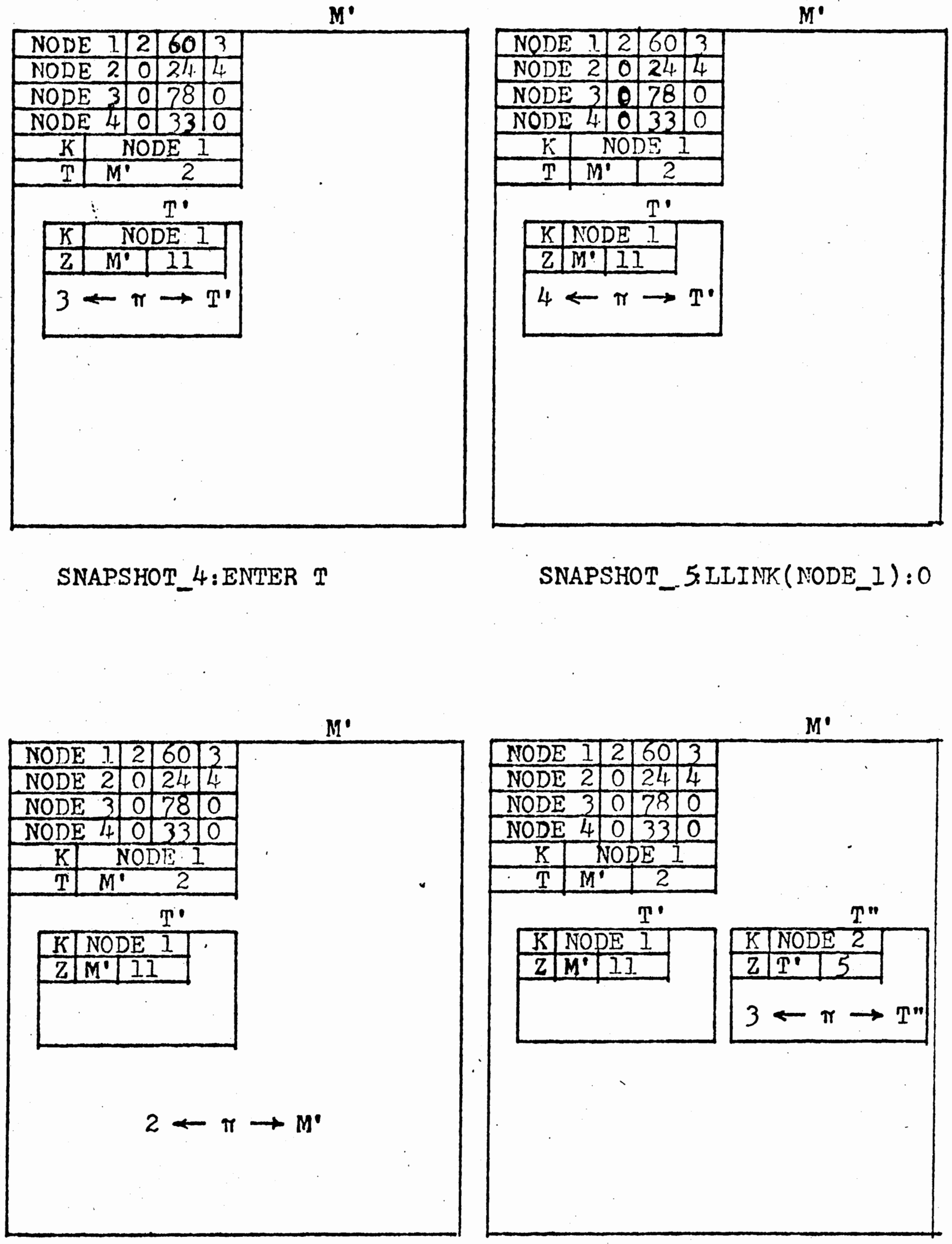
Figure 14. SNAPSHOT_0 to SNAPSHOT_3 of Example 3

In snapshot 5, instruction 3 is to be executed. The height of instruction 3 is 2, P.dsp.1 must be initialized to point to T'. A logical operation testing if the node pointed by K has left child. If it does, then instruction 4 is the instruction to be executed next. If it does not, then instruction 5 will be executed next. Since NODE 2 is the left child of NODE 1, the instruction to be executed is 4. The AHPL description of this snapshot is:

```
P.ih ←—— 2;
P.dsp.1 ←—— P.ptr;
P.soa.ip ←—— 4.
```

In snapshot 6, a call on the procedure T with the actual parameter LLINK(K) is executed. The subcell K to be accessed is in T' instead of M'. After getting the value of K, both T and LLINK(K) can be accessed through P.dsp.0. Instruction 5 is the instruction to be executed next in normal sequence, so both its address and environment must be stored in P.lab at the end of instruction 4. The AHPL description of snapshot 6 is:

```
P.soa.ip ←—— 5;
P.lab.ep ←—— SUB(M'(T)).ep;
P.lab.ip ←—— SUB(M'(T)).ip;
TEMP ←—— P.soa.ep;
P.soa.ep ←—— P.lab.ep;
P.lab.ep ←—— TEMP;
TEMP ←—— P.soa.ip;
P.soa.ip ←—— P.lab.ip;
P.lab.ip ←—— TEMP.
```



SNAPSHOT_4:ENTER T

SNAPSHOT_5:LLINK(NODE_1):0

SNAPSHOT_6:CALL T(LLINK(K))

SNAPSHOT_7:ENTER T

Figure 15. SNAPSHOT_4 to SNAPSHOT_7 of Example 3

In snapshot 7, a second copy of algorithm contour T is made and allocated immediately inside M'. In order to distinguish from the first copy T', the second copy is denoted by T". The AHPL description of this snapshot is:

P.ptr ← ADDR(T");

P.soa.ep ← P.ptr;

P.soa.ip ← 3;

SUB(T"(K)) ← SUB(M'(SUB(T'(K)))).LLINK;

SUB(T"(Z)).ep ← P.lab.ep;

SUB(T"(Z)).ip ← P.lab.ip.

In snapshot 8, the same procedure will be followed as that in snapshot 5. Since NODE 2 has no left child, the instruction to be executed next is instruction 5.

In snapshot 9, the DATA of NODE 2 is printed out, and P.soa.ip points to instruction 6.

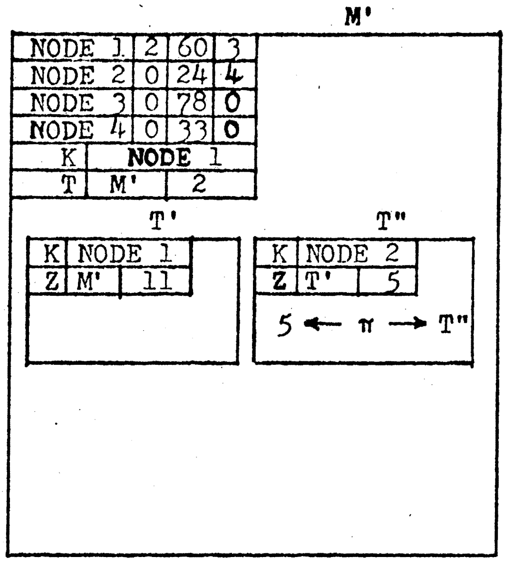
In snapshot 10, a logical operation testing whether NODE 2 has a right child or not. Since NODE 3 is the right child of NODE 2, the instruction to be executed next is instruction 7.

In snapshot 11, procedure T is called with the actual parameter RLINK(K). The subcell K is accessed from T" first, then RLINK(K) can be accessed through P.dsp.0. The AHPL description of this snapshot is exactly the same as that of snapshot 6 except P.soa.ip will point to instruction 8.

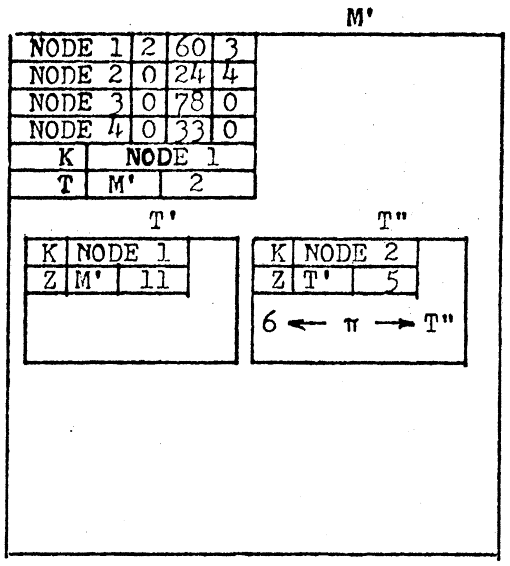
In snapshot 12, a third copy, T'", of algorithm contour T is made. The corresponding AHPL description is:

P.ptr ← ADDR(T'");

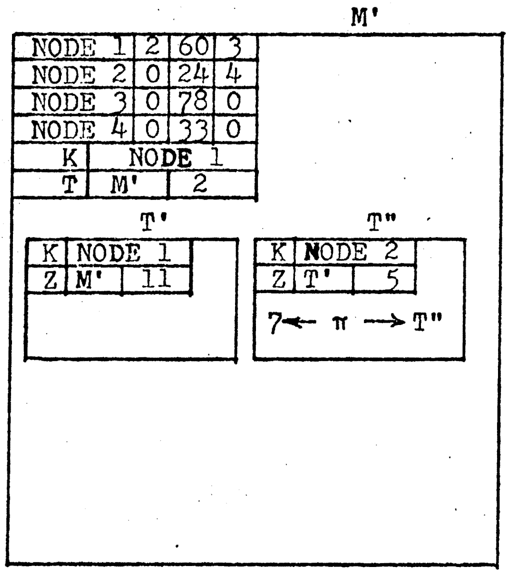
P.soa.ep ← P.ptr;



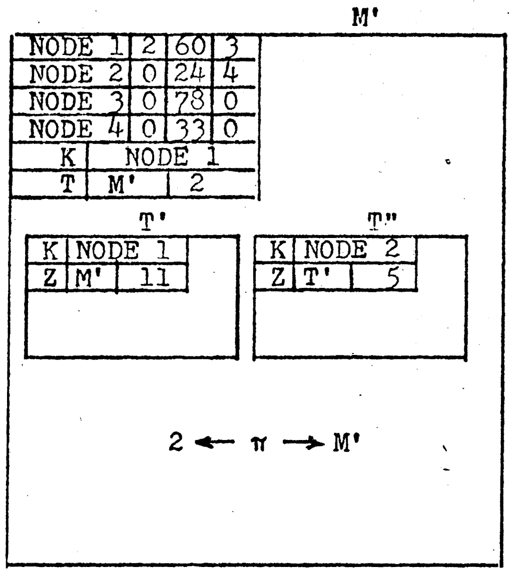
SNAPSHOT_8:LLINK(NODE 2):0



SNAPSHOT_9:PRINT NODE_2



SNAPSHOT_10:RLINK(NODE_2):0



SNAPSHOT_11: CALL T(RLINK(K))

Figure 16. SNAPSHOT_8 to SNAPSHOT_11 of Example 3

P.soa.ip $\longleftarrow$ 3;

SUB(T'''(K)) $\longleftarrow$ SUB(M'(SUB(T''(K)))).RLINK;

SUB(T'''(Z)).ep $\longleftarrow$ P.lab.ep;

SUB(T'''(Z)).ip $\longleftarrow$ P.lab.ip.

In snapshot 13, the same procedure as that of snapshot 8 is followed. Since NODE 4 is a leaf, the next instruction to be executed is instruction 5.

In snapshot 14, the DATA of NODE 4 is printed out, and P.soa.ip will point to instruction 6.

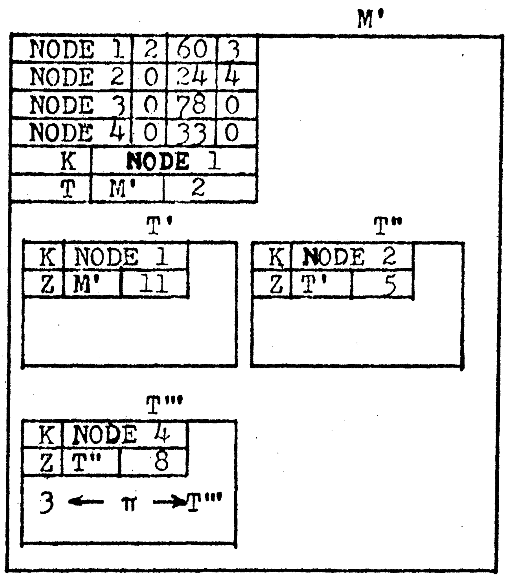
In snapshot 15, the logical test on the value of the right link of NODE 3 is made. Since this test fails, the instruction to be executed is instruction 8.

In snapshot 16, a return instruction with return address specified by the subcell Z of T''' is executed. After executing this instruction, T''' is no longer able to be accessed. The environment pointer of the new site of activity points to record contour T'', and its instruction pointer points to instruction 8.

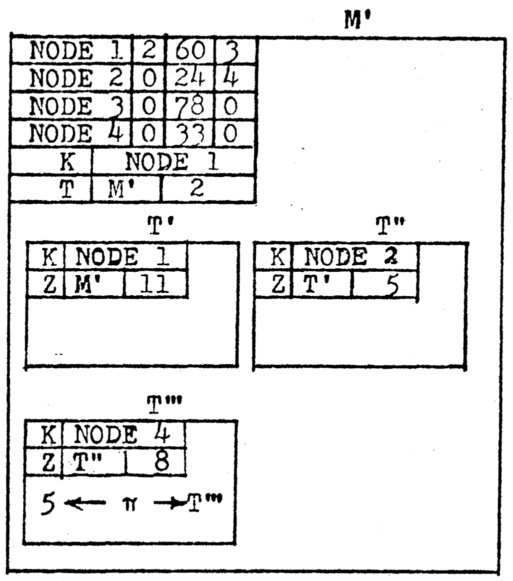
In snapshot 17, the instruction to be executed is still instruction 8. The new site of activity is a copy of the subcell Z of recore contour T''. Also, record contour T'' will be deallocated after this snapshot.

In snapshot 18, the DATA of NODE 1 is printed out, and P.soa.ip will point to instruction 6.

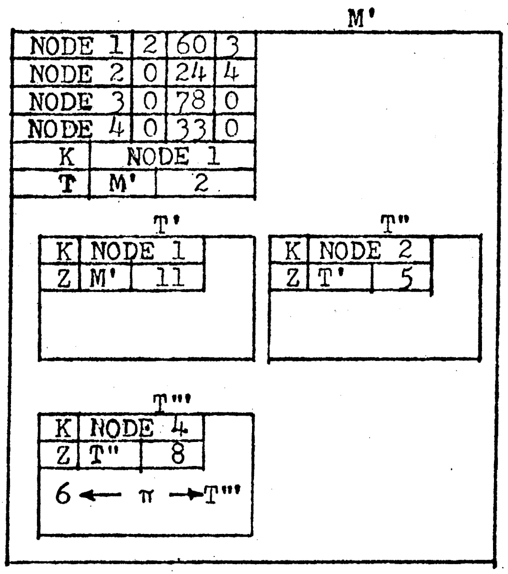
Since NODE 1 has a right child NODE 4, the instruction to be executed is instruction 7 in snapshot 20. Instruction 7 is a call on T with an actual parameter RLINK(K), so a fourth copy. T'''', of algorithm contour T is allocated immediately inside M' in snapshot 21. The steps



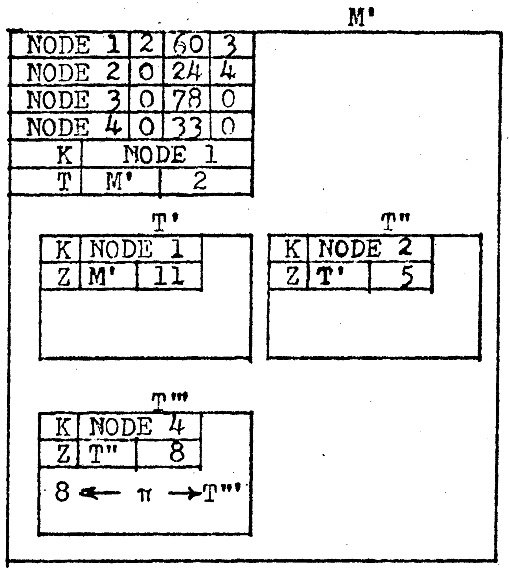
SNAPSHOT_12:ENTER T



SNAPSHOT_13:LLINK(NODE 4):0

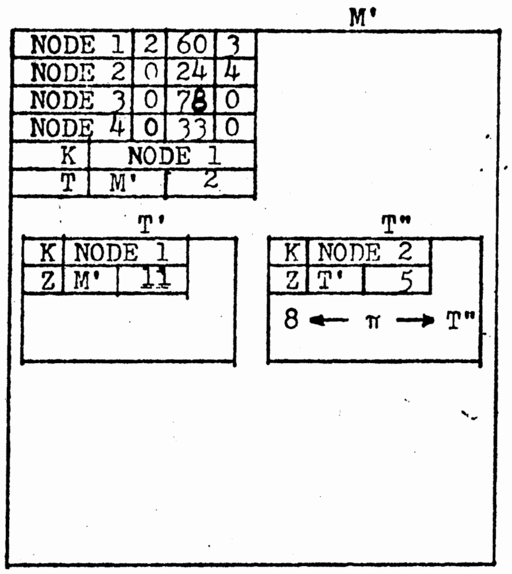


SNAPSHOT_14:PRINT NODE 4

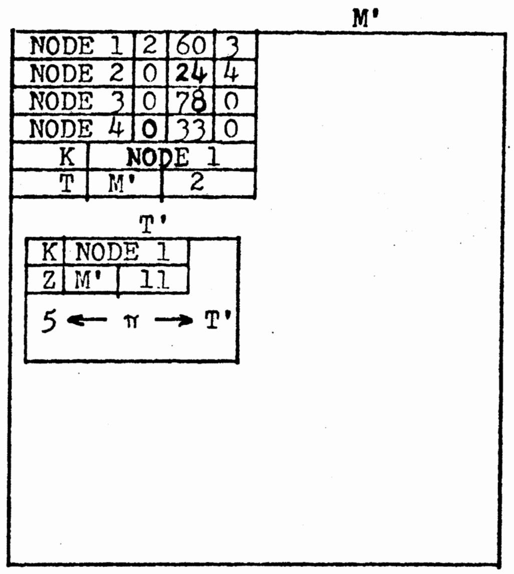


SNAPSHOT_15:RLINK(NODE 4):0

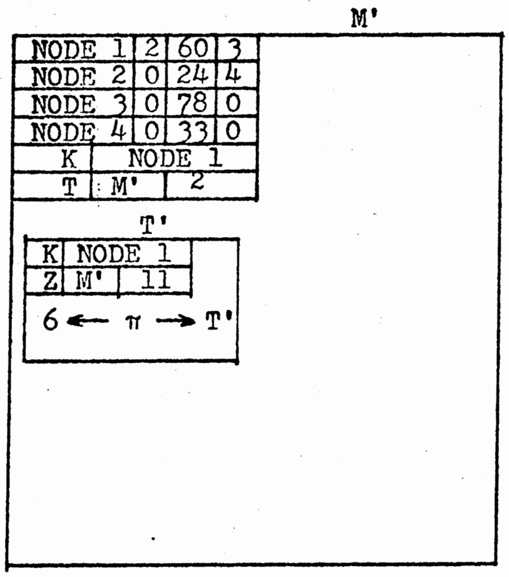
Figure 17. SNAPSHOT_12 to SNAPSHOT_15 of Example 3



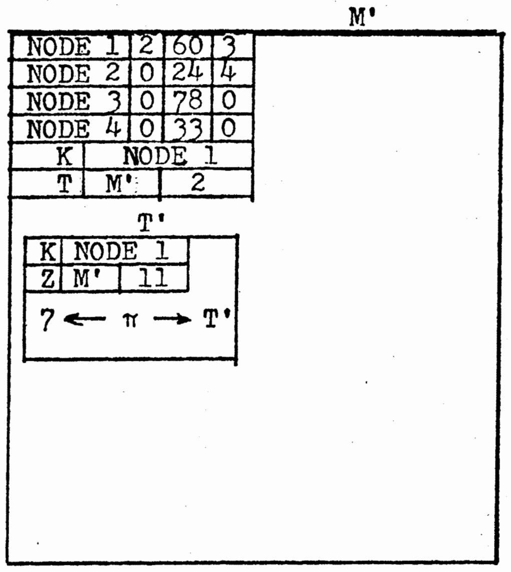
SNAPSHOT_16:GO TO Z(RETURN)



SNAPSHOT_17:GO TO Z(RETURN)

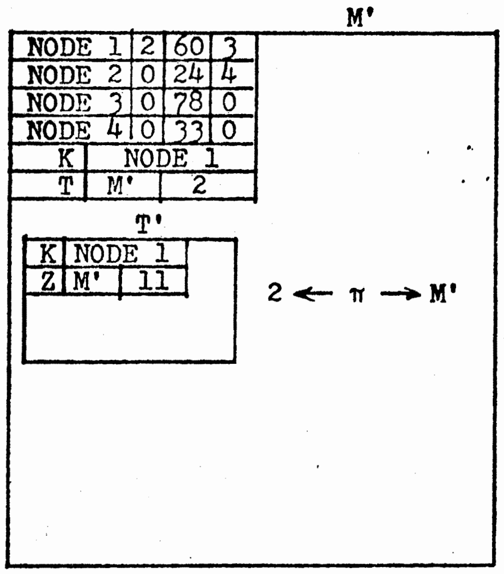


SNAPSHOT_18:PRINT NODE 1

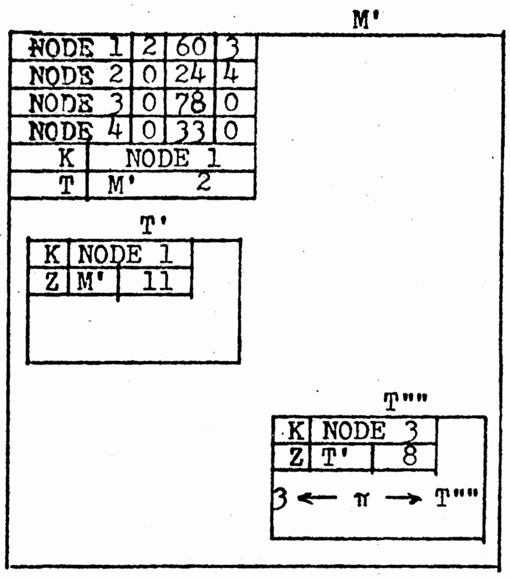


SNAPSHOT_19:RLINK(NODE 1):0

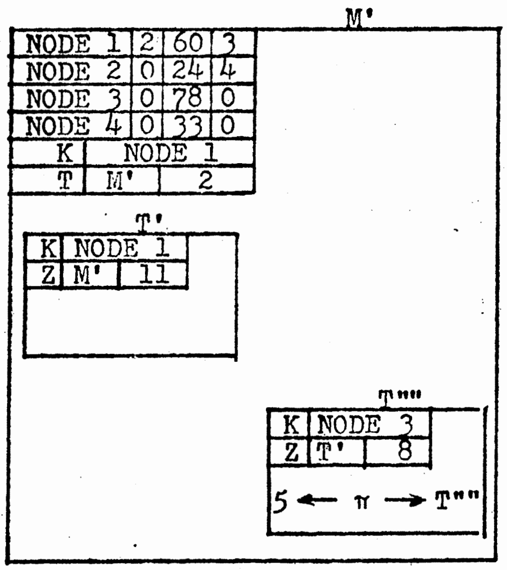
Figure 18. SNAPSHOT_16 to SNAPSHOT_19 of Example 3



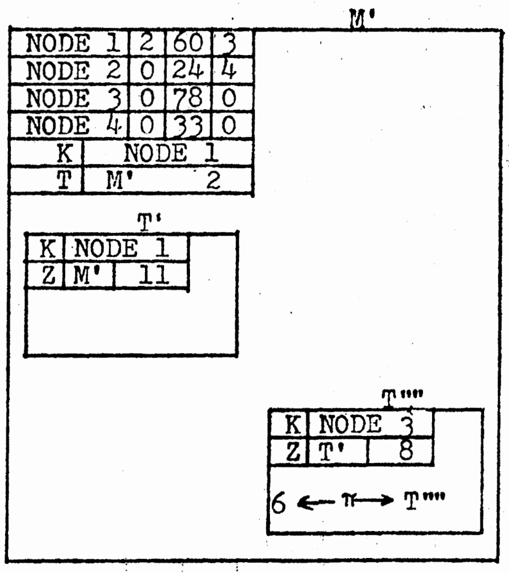
SNAPSHOT_20:CALL T(RLINK(K))



SNAPSHOT_21:ENTER T



SNAPSHOT_22:LLINK(NODE 3):0



SNAPSHOT_23:PRINT NODE 3

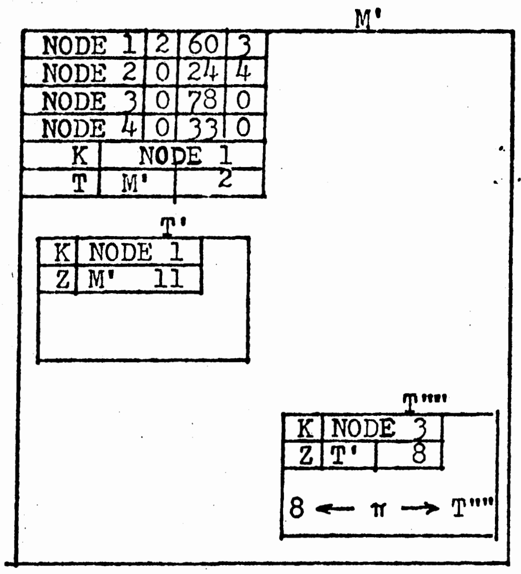
Figure 19. SNAPSHOT_20 to SNAPSHOT_23 of Example 3

performed from snapshot 22 through snapshot 24 repeat that of snapshot 13 through snapshot 15 with the exceptions that T'''' substitutes T''' and the DATA of NODE 3 is printed out in snapshot 23. Records T'''', T', and M' will be deallocated after snapshots 25, 26, and 27 respectively. Finally, the two pointers of P.soa point to nowhere, and EX_3 is terminated after snapshot 27.

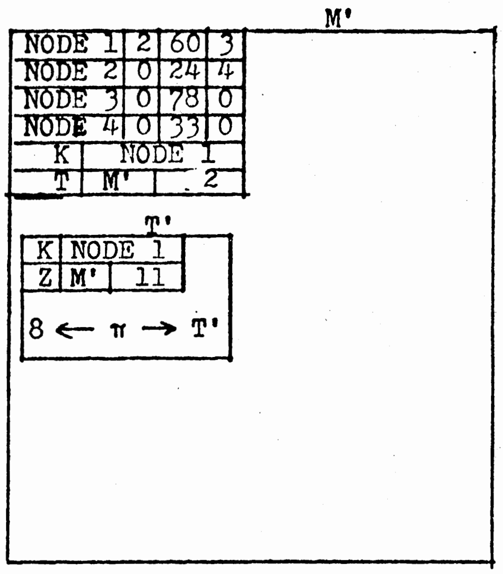
In this example, procedure T was called recursively. At each call of T, a new record contour, which is a copy of algorithm contour T, is allocated and initialized. At any moment during the execution of EX_3, if more than one copy of T exist, they all must have the same height.

Stack Model

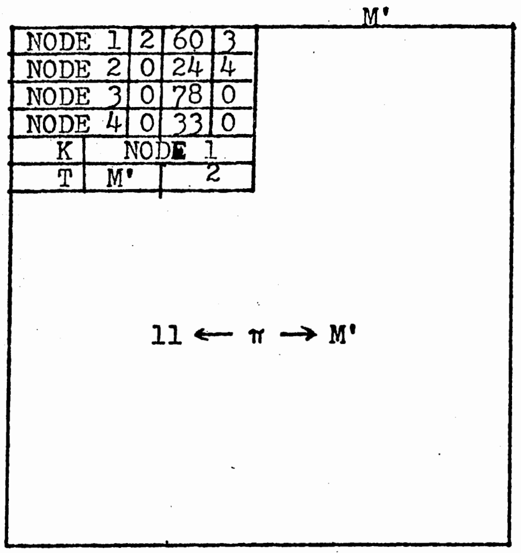
In the stack model, a display and a data area are needed for each procedure at runtime. p.170 — Gries (3) puts the display of each procedure in the first few locations of that data area, and uses a global index register, called ACTIVEAREA, to contain the address of the active display. All data within the active procedure can be accessed by using ACTIVEAREA and the display referenced by it. He also points out each procedure data area consists of the following information: the display for the procedure; a location named STACKTOP containing the address of the top stack location just after this procedure data area has been allocated; the return address; the actual parameter Display address; the global Display address; the top stack location address at the point of call; the actual parameters themselves (or their addresses); and each block data area within the procedure. He defines the return address, the actual parameter Display address, the global Display address, and the top stack location address at the point of call be four implicit



SNAPSHOT_24:RLINK(NODE 3):0



SNAPSHOT_25:GO TO Z(RETURN)



SNAPSHOT_26:GO TO Z(RETURN)

$\text{NULL} \longleftarrow \pi \longrightarrow \text{NULL}$

SNAPSHOT_27:TERMINATE

Figure 20. SNAPSHOT_24 to SNAPSHOT_27 of Example 3

Fig. 5, P. 14

parameters. Use EX_1 in Chapter I as an example, if EX_1 is executed by the stack model, the implicit parameters when procedure R is called in snapshot 8 are:

(i) return address;

(ii) procedure B's data area address;

(iii) procedure C's data area address;

(iv) STACKTOP at the call.

Abd-alla and Meltzer (1) implements a push down stack to solve the problems caused by "reentrant" procedures. The "reentrant" procedure is a kind of subroutine which can be called by one process before some other process has completed using that subroutine. They ultilizes a fixed maximum amount of storage allocated to each reentrant subroutine. The subroutine then allocates some of this space to each call made to it. After a call is completed, the subroutine frees the space allocated to that particular call and can reallocated the space to a subsequent call. Thus, the calls are completed in the reverse order in which they are called: that is, the first call made to the subroutine is the last call to be completed. Since only a fixed amount of storage is allocated for the push down stack, the mechanism to test stack overflow must be developed.

Comparison

The differences arising from the storage organization and the software mechanisms between the stack model and the contour model are discussed briefly below.

In the stack model, the data area of each block is allocated on the top of the data area of the procedure which immediately encloses that block. In the contour model, the record contour of a block is allocated separately from that of the procedure enclosing it.

In the stack model, the display and the data area of each procedure are put together, and at least one storage location serves as the stack-top to point to the address of the last memory location allocated for that data area. In the contour model, only those variables declared explicitly in a procedure and perhaps the return address reside in the record contour of that procedure.

In the stack model, the display of each procedure is stored in the first few locations of its data area and will not be changed until that data area is deallocated. In the contour model, the display of each procedure is stored in the executing processor's display registers and will be destroyed when the site of activity is changed.

When a procedure is called, there are at least four implicit parameters must be passed to and stored in the data area of the called procedure in the stack model. But this does not happen to the contour model, since even the return address is treated as an actual parameter (at the model level) and passed to the data area of the called procedure.

✓ The contour model simplifies the execution of a block enter or a block exit. When a block is entered, its record contour is allocated and adjoined to the top of the active environment. When a block is exited, its record contour is removed from the top of the active display register.

Neither the actual display nor the global display is needed in the contour model. Since the processor's two pointers, P.soa.ep and P.soa.ip, always specify the current site of activity, the active display can be created through the static link of each record contour and the height of the instruction being executed. The AHPL description in snapshot 12 of example 1 in Chapter I is an example to construct a new display.

In the contour model, there is no limitation on the number of calls to a reentrant procedure, and there is no restriction that the first call on the reentrant procedure be completed last.

Modifications

In Johnston's paper, array allocation is never mentioned. In EX_1 and EX_2, array elements are treated as different variables, but this is not the true situation in the contour model. If we regard an array as a special matrix, then the number of dimensions of that array is always known at compilation time, but not necessarily the number of values in each dimension if a flexible array[1] is implemented. The way to allocate an array is to allocate an array descriptor from the algorithm contour in which that array is declared at compilation time , and allocate the array itself somewhere else at runtime. The contents of the array descriptor may be left unspecified for a flexible array, but will be initialized at runtime. Hedrick (5) implements a set of storage locations called a "heap" to store the elements of an array at runtime. The following from (5) demonstrates the allocation of arrays.

> Assume that an ALGOL 68 particular program contains the declarations:
> flex (1:0) real A1, A2, A3; and the assignations;
> A1:=(1.0, 2.0);
> A2:=(3.0, 4.0, 5.0);
> A3:=(6.0);
> These assignations set new bounds for A1, A2, and A3 respectively and cause three consequtive allocations on the heap (p. 25).

[1]If the size of an array is not fixed during runtime, then it is a flexible array. For example, if the dimension of an array declared in a block as A(I,J,K), where I,J, and K are variables, then A is a flexible array.

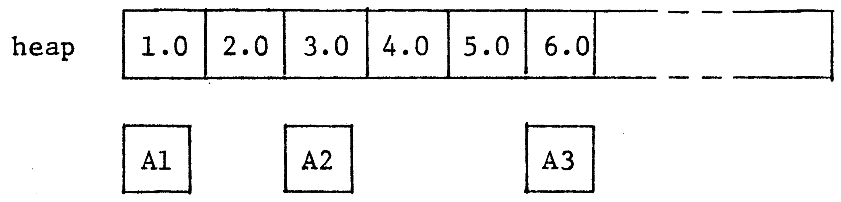


Figure 21. Heap

If the heap were empty previously, it would now look as shown in Figure 21 after A1, A2, and A3 are allocated on the heap.

Statement of the Problem

From the previous paragraphs, it is obvious that the contour model has at least three advantages over the stack model. They are:

(i) ease of multiprocessing;

(ii) it is easier for the compiler writer to generate code - the hardware handles problems of block entry and block exit;

(iii) pseudo-parallel processing is easier.

The purpose of this paper is to make a preliminary design toward a hardware implementation of the contour model. Since many places in Johnston's paper are either left for future use or left unspecified. There are some difficulties when implementing the contour model in the hardware. The objective of this paper is mainly to design the central processing unit of the contour model. This model will be called CM_1 in this paper for convenience. It specifies:

(i) how the different types of data generated and stored in the main store, how each one of these basic data can be accessed and transferred to the destination register properly;

(ii) how the virtual processors implemented together with the central processing units to handle multitasking or parallel processing;

(iii) how the instructions of CM_1 can be fetched, executed, and sequenced.

CHAPTER II

STORAGE ORGANIZATION AND HARDWARE REGISTERS

The contour model is a cell-based model (10), each of its data items is represented as a cell. A cell is composed of two parts: organization and residence (10). The organization is used to describe the general information of that particular cell. The residence consists of a set of subcells and each subcell consists of 16 bits. Several subcells contain one of the basic items: an integer, a label, or a pointer. If a cell contains more than one basic item, then it is a compound cell. Contours and virtual processors are examples of compound cells. The storage structure of each type of datum was shown in Figure 3, and is discussed in detail below.

Algorithm

An algorithm is a finite set of instructions which accomplish a particular task (9). The terms program and algorithm are used interchangeably when the program specifies the algorithm.

One programming language used with the contour model is Contour Model Assembler Language (CMAL) (11). A computer system with a CMAL translator converts those languages which have the necessary or potential contour retention properties into CMAL. If the CMAL is a two pass system, CMAL creates a symbol table, which includes all the variables and the names of algorithm contours, from the names used in the source statements and also checks for certain possible conditions and diagnostic messages during the first pass. During pass two, CMAL again examines each of the statements in the source program along with the symbol table and produces a binary-coded program and its algorithm contours (7). Instead of discussing CMAL and its translator, which are beyond the scope of this paper, some basic characteristics and its storage structure of algorithm contours are presented here.

The algorithm models the static portion of the program (5). It is specified by the syntactically correct code of that program. For each procedure or each block of the program, there is a corresponding algorithm contour. Algorithm contours remain invariant during the execution of that program and serve as templates for the formation of the contour structure of records (10). It is conceptualized that the algorithm contains a flowchart network of instructions embedded in its nested set of contours (10).

The algorithm is treated as a data structure whose basic elements are storage cells (5). As shown in Figure 3, the algorithm contour contains several subfields, each of its functions is shown in Table II.

Algorithm contours and the binary-coded program may be stored separately in the main store. This implies that the instructions of the algorithm to be executed are not stored physically in the main store as parts of the algorithm contours, although they are conceptually embedded in the nested set of algorithm contours. The relationship between the algorithm contours and instructions are discussed later in this chapter.

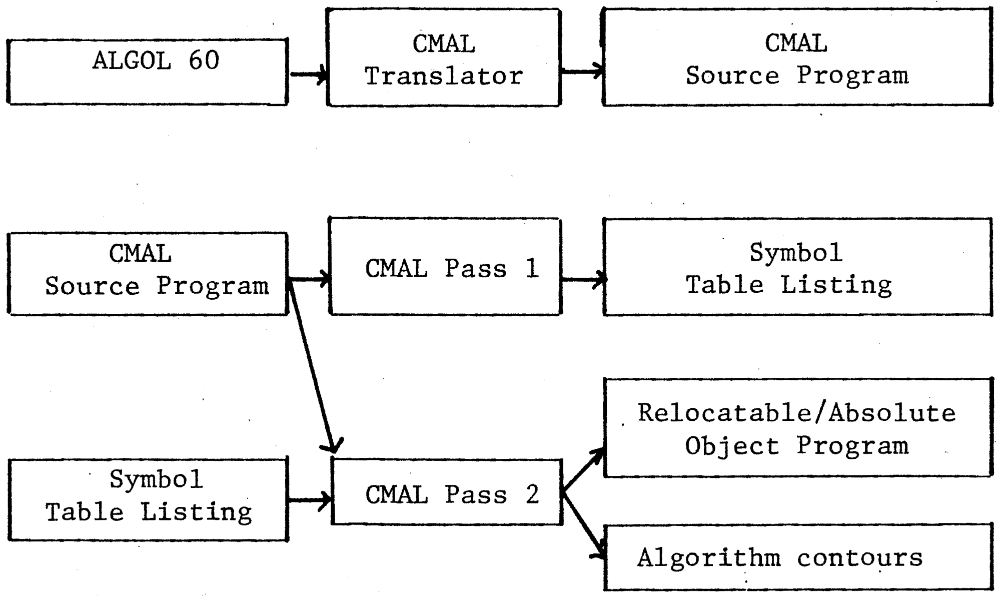


Figure 22. CMAL Processing by Using an ALGOL 60 Program as an Example

TABLE II

ALGORITHM CONTOUR

SUBFIELD	BIT POSITION	FUNCTION(S)
type	0-2	Type is used to specify the subsequent subcells are treated as an algorithm contour. The type code of algorithm contour is 111.
---	3-14	For future use.
valid bit	15	The valid bit of any algorithm contour is set off.
height	16-22	In contour model, it is nece-ssary to identify each contour of the nested set of contours. Not only for checking purpose, the height is also used to update the display registers if necessary. The range of the height of any algorithm contour can be from 0 to 127.
size	23-31	For the storage allocation for the record contour, size spe-cifies how many subcells are needed to construct that algo-rithm contour, so the same number of subcells are needed for any copy of it.
static link	32-47	The static link of an algori-thm contour points to the algorithm contour which immedi-ately enclose it.
antecedent link	48-63	This subfield is left unspeci-fied for any algorithm contour.
array subcell(s)	from 64	Each subcell of an algorithm contour contains either a local identifer or a return label. In algorithm contours, parts of the values of their subcells may be uninitialized.

Using the program EX_1 in Chapter I as an example, the algorithm contour A has four identifiers: one integer Y and three labels X, B, and Z. At the time A is allocated, Y has not been initialized yet, and X, B, Z are three invalid lables. Since algorithm contours remain invariant and will not be deallocated until the execution of that algorithm is over, there is no need to use a reference count in the algorithm contours for deallocation purposes. An algorithm contour contains an antecedent link, though it is left unspecified, either for future use or for a record contour can be constructed easily from it. The storage structure of the algorithm contour A is shown in Figure 23 with the assumption that the number on the right side of each instruction of EX_1 is the address of that instruction stored in the main store.

Record

In the stack-based model, a data area for storing display registers, actual parameters, implicit parameters, local variables, and dope vectors is needed for each procedure at runtime (3). In the contour model, a record contour plays this role. Each contour of the record of execution is a copy of some specific algorithm contour with some modifications to the contents of its subcells. If record contours A' and B' are copies of algorithm contours A and B, and if B' is immediately enclosed by A', then B is necessarily immediately enclosed by A.

The set of record contours model the sematics of the program (5). When an active processor is entering a block or a procedure, a copy of that corresponding algorithm contour is made: that is, a record contour is allocated. The algorithm contour is defined to be the antecedent of

```
 111 ----------- 0 0000001 000001011 address of M  ---------------
  a       b      c   d         e           f               g

 011 ----------- 0 0 --------------- ---------------  000 ----------
  h       i     j k        l                 m         n       o

 0 0 ---------------- 011 ----------- 1 0 ---------------- 00000000000
 p q        r          s        t     u v         w                 x

 00011 011 ----------- 0 0 --------------- ---------------
   x    y       z      @ #        $                ¢
```

a: data code of algorithm contour is 111
b: unspecified field for future use
c: valid bit of algorithm contour is always set off
d: height of A is 1
e: algorithm contour contains 11 subcells
f: the static link of A points to algorithm contour M
g: the antecedent link of any algorithm contour is left unspecified
h: subcell X is a label, the data code of a label is 011
i: in this paper, inhibit boxes are left unspecified
j: the present bit of X is set off for both of its two pointers are not initialized
k: algorithm contours only contain invalid labels
l: the environment pointer of X has not been initialized yet
m: the instruction pointer of X has not been initialized yet
n: subcell Y is an integer, the data code of an integer is 000
o: the same as i
p: Y has not been initialized yet, its present bit is set off
q: Y is invalid
r: Y has not been initialized
s: subcell B is a label
t: the same as i
u: the present bit of B is set on for its ip is not null
v: the same as k
w: the environment pointer of B has not been initialized yet
x: the instruction pointer of B points to instruction 3
y: subcell Z is a label
z: the same as i
@: the same as j
#: the same as k
$: the environment pointer of Z points to nowhere
¢: the instruction pointer of Z has not been initialized yet

Figure 23. An Example of The Data Structure of An Algorithm Contour

the record contour if the latter is a copy of the former, while the record contour is the decedent of the algorithm contour (10). When an active processor is exiting an algorithm contour, the record contour pointed by the environment pointer will be deallocated after executing that EXIT instruction. Since the record contours will be allocated and deallocated during the execution of an algorithm, they form a set of time-variant data structures.

Record contours are treated as data structures whose basic elements are storage cells. As shown in Figure 3.e., the function and bit position of each subfields are the same as those of an algorithm contour except two subfields: the reference count and the antecedent link.

The reference count is maintained equal to the number of pointers which point to the record contour for the purposes of deallocation (10). As mentioned in Chapter I, whenever a record contour is allocated, the service routine GETAREA (3) returns the address of the first memory location available for that record contour to the pointer register of the executing processor, and the reference count of that record contour is increased by one which is the consequence of the pointer in the pointer register. If a record contour has a zero reference count, it can no longer be accessed and will be deallocated during the next garbage collection (3, 5, 9). The third bit through the ninth bit of a record contour are reserved for the reference count, so a total of 127 pointers to a record contour are allowed.

Each record contour, say C', is a copy of some specific algorithm contour, say C, then C is the antecedent of C', or C' is the decendent of C. One algorithm contour may have more than one decendent, but each record contour has only one antecedent (10). The antecedent link of a record contour specifies its antecedent. As in Chapter I, π is used to denote a processor, its two pointers (ep and ip) are the focus of current control, the antecedent of the record of execution pointed by ep must immediately enclose the instruction pointed by ip at any moment.

The valid bit of a record contour is used to specify its accessibility. When a record contour is newly created and initialized, its valid bit is set on for it becomes the innermost environment. At any time, only those records of execution with their valid bits on constitute the current environments: that is, their subcells can be accessed by the processor through its display registers. Record contours with their valid bits off cannot be accessed.

If record contour C' is a copy of its antecedent C, and C contains no arrays, the formation of C' is described as follows:

(i) the subfield size of C decides the number of subcells (excluding the organization part) needed for the allocation of C';

(ii) if C is the associated algorithm contour of a called procedure, then the contents of those subcells in C' are copies of actual parameters;

(iii) if C is an algorithm contour with its height equal to zero, or C is an algorithm contour of a block, then the contents of subcells in C' are copies of the corresponding subcells in C except for those subcells holding a label. Since any algorithm contour contains only incomplete labels; that is, labels consist of a null environment pointer (ep) and a non-null instruction pointer (ip), the conversion of those incomplete labels into complete labels during the allocation of a record contour is necessary. The conversion consists of coping the ip of the corresponding incomplete label and inserting an ep pointing to the newly allocated record contour.

Again, using program EX_1 as an example, the storage organization of the record contour A' after it is allocated is shown in Figure 24. It is the direct result of those steps in snapshot 5 of EX_1 in Chapter I.

Processors

In the contour model, there are additional elements together with the ep and the ip to make the processor serve as the locus of the execution of an algorithm. In order to enable the contour model to handle simulation languages, operating system processes, multiprogramming, or coroutine (6), a set of processors are implemented. A processor in this scene is not a hardware processor, but rather a virtual processor and a set of hardware registers in the CPU. A contour model with 8 virtual processors VP_i $(0 \leq i \leq 7)$, and three central processing units, CPU_j $(0 \leq j \leq 2)$, as an example is shown in Figure 25.

In Figure 25, VP_2 and CPU_0 form a processor pair (VP_2, CPU_0), which is executing PROCESS I, while (VP_5, CPU_2) and (VP_3, CPU_1) are executing PROCESS II and PROCESS III, respectively. Since this model contains three central processing units, there are at most three processes currently being executed: that is, there are at most three of the eight virtual processors awake, while the rest are either asleep or terminated.

100	0000001	-----	1	0000001	000001011	address of M'	address of A	011
a	b	c	d	e	f	g	h	i

-----------	1	1	address of C'	0000000000001001	000	-----------	0 0
j	k	l	m	n	o	p	q r

0000000000000001	011	-----------	1	1	address of A'	0000000000000011
s	t	u	v	w	x	y

011	-----------	1	1	address of C'	0000000000001101
z	@	#	$	¢	*

a: data code of record contour is 110
b: the reference count of A' is 1
c: unspecified field
d: valid bit is set on
e: height of A' is 1
f: record contour A' contains 11 subcells
g: the static link of A' points to M'
h: the antecedent link of A' points to A
i: subcell X is a label, the data code of a label is 011
j: in this paper the inhibit boxes are left unspecified
k: the present bit is set on whenever that subcell contains a complete label
l: record contours contain only valid labels
m: the environment pointer of X points to record contour C'
n: the instruction pointer of X points to instruction 9
o: subcell y is an integer, the data code of an integer is 000
p: the same as j
q: the present bit of Y is set on
r: the valid bit of Y is set on
s: the value of Y is 1
t: subcell B is a label, the data code of a label is 011
u: the same as j
v: the same as k
w: the same as l
x: the environment pointer of B points to record contour A'
y: the instruction pointer of B points to instruction 3
z: subcell Z is a label
@; the same as j
#: the same as k
$: the same as l
¢: the environment pointer of Z points to record contour C'
*: the instruction pointer of Z points to instruction 13

Figure 24. An Example of The Data Structure of A Record Contour

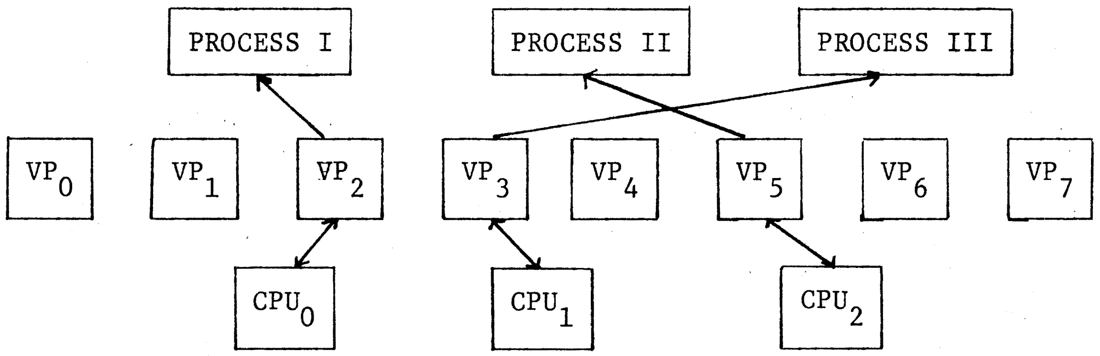


Figure 25. Multiprocessors

Virtual Processor

A virtual processor is not a set of devices in a model; rather, a sequence of memory locations, which is used to realize what is called a site of activity (SOA). At any time, a valid virtual processor is executing or about to execute a process, which is identified by the SOA of that virtual processor. The function and the dimension of each subfield of a virtual processor are shown in Table III.

The reference count of a virtual processor P.ref has the same functions as that of a record contour. The state of a virtual processor specifies its current state. The site of activity of a virtual processor P.soa contains a label, which specifies both the environment and the instruction being executed or to be executed. The instruction height of a virtual processor contains a copy of the height of the instruction pointed by P.soa.ip. Besides the pointer register P.ptr, there are two other pointer registers containing the general purpose register pointer P.gpr.ptr and the display register pointer P.dsp.ptr separately. For general purposes (14), 16 pseudo working registers are used for each valid virtual processor. Each general purpose register P.gpr.i,

TABLE III

VIRTUAL PROCESSOR

SUBFIELD	SYMBOL	BIT POSITION	FUNCTION(S)
type	P.type	0-2	
reference count	P.ref	3-9	deallocation

state	P.sta	10-11	P.sta=00 invalid
			P.sta=01 terminated
			P.sta=10 asleep
			P.sta=11 awake

site of activity	P.soa	16-47	designates the current environment and the address of the instruction being executed
label register	P.lab	48-79	contains the new site of activity when a procedure is called or used as temporary storage
instruction height	P.ih	80-86	specifies the height of the instruction pointed by P.soa.ip and is used to update the display registers
pointer register	P.ptr	96-111	contains the address of the memory location from which a set of subcells is allocated by system service routine GETAREA
general purpose register pointer	P.gpr.ptr	112-127	points the pseudo GPR_0 if the virtual pro cessor is either awake or asleep, and is set equal to null otherwise
display register pointer	P.dsp.ptr	128-143	if P.sta is either awake or asleep and P.ih=n, then P.dsp.ptr points to P.dsp.n-1, and is set equal to null otherwise

$0 \leq i \leq 15$, is a 16-bit storage location, so a total of 256 consecutive bits are needed for a valid virtual processor. The address of P.grp.0 is stored in the general purpose register pointer register P.gpr.ptr. The reason that a virtual processor contains a general purpose register pointer instead of a set of general purpose registers is that there is no need to use 32-byte storage cells for an invalid or a terminated virtual processor. Since the number of display registers of a virtual processor fully depends on the value of P.ih, also it wastes memory locations to keep display registers for an invalid virtual processor, it is better for a virtual processor to hold a display register pointer which points to the current innermost environment. As shown in Figure 26, P.dsp.ptr contains the address where P.dsp.3 located if the instruction height is set equal to 4. If P.ih is 0, then set P.dsp.ptr equal to null. The hexidecimal number on the left side of each subfield is the address of that memory location.

When a virtual processor is allocated, its state is initialized invalid and all its other subfields are initialized empty. When an invalid virtual processor becomes a valid virtual processor through the execution (by another valid virtual processor) of the INITIALIZE-PROCESSOR instruction, discussed in Chapter IV, its state becomes one of the three possible states: awake, asleep, or terminated, and its site of activity register contains a valid label which designates both the environment and the address of the first instruction of a process to be executed at that time. A valid virtual processor is deallocated only when its reference count is 0, the storage locations reserved for its display registers and its general purpose registers must be returned

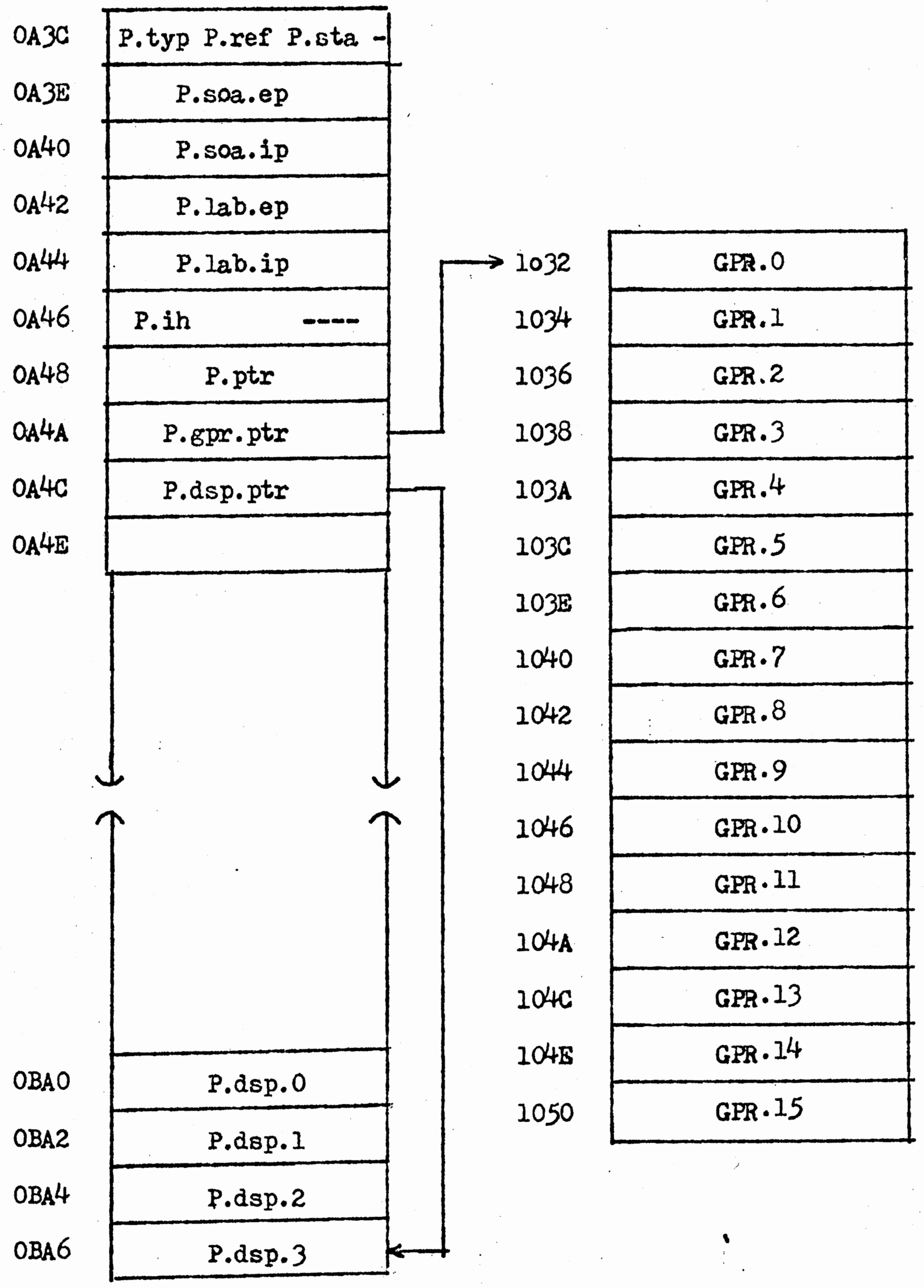


Figure 26. The Storage Organization of a Virtual Processor

to the free storage pool at the same time.

When CM_1 is ready to execute a process, the first thing for the system to do is to initialize a single virtual processor. If P is the virtual processor used in the snapshot 0 of any of the three examples in Chapter I, the display register pointer P.dsp.ptr is set null and its other fields are set as follows:

P.sta: awake;

P.soa.ep: null;

P.soa.ip: the address of the first instruction of the process to be executed by P;

P.lab.ep: null'

P.lab.ip: null;

P.ih: empty;

P.ptr: null;

P.grp.ptr: a pointer to a memory location from which 16 pseudo general purpose registers are reserved;

P.ref: equal to 1.

The reference count of P is set equal to 1, reflecting the fact that the awake state must have a retentive effect (10).

At any moment, the addresses of all valid virtual processors must be kept, through which they can be retrieved. In CM_1, a virtual processor table is necessary. Its functions are mostly like that of a symbol table except it is a dynamic data structure. For convenience, indexes are used to distinguish different virtual processors. For example, three virtual processors, VP_1, VP_2, and VP_3, are currently valid with their states awake, asleep, and terminated, respectively, then the virtual processor table looks like Figure 27.a. If some time later, a new virtual processor VP_4 is allocated, then the system service routine not only returns the address of the first memory location allocated for VP_4 to the pointer register P.ptr, but also to the virtual processor table, as shown in Figure 27.b. In Figure 27.b, the address of each virtual processor is arbitrarily chosen. Since VP_4 is just allocated, its state is still invalid.

The virtual processor table is not fixed, its size is increasing or decreasing from time to time if some virtual processors are allocated or deallocated. If any virtual processor is changed to a new state, the virtual processor table must be updated at that time also.

There are many advantages of keeping a virtual processor table. Simplifying the service routine when intercommunication between virtual processors is necessary is an example. Also, when a new virtual processor is to be allocated, the virtual processor table can furnish the information regarding whether there is a terminated virtual processor. If there is, then the allocation of that new virtual processor is not necessary since an update of the terminated virtual processor creates a new virtual processor.

Central Processing Unit

In CM_1, a central processing unit (CPU) and a virtual processor (VP) form a processor pair (VP_i,CPU_j), here i and j are indexes of virtual processor and central processing unit, respectively. So, CM_1 can execute in parallel as many processes as the number of central processing units

INDEX	STATE	ADDRESS (hexidecimal)
1	11	0402
2	10	012A
3	01	0234

a) Virtual Processor Table
Before VP_4 is Allocated

INDEX	STATE	ADDRESS (hexidecimal)
1	11	0402
2	10	012A
3	01	0234
4	00	0374

b) Virtual Processor Table
After VP_4 is Allocated

Figure 27. Virtual Processor Table

and as many tasks as the system permits virtual processors.

In CM_1, CPU is the hardware device which is responsible for the execution of a process with the assistance of a virtual processor. When a program is ready to be executed, the site of activity of that process is specified by P.soa of that associated processor. If the instruction pointed to by P.soa.ip is not the first instruction of a program, then the contents of all subfields of the associated virtual processor must be transferred to appropriate registers in CPU first, then CPU can execute that program properly.

Before presenting the method of how a processor pair executes a program, a set of hardware registers in the CPU must be defined first, then the concept of processor pair can be discussed in detail in Chapter III.

In this section, we mainly deal with the dimensions of those working registers and the data path between them. Some of the functions of each of those registers will not become clear until the processor pair and the control unit discussed in the next two chapters are covered.

There are many different kinds of registers needed in the CPU for different purposes. A register can be a shift register, a rotate register, an accumulator, a status register, an index register, a floating point register, a general purpose register, or one of the control registers. In this paper, only control registers will be discussed in detail, though most other kinds of registers are still used in the CPU of CM_1 in their usual ways.

Memory Access Registers. A memory address register (MAR) is used to contain the address of the memory location to be accessed when either a read instruction or a write instruction is being executed. Associated with the MAR, a memory data register (MDR) is used to contain the information which is either to be stored in or to be read out from the location specified by MAR. Both MAR and MDR are 16-bit registers.

Site of Activity Register. The address and the environment of the instruction to be executed are always specified by the site of activity registers (SAR). It is a pointer pair register. Its first 16 bits, which is denoted by SAR_0, contains a pointer pointing to the current innermost environment. Its second 16 bits, which is denoted by SAR_1, contains a pointer pointing to the instruction to be executed.

Instruction Register. The instruction register (IR) contains the instruction being executed. It is a 6-byte register. The first two bytes, denoted by IR_0, contains the operation code and the instruction height. In case the type of the instruction being executed is other than a register-register instruction, the second two bytes, denoted by IR_1, contains either the first operand or the address of the first operand of that instruction, and the third two bytes, denoted by IR_2, contains the second operand or the address of the second operand. When a register-register instruction is being executed, the two bytes of IR_1 contain the two indexes of the two general purpose registers, respectively.

Organization Register. Since each type of data has a different storage organization, a 32-bit organization register is needed to furnish the information about what type of data being accessed and how to access that type of data. Since the organization portion of a particular type of cell is composed of different subfields and each subfield furnishes important information for that particular cell to be accessed correctly, the organization register has multiple functions in CM_1. For example, the tag of an instruction specifies whether a direct or an indirect address is used in that instruction. Another example is the size of a record contour, it specifies how many subcells are in the residence part of a record contour, so how many memory locations are needed when it is to be allocated or how many memory locations will return back to the free pool when it is deallocated can be decided. But those subfields cannot be used properly until the type is decoded first. That is, always load the first two bytes of the target cell in a memory reference instruction to the leftmost 16 bits of the organization register (OR_0), decode the first three bits of OR_0 and decide the type of the target cell: if it is an integer, a pointer, or a label, then the rest bits of OR_0 contain inhibit boxes, present bit, and valid bit; if it is an instruction, the rest bits of OR_0 contain the format and the tag of that instruction; if it is either an algorithm contour or a record contour, load the second two bytes of the target cell into the second 16 bits of the organization register (OR_1), so the rest bits of OR_0 contain the reference count, and OR_1 contains the height and the size; if it is a virtual processor, the rest bits of OR_0 contain the reference count and the state.

Label Register. The label register is used when a new label is created. The lable register is also used to contain the return address display registers are memory (main mem.).

when a procedure is called. Since a valid label consists of two pointers, an environment pointer and an instruction pointer, two 16-bit subregisters are needed. The first subregister (LR_0) contains the environment pointer and the second subregister (LR_1) contains the instruction pointer.

Pointer Registers. The functions of subfields P.ptr, P.gpr.ptr, and P.dsp.ptr of a virtual processor were discussed before. In the central processing unit of CM_1, three registers PR_0, PR_1, and PR_2 play the same roles as P.ptr, P.gpr.ptr, and P.dsp.ptr do in a virtual processor, respectively. Each of them is a 16-bit register.

Static Pointer Register. The static pointer (link) register (SPR) contains the static link of the instruction being executed currently. The function of SPR is used to check if the current instruction is well organized: that is, to check if the algorithm contour pointed by SPR is the antecedent of the currently innermost record contour. If it is not, then an interrupt will be caused. The length of SPR is 16 bits.

Data transfer between registers in the CPU is necessary during the execution of an algorithm. In CM_1, a common bus connects registers and main store. Each register has an "out" gate and an "in" gate which enables us to put information on the bus and take it off the bus when and where we want (2).

The block diagram of a central processing unit, together with the main store, is drawn in Figure 28. The common bus is a 16-bit OR gate.

Instructions

The Contour Model Assembler Language (CMAL) is left unspecified

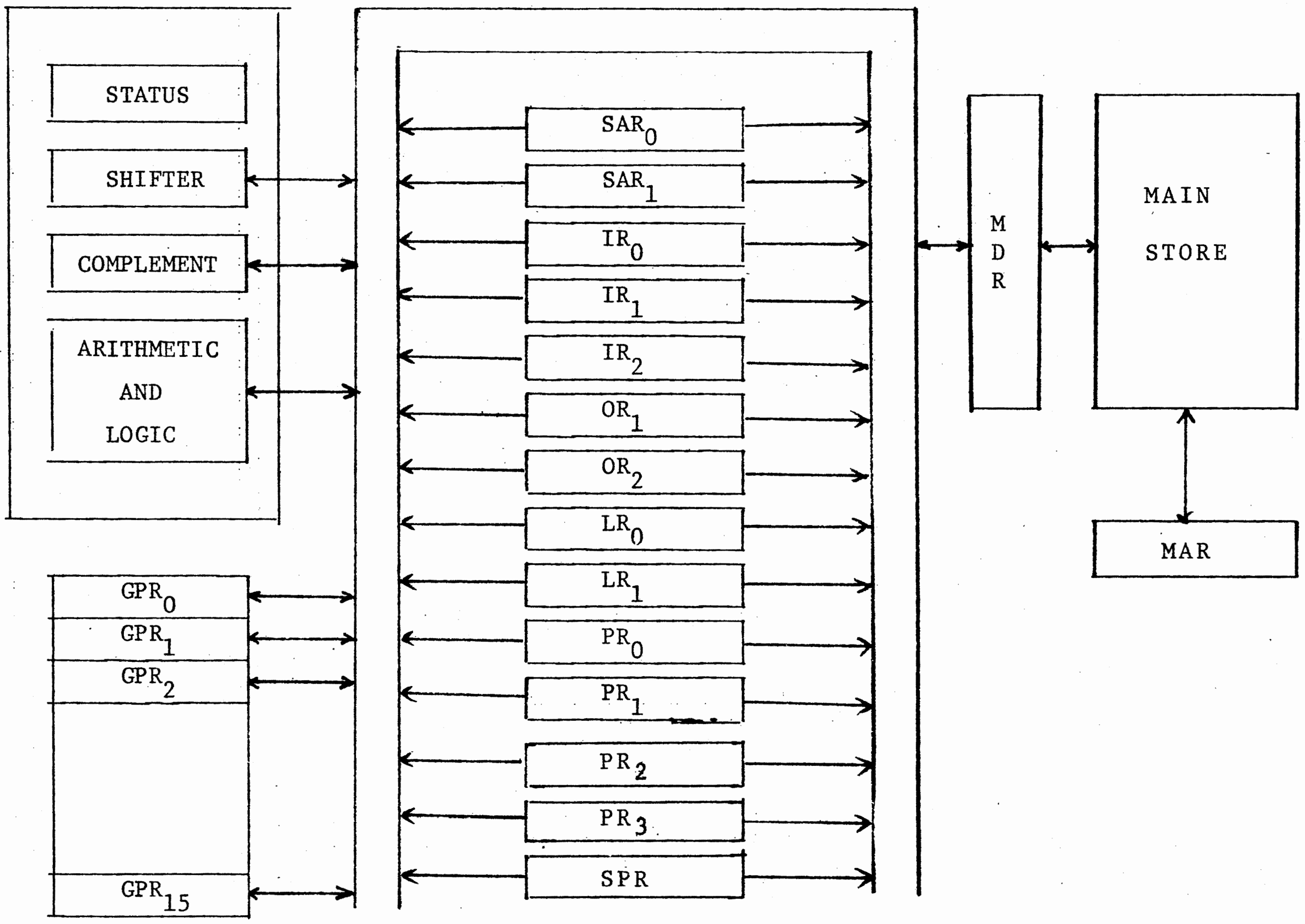


Figure 28. Central Processing Unit

in this paper. However, for the purpose of discussing its storage organization, some basic ideas of CMAL are presented here. At least five formats of instructions appear in CMAL, as shown in Figure 29. In Figure 29, the instruction length and the format code of each format of instructions are also shown, and the number of bits in each subfield of the instruction is shown in the upper right-hand of each subfield.

The format of CMAL offers the information to the control unit regarding how many bytes are to be accessed and transferred to appropriate working registers in the CPU. The static link and the instruction height are used to build an appropriate relationship between the instruction, say I, and the algorithm contour, say C, which immediately encloses I, the static link of I must point to C and the instruction height of I must be one greater than the height of C. The instruction height plays an important role in CM_1. At any moment, the available display register with the highest index is equal to the instruction height of the instruction being executed minus one. In other words, the display register with index i always points to a record contour with height i. As mentioned earlier, the algorithm contour and its associated instructions can be stored in separate memory locations, however, the relationship between these two data structures must be specified by their heights, so the algorithm can be executed properly. The successor link of an instruction points to the instruction to be executed next under normal conditions. In CM_1, successor links and lables are used for the processor's flow of control. The display register index and the subcell index are used to determine the address of an operand in an instruction which accesses memory. The memory location with its address specified by both the

Register-Main Store (RM) length=12 bytes, Code=100

A 3	B 3	C 3	D 7	E 16	F 16

G 9	H 7	I 7	J 9	K 16

Main Store-Main Store (MM) Length=12 bytes, Code=101

A 3	B 3	C 3	D 7	E 16	F 16

G 9	H 7	I 7	J 9	I 7	J 9

Main Store-Immediate (MI) Length=12bytes, Code=110

A 3	B 3	C 3	D 7	E 16	F 16

G 9	H 7	I 7	J 9	L 16

Register-Immediate (RI) length=12 bytes, Code=001

A 3	B 3	C 3	D 7	E 16	F 16

G 9	H 7	K 16	L 16

Register-Register (RR) Length=10bytes, Code=010

A 3	B 3	C 3	D 7	E 16	F 16

G 9	H 7	K 8	K 8

A: type B: format C: tag
D: unspecified field E: static link F: successor link
G: operation code H: instruction height
I: display register index J: subcell index
K: general purpose register index
L: immediate operand

Figure 29. Instruction Format

display register with index i and the subcell with index j contains either an operand or the address of an operand fully depend on the tag. If it contains an operand, then direct address is used, if it contains the address of an operand, then indirect address is used. For example, in the snapshot 3 of EX_2 in Chapter I, store the immediate operand 2 into subcell B, the corresponding CMAL will look like

STI   0(3),2,   (2.1)

where 0 is the index of the display register (since record contour M' is pointed by P.dsp.0), the integer 3 in the parentheses is the subcell index (since subcell B is the third subcell in the residence part of record contour M' excluding the organization part of record contour M'), and the integer 2 is the immediate operand. The organization part of a record contour consists of 8 bytes, so the effective address of B is

$$\text{effective address of B} = (\text{p.dsp.0}) + 8 + 2 \times (3-1) \qquad (2.2)$$

here (P.dsp.0) is the contents of P.dsp.0, and 2 x (3-1) is the offset of subcell B in record contour M'. Each subcell is 2 bytes in length and the third subcell means there are two subcells before it. The statement (2.2) is the effective address of subcell B if direct address is specified by the tag of the instruction (2.1). Other methods through which an operand can be accessed is by using the pointer register or a general purpose register which contains a pointer to somewhere in the main store. It is not difficult to develop variants and extensions of addressing methods (10). The effective address of an operand are discussed in detail in Chapter IV.

Allocation and Deallocation

The storage allocation and deallocation of record contours and virtual processors are necessary when an algorithm is being executed.

Since each record contour is a copy of some algorithm contour, which is allocated at compile time, with the number of subcells needed specified in its size, so a record contour occupies the same number of memory locations as its antecedent does. If a virtual processor is to be allocated, its display registers will not be allocated until its height is initialized. The storage allocation can be accomplished by using any one of the three methods: First Fit, Best Fit, or Buddy System. The details of these three methods are beyond the scope of this paper, so only their basic ideas are presented here. If the system service routine GETAREA is called to request a block of memory of N subcells, First Fit will search down the list of free blocks finding the first block containing more than or equal to N subcells; Best Fit will search down the list of free blocks finding a free block containing subcells as close to N as possible, but not less than N; while the Buddy System will search down the available list, each of which is a set of free blocks of equal size (a power of 2), finding the list whose elements containing no less than N subcells, then taking the first block of that list and repeatedly dividing that block if necessary until finding a subblock of smallest size but still greater than or equal to N (12, 13). In CM_1, any one of the three methods above can be used.

The reference count in the data structures discussed in this paper is used mainly for deallocation purpose. When any data structure possesses a reference count equal to 0, then the memory location it occupies will return to the available storage pool during the next garbage collection. If storage compaction is necessary, then the implementation of a relocation register associated with each process is needed to find the effective address of each subcell.

Heap

If a procedure or a block contains a n-dimension array $A(L_1:U_1,L_2:U_2, \ldots,L_n:U_n)$, with the L's and U's are lower bounds and upper bounds, respectively, then its algorithm contour contains an array descriptor (a dope vector). If the lower and upper bounds of an array are known at compilation time, then CM_1 can allocate storage cells for the array within the contour and generate the code for referencing the array elements using the lower and upper bounds. If the bounds are not known until runtime, a fixed size of space (depending on n) is allocated to the descriptor in the algorithm contour with which the array is associated. Although the contents of that space are left unspecified, the storage cells for the array itself will not be allocated until the procedure or the block in which the array is declared is entered. The reason for a fixed amount of space can be allocated for an array in the algorithm contour is that its dimension is always known at compilation time (3).

The elements of an array which is located in continuous memory space are placed either in ascending or descending order, either in row major or in column major (3). The usual way is to store them in a data area by row and in ascending order. For example, if A(i,j. ...l,m) is the element to be accessed, then its address is found to be

$$\text{BASELOG} + (i-L_1)xd_2xd_3x\ldots xd_n + (j-L_2)xd_3x\ldots xd_n + (1-L_{n-1})xd_n + (m-L_n), \qquad (2.3)$$

where BASELOG is the address of A(1,1,...1) and $d_1=U_1-L_1+1$, $d_2=U_2-L_2+1, \ldots, d_n=U_n-L_n+1$. For convenience, the element's address can be factored and it turns out to be

$$\text{BASELOG} - ((\ldots((L_1xd_2+L_2)xd_3+L_3)xd_4+\ldots+L_{n-1})xd_n+L_n) + (\ldots((ixd_2)+j)xd_3+\ldots+1)xd_n+m \qquad (2.4)$$

If the bounds of A are known, then the middle term in (2.4) is a constant while the third is a variant (3).

With the knowledge of how to find the address of an element in an array, it seems natural for an array descriptor to have the structure as shown in Figure 30.

	n	BASELOG	L_1	U_1	d_1	L_2	U_2	d_2	...
...	L_n	U_n	d_n	CONSTANT					

Figure 30. Array Descriptor

If the bounds of an array are known at compilation time, then the algorithm contour with which that array is associated contains an array descriptor of which subfields are properly initialized, but the data structure of the array itself will not be allocated until the procedure or the block in which the array is declared is entered. This means the BASELOG is left unspecified until execution time. If the bounds are not known at assembly time, an uninitialized array descriptor is allocated within the associated algorithm contour. The flexible array is allocated out of the heap at execution time. The heap contains a set of subcells which, in theory, are not associated with any contour (5). In the case of flexible arrays, the base address points into the heap and the values of L_i and U_i in the descriptor are changed as required. It is sometimes necessary to change the base address as well (5).

Summary

(1) The algorithm contours model the static portion of the programs.

(2) The set of record contours model the semantics of the programs.

(3) Record contours are copies of algorithm contours, a record contour has the same data structure as its antecedent.

(4) A virtual processor consists of a set of pseudo-registers which are allocated in consecutive memory locations except its display registers and general purpose registers.

(5) The Central Processing Unit (CPU) is a set of hardware devices. CPU and an virtual processor constitute a processor pair (VP, CPU).

(6) If the bounds of an array are known at compilation time, an array descriptor is allocated within the algorithm contour with which the

array is associated, the array descriptor is properly initialized but its BASELOG is left unspecified. The array itself will not be allocated until the execution of the procedure in which the array is declared is entered.

(7) If the bounds of an array are not known, or the attribute of an array is changeable, an uninitialized array descriptor is allocated within the algorithm contour with which the array is associated. The array itself will be allocated out of a heap when the procedure in which the array is declared is entered.

(8) The heap contains a set of subcells which, in theory, are not associated with any contour, except when a descriptor within a contour points to a set of heap cells.

CHAPTER III

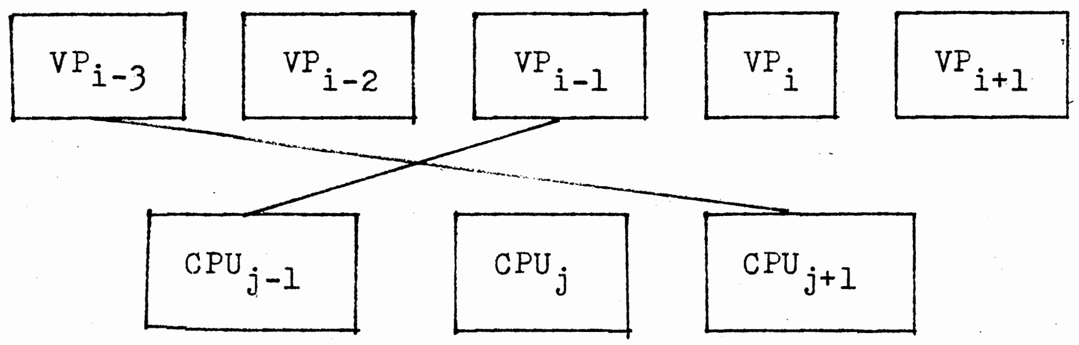
PROCESSOR PAIR

CM_1 is a parallel system in which more than one processor might be active; that is, more than one process might be being executed.

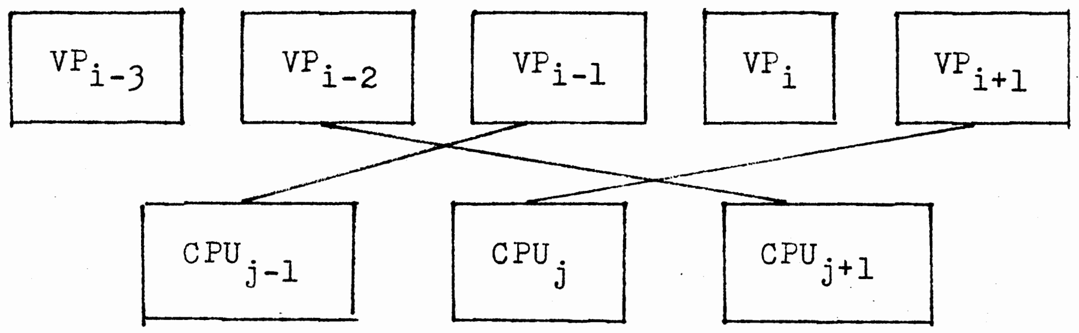
A working processor pair (VP_i, CPU_j) consists of one awake virtual processor VP_i and one active central processing unit CPU_j. The CPU_j will execute a process specified by its first instruction address and its innermost environment or by the site of activity of VP_i.

There are two possible sources that one VP with index i and one CPU with index j can be combined together as a processor pair (VP_i, CPU_j); they are either asked by the supervisor or arranged by the system service routine. The instruction to construct a new processor pair is PAIR(I,J), where I represents the virtual processor with index i, and J represents the central processing unit with index j. The value of I is specified, in this instruction, only when VP_i can be found in the virtual processor table. Otherwise, its format is PAIR(*,J). The value of J can be specified by one internal register in the CPU.

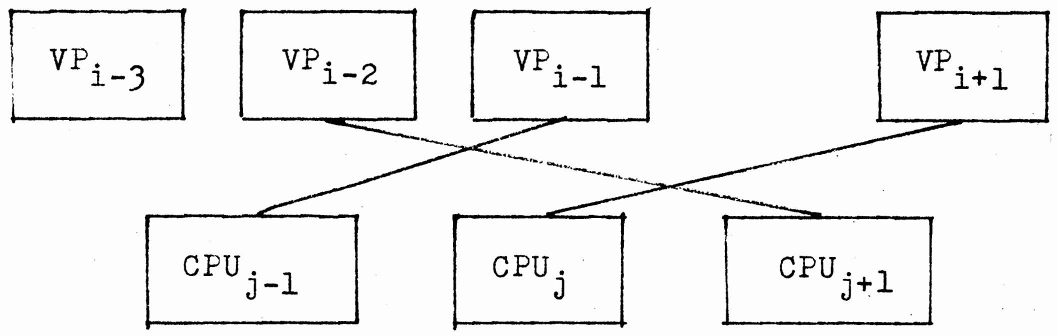
There are three possible cases at the time when a processor pair (VP_i, CPU_j) is to be created and they are shown in Figure 31. In Case 1, as shown in Figure 31.a, VP_i is either asleep or terminated and CPU_j



a.) Case 1



b.) Case 2



c.) Case 3

Figure 31. Processor Pair

is not active. In Case 2, as shown in Figure 31.b., (VP_{i-2}, CPU_{j+1}), (VP_{i-1}, CPU_{j-1}), and (VP_{i+1}, CPU_j) are three active processor pairs and VP_i is either asleep or terminated. In Case 3, as shown in Figure 31.c, all conditions are the same as Case 2 except VP_i has not yet been initialized. In Case 1, CPU_j is not active when the processor pair (VP_i, CPU_j) is asked to be created, so the instruction PAIR(I,J) can be executed immediately. However, this is not the situation in the other two cases, since CPU_j is contained in the working processor pair (VP_{i+1}, CPU_j), which is executing an instruction, say M, of a process, say PGM, the instruction PAIR(I,J) will not be executed until (VP_{i+1}, CPU_j) finishes executing M and checking whether M is the last instruction of PGM. If M is the last instruction of PGM to be executed, then the state of VP_{i+1} will be changed to be terminated. If it is not, then the address of the instruction of PGM should be executed next by (VP_{i+1}, CPU_j) and its innermost environment, together with the contents of most of the registers in CPU_j, will be stored in the site of activity and other appropriate subfields of VP_{i+1}, respectively, and the state of VP_{i+1} will be changed to be asleep. In this way, the rest of the instructions of PGM can be executed subsequently.

At the beginning of creating a processor pair (VP_i, CPU_j), if either VP_i cannot be found in the virtual processor table; that is, the VP_i has to be initialized first, or the state of VP_i is terminated, then the site of activity, of which its environment pointer is null and its instruction pointer points to the first instruction of the process to be executed by (VP_i, CPU_j), must be specified by supervisor or the system service routine. Finally, the contents of all subfields of VP_i must be fetched to the appropriate registers in CPU_j. At this moment, the new processor pair (VP_i, CPU_j) is ready to begin to execute that process. The flow chart of creating a new processor pair is shown in Figure 32.

To simplify the AHPL description of creating a processor pair, some instructions such as WAKE, ALLOC, and INIT are discussed in detail in the next chapter, and some control bits or indexes are used as follows:

(1) The registers used in the AHPL description will not be specified explicitly when they are registers in CPU_j;

(2) The bit B is used to denote whether the instruction M being executed by the processor pair (VP_{i+1}, CPU_j) is finished;

(3) An index k is used to trace loops;

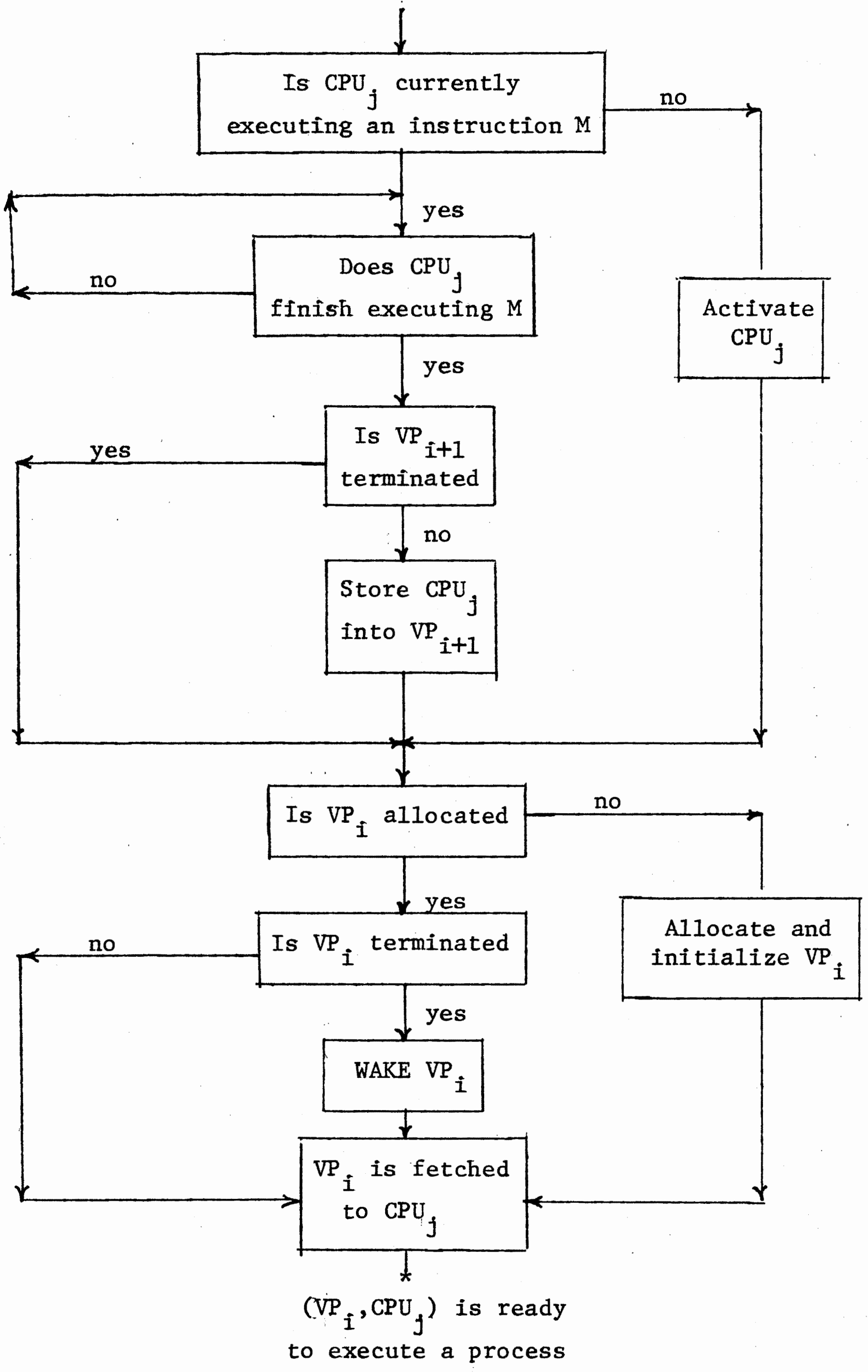
(4) Ready is a bit which denotes the creation of the new processor pair (VP_i, CPU_j) is finished and ready to execute the first instruction of the process specified by VP_i;

(5) Instruction WAKE is used to change the state of a virtual processor from either terminated or asleep to awake;

(6) Instruction ALLOC XXX is used to allocate cells for a specific data type, which is specified by its argument XXX. For example, ALLOC 100 is to allocate a virtual processor having a null initial contents;

(7) Instruction INIT is used to initialize the target virtual processor with a specified site of activity;

(8) The bit STA is used to denote if CPU_j is currently awake or asleep.



Is CPU_j currently executing an instruction M
no
yes
Does CPU_j finish executing M
no
yes
Activate CPU_j
Is VP_{i+1} terminated
yes
no
Store CPU_j into VP_{i+1}
Is VP_i allocated
no
yes
Allocate and initialize VP_i
Is VP_i terminated
no
yes
WAKE VP_i
VP_i is fetched to CPU_j
*
(VP_i, CPU_j) is ready to execute a process

Figure 32. Flow Chart of Matching a Processor Pair (VP_i, CPU_j)

The AHPL description of the execution of instruction PAIR(I,J) is depicted in Table IV.

Since the execution of some instructions need to use the contents of some registers, the contents of the registers in CPU_j must be stored into VP_{i+1} and the contents of subfields of VP_i have to be fetched to CPU_j before (VP_i,CPU_j) can execute a process. For example, the process PRO being executed by (VP_{i+1},CPU_j) is shown in Figure 33. After the instruction ADD 5,9 is finished, the next instruction to be executed should be BN 5,Label. However, the instruction PAIR(I,J) is asked by supervisor, so the address of the instruction BN 5,Label together with its innermost environment and the contents of those registers in CPU_j are stored into VP_{i+1}. In this way, the contents of GPR_5 will not be destroyed when PRO is to be executed later.

```
PRO:    .
        .
        .
        ADD  5,9              /* Add the contents of GPR5 to the
                                 contents of GPR9, and store the
                                 sum into GPR5
                              */
        BN  5,Label           /* Branch to the instruction
                                 specified by Label if the
                                 contents of GPR5 is negative
                              */
        .
        .
Label   ADD  5,7
        .
        .
        WND
```

Figure 33. Program PRO

TABLE IV

AHPL OF PAIR (I,J)

INSTRUCTION SEQUENCE	COMMENT
(1) $\longrightarrow (STA \times 2) + (\overline{STA} \times 23)$	Is CPU_j currently executing a process.?
(2) $\longrightarrow (B \times 3) + (\overline{B} \times 2)$	Does CPU_j finish executing M.?
(3) $MAR \longleftarrow ADDR(VP_{i+1})$	Get the address of VP_{i+1}.
(4) $\longrightarrow (M^{\perp MAR}{}_{10}(1) \times 5) + \overline{(M^{\perp MAR}{}_{10}(1)} \times 24)$	Test the state of VP_{i+1} is awake or terminated after executing M. If it is still awake(M is not instruction of PGM), the contents of most of the registers in CPU_j must be stored into the appropriate subfields of VP_{i+1}.
(5) INC(MAR,2)	
(6) $M^{\perp MAR} \longleftarrow SAR_0$	Store the contents of SAR_0 into P.soa.ep of VP_{i+1}.
(7) INC(MAR,2)	
(8) $M^{\perp MAR} \longleftarrow SAR_1$	Store the contents of SAR_1 into P.soa.ip of VP_{i+1}.
(9) INC(MAR,2)	
(10) $M^{\perp MAR} \longleftarrow LR_0$	Store the contents of LR_0 into P.lab.ep of VP_{i+1}.
(11) INC(MAR,2)	
(12) $M^{\perp MAR} \longleftarrow LR_1$	Store the contents of LR_1 into P.lab.ip of VP_{i+1}.
(13) INC(MAR,2)	
(14) $\alpha^7/M^{\perp MAR} \longleftarrow w^7/IR_0$	Store the right seven bits of IR_0 into P.ih of VP_{i+1}.
(15) INC(MAR,2)	
(16) $M^{\perp MAR} \longleftarrow PR_0$	Store the contents of PR_0 into P.ptr of VP_{i+1}.

TABLE IV (continued)

INSTRUCTION SEQUENCE	COMMENT
(17) $MAR \longleftarrow PR_1$	Steps from (17) to (22) are used to store the contents of GPR_k of CPU_j into GPR_k of VP_{i+1}.
(18) $K \longleftarrow 0$	
(19) $M^{\downarrow MAR} \longleftarrow GPR_k$	
(20) INC(MAR,2)	
(21) $K \longleftarrow K+1$	
(22) $K:16,(<,=) \longrightarrow (19,24)$	
(23) $STA \longleftarrow 1$	
(24) $MAR \longleftarrow ADDR(VP_i)$	If VP_i can not be found in the virtual processor table, then MAR contains a null pointer after (24).
(25) $\longrightarrow (\overline{v/MAR} \times 26) + (v/MAR \times 29)$	Test if VP_i is in the virtual processor table.
(26) ALLOC 100	Allocate VP_i.
(27) INIT 11	Initialize VP_i with its state awake.
(28) $\longrightarrow (31)$	
(29) $PR_0 \longleftarrow ADDR(VP_i)$	PR_0 contains the address of VP_i after this instruction.
(30) WAKE	Change the state of VP_i to be awake.
(31) $MAR \longleftarrow ADDR(VP_i)$	Steps (31) to (51) are used to fetch CPU_j from VP_i.
(32) INC(MAR,2)	
(33) $SAR_0 \longleftarrow M^{\downarrow MAR}$	Load SAR_0 with P.soa.ep of VP_i.
(34) INC(MAR,2)	
(35) $SAR_1 \longleftarrow M^{\downarrow MAR}$	Load SAR_1 with P.soa.ip of VP_i.
(36) INC(MAR,2)	
(37) $LR_0 \longleftarrow M^{\downarrow MAR}$	Loas LR_0 with P.lab.ep of VP_i.
(38) INC(MAR,2)	
(39) $LR_1 \longleftarrow M^{\downarrow MAR}$	Load LR_1 with P.lab.ip of VP_i.
(40) INC(MAR,2)	
(41) $\omega^7/IR_0 \longleftarrow \alpha^7/M^{\downarrow MAR}$	Load the right seven bits of IR_0 with P.ih of VP_i.
(42) INC(MAR,2)	

TABLE IV (continued)

INSTRUCTION SEQUENCE	COMMENT
(43) $PR_0 \longleftarrow M^{\lfloor MAR}$	Load PR_0 with P.ptr of VP_i.
(44) INC(MAR,2)	
(45) $PR_1 \longleftarrow M^{\lfloor MAR}$	Load PR_1 with P.gpr.ptr of VP_i.
(46) $MAR \longleftarrow PR_1$	MAR contains the address of GPR_0 of VP_i after (46).
(47) $K \longleftarrow 0$	Steps (47) to (51) are used to load GPR_k of CPU_j with GPR_k of VP_i.
(48) $GPR_k \longleftarrow M^{\lfloor MAR}$	
(49) INC(MAR,2)	
(50) $K \longleftarrow K+1$	
(51) $K;16,(<,=) \longrightarrow (48,52)$	
(52) $Ready \longleftarrow 1$	

In order to handle multitasking or coroutines, CM_1 uses a processor pair to execute a process. Strictly speaking, a CPU of CM_1 plays the same role as it does in most other models except it needs the assistance of a valid virtual processor which specifies the first instruction address and its environment; that is, the site of activity of a new process.

CHAPTER IV

CONTROL UNIT

The control unit has three major functions. The first one is to control the sequence of instructions of a program to be executed and transfer the data among different registers in the CPU. The second one is to issue signals to each unit to activate operations assigned to it at the proper time. The third one is to handle interrupts under either expected or unexpected conditions. Interrupt handling is not covered in this paper. The purposes of the control unit presented here are: fetch the instructions of a program from memory, according to the sequence designed by the program to be executed, interpret every instruction, issue the necessary signals to the proper units, and perform the operation.

Sequencing of Operations

Those parts of the control unit which use one of a set of predetermined instructions plus a clock pulse as input and produce a sequence of control signals as output, are referred to as a control sequencer. In the control sequencer, each signal other than the first one results from the preceding one by executing one operation. Each operation takes a finite time to be accomplished. The time interval between two signals is called the control delay. For example, the control sequence of the execution of an ADD instruction is shown in Figure 34 if the address of

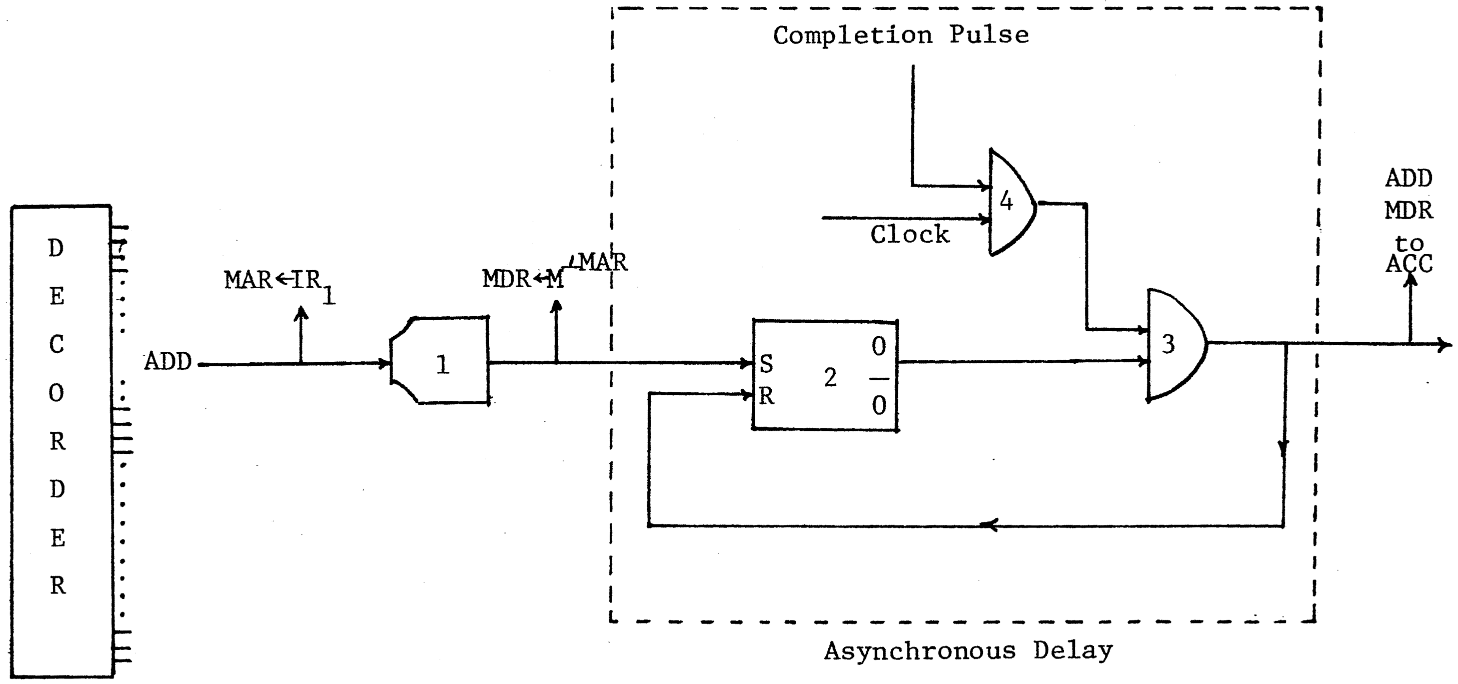


Figure 34. Control Sequencer

the addend is in the IR_1, Box 1 is the symbol for a control delay. Since some operations such as READ and WRITE take more than one clock period to be finished, the completion pulse of such instructions must be synchronized with the clock pulse before the next signal can be generated. For this purpose, the feedback flip-flop circuit (Box 2) in Figure 34 is implemented and called an asynchronous delay. Boxes 3 and 4 are two AND gates. The output of Box 3 not only resets Box 2 but also activates the next operation.

Instruction Classification and Instruction Format

There are at least six categories of instructions in CMAL. They are:

1. Read/Store instruction;
2. Jump instruction:
3. Operate instruction;
4. Register reference instruction;
5. Contour control instruction;
6. Input/Output instruction.

The Read and Store instructions are also called memory reference instructions. Memory reference instructions are used to convey the data between two memory locations or one memory location and one of the working registers in the CPU. Jump instructions are also referred to as branch or transfer instructions. A jump instruction without a comparison is an absolute jump instruction. Otherwise, it is a conditional jump instruction. As far as a conditional jump instruction is concerned, a test must be met before the jump instruction's branching address is sequenced. If the test fails, the sequencing is based on the successor link. In a conditional jump instruction, the instruction to be sequenced must have a static link identical to that of the transfer instruction: that is, the new site of activity has exactly the same environment pointer as that of the old one (10). The register reference instruction is used to act on the data transfer between two general purpose registers or one general purpose register and one working register in CPU. A load of a general purpose register with an immediate value is also completed by a register reference instruction. The operate instruction (8) can perform a logical or an arithmetic operation on the contents of one or two registers. They could be shift or rotate the contents of one register, modify the contents of a register by adding to it or subtracting from it a constant number, and test the contents of a register and take decisions based on result of that test. The input/output instructions are used to carry data between the main store and the peripheral devices. The operate instruction and the contour control instruction will be discussed in detail later in this Chapter.

Since different kinds of instructions have different length in bytes, the format of instructions must be specified. In this way, the control unit can access each of those different kinds of instructions properly. In some machines, the format of an instruction is implicitly specified by its operation code. In this paper, the instruction format is explicitly specified; thus the control of access to the instructions.can be discussed even though the CMAL is unspecified. As shown in Figure 29, the five formats of CMAL, the RR type of instruction occupies 10 bytes while the rest need 12 bytes.

Addressing

For simplicity, without loss of generality, the implementation of a base register or an index register in the determination of the effective address of an operand is not covered in this paper. As shown in Figure 29, the tag and the (i,j) pair in the address field of a CMAL are used together to decide the effective address which makes a successor access to a subcell of a record contour possible. The tag field specifies a direct addressing or an indirect addressing is being used in that particular memory access instruction. The (i,j) pair identifies the j-th subcell of the record contour pointed by the active display register with index i (P.dsp.i). If direct addressing is used, the effective address of (i,j) is the sum of the contents of P.dsp.i and j-1. Since the subcell j is counted from the residence part (exclude the organization part) of a record contour, 8 bytes occupied by the organization part must be added: that is,

$$\text{EFFECTIVE ADDRESS} = (\text{P.dsp.i}) + 8 + 2(j-1) \qquad (4.1)$$

$$= (\text{P.dsp.i}) + 2j + 6 \qquad (4.2)$$

If indirect addressing is used, then the contents of the memory location pointed to by (4.2) specifies the address of the operand: that is

$$\text{EFFECTIVE ADDRESS} = {}_{M}\!\perp(\text{P.dsp.i}) + 2J + 6. \qquad (4.3)$$

The effective address(es) of the operand(s) must be stored back into IR_1 (and IR_2) before starting of the execution cycle of that instruction: that is, before the instruction code is decoded, although there is no particular reason to do this.

Since the pointer is one of the basic types of data of the contour model, the implementation of pointers adds power to the CM_1 machine, especially in the determination of operands.

Instruction Sequencing

The execution of a CM_1 instruction consists of three stages: fetch, execute, and sequence.

In the fetch cycle, there are two tasks to perform. The first one is to fetch the instruction specified by the pointer in SAR_1 (the site of activity instruction pointer register) and transfer it to IR (instruction register). Due to the fact that P.soa.ip always points to the instruction to be executed and that SAR_1 is the hardware register corresponding to P.dsp.ip, SAR_1 points to the instruction being executed, but after the fetch cycle of that instruction, SAR_1 points to the instruction to be executed next. There are three advantages to this:

(1) It minimizes the number of working registers in the CPU, since no program counter is needed;

(2) It reduces the necessary data transfers because the successor link of the current instruction is loaded directly into SAR_1 without being transferred to a program counter first; and

(3) When the current instruction is a subroutine call instruction, SAR_1 contains the return address already.

The second task to be done in the fetch cycle is to construct a new set of display registers if necessary.

In the execution cycle, the instruction code in the first nine bits of IR is decoded to decide which instruction must be performed and an order for a proper control signal to be issued to initiate a set of finite steps to perform that instruction. If a branch instruction or a procedure call instruction is being executed, then a conditional flip-flop FF is set on. The reason for this becomes obvious in the next few paragraphs.

In the sequence cycle, nothing will be done under normal conditions. If the instruction to be executed next is not pointed to by the successor link of the current instruction, then the contents of SAR_1, which is updated in the fetch cycle, must be modified again. If the current instruction is a subroutine call instruction, then the site of activity of the called procedure is stored in the LR (label register). So, a copy of LR_0 and LR_1 into SAR_0 and SAR_1, respectively, is necessary. In the branch instruction, the address of the instruction to be executed next is specified by its operand rather than by its successor link.

Under normal conditions, the instruction to be executed next is pointed by the successor link of the current instruction, neither the environment pointer nor the display registers of the active processor must be modified. But this is not the situation if either an unconditional jump instruction or a procedure call instruction is being executed. When an unconditional jump instruction is being executed, its operand, which must be a label, specifies the new site of activity of the next instruction. That is, as long as the conditional flip-flop FF is set on, the following three things must be done:

(1) Replace the two pointers in SAR by the two pointers of the label

which specifies the destination of that jump instruction,

(2) Release the memory locations occupied by the old display registers, and

(3) Construct a new set of display registers through the static link of each record contour in the fetch cycle of the next instruction, after the new set of display registers are constructed, the condition flip-flop FF is set off.

The same scheme can be used when a procedure call instruction is being executed. The overall control diagram of the execution of an instruction is shown in Figure 35.

In Figure 35, each box is identified by a number, the numbers inside the parentheses on both sides of box 6 represent the instruction categories. Also, a MI type store instruction is used in box 13 in Figure 35. If a M-R type instruction is used, the box 13 should be

$$M^{\downarrow MAR} \leftarrow GPR_{\downarrow IR_2} . \tag{4.4}$$

That is, store the contents of the general purpose register with an index specified by IR_2 into the main store. The same argument can be implemented in box 15 except the source and the destination of the data transfer are reversed.

Operate Instruction

The instruction code of an operate instruction is expanded to 25 bits: the leftmost 9 bits of IR_0 and the whole 16 bits of IR_1. The bits in IR_0 specify that it is an operate instruction (so not all of them are

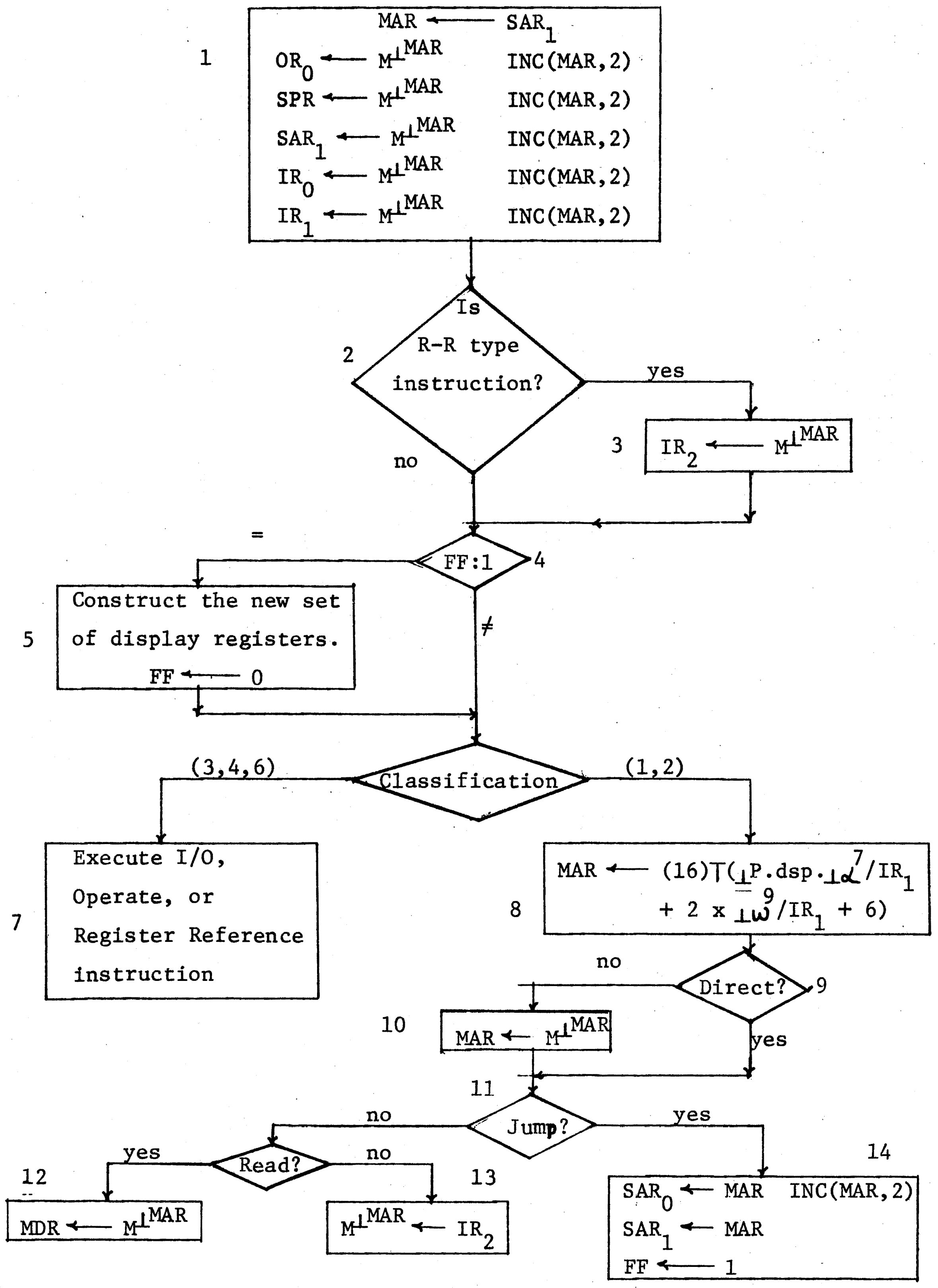

1
MAR ← SAR$_1$
OR$_0$ ← M$^{\perp MAR}$ INC(MAR,2)
SPR ← M$^{\perp MAR}$ INC(MAR,2)
SAR$_1$ ← M$^{\perp MAR}$ INC(MAR,2)
IR$_0$ ← M$^{\perp MAR}$ INC(MAR,2)
IR$_1$ ← M$^{\perp MAR}$ INC(MAR,2)
2
Is R-R type instruction?
yes
no
3
IR$_2$ ← M$^{\perp MAR}$
=
FF:1
4
≠
5
Construct the new set of display registers.
FF ← 0
(3,4,6)
Classification
(1,2)
7
Execute I/O, Operate, or Register Reference instruction
8
MAR ← (16)⊤($\perp$P.dsp.$\perp\overline{\alpha}^7$/IR$_1$ + 2 x $\perp\omega^9$/IR$_1$ + 6)
no
Direct?
9
10
MAR ← M$^{\perp MAR}$
yes
11
Jump?
no
yes
Read?
yes
no
12
MDR ← M$^{\perp MAR}$
13
M$^{\perp MAR}$ ← IR$_2$
14
SAR$_0$ ← MAR INC(MAR,2)
SAR$_1$ ← MAR
FF ← 1

Figure 35. Overall Instruction Control Diagram

used), the bits in IR_1 specify what operation must be performed. Sixteen bits can specify 2^{16} or over 64,000 operate instructions. It is not only difficult for the programmer to memorize that many instructions, but also is very hard for the hardware designer to build such a machine. Instead, we divide these 16 bits into several, say n, groups and each group takes an event time (8) to finish the operation specified by that group. In CM_1, each operate instruction consists of n operations executed in sequence. If three groups are used, 6 bits describe each of the first two event times, and 3 bits describe the third event time, the coding is tabulated in Table V. For example, if an operate instruction has its code (in hexadecimal) in IR_1 is

2 4 0 2

would cause ACC logical shift left and then complemented, if the resulting ACC is less than zero, the next instruction in normal sequence would be skipped.

Of course, there are other approaches for building this table as well, and any combination of bits which do not result in a logical conflict may be specified for the three event times.

Contour Control Instruction

In CM_1, there are several instructions associated with the virtual processors that should be included in the service routines (operating system). Since they are so concerned with the characteristics of the contour model and connected to some of the working registers in CPU, those instructions should be brought out here although they have nothing to do with the users. Also, the two instructions EXIT and ENTER are so

TABLE V

CODING OF OPERATE INSTRUCTION

Bit(s)		
0	0	if shift or rotate is specified, the shift or rotate is left
	1	shift/rotate right
		First Event Time
1	0	No Op
	1	arithmetic shift ACC
2	0	No Op
	1	logical shift ACC
3	0	No Op
	1	rotate ACC
4,5,6	000	No Op
	001	clear ACC
	010	complement ACC
	011	increase ACC
	100	set link
	101	clear link
	110	set FF
	111	clear FF
		Second Event Time
7,8	00	No Op
	01	increase MAR by the value specified by bits 9 to 12
	10	deposit the contents of the GPR with an index specified by bits 9 to 12 to ACC
	11	deposit the contents of ACC to the GPR with an index specified by bits 9 to 12
9,10,11,12		specify the value of increment to MAR or the index of GPR
		Third Event Time
13,14,15	000	No Op
	001	skip if $\perp$ACC=0
	010	skip if $\perp$ACC<0
	011	skip if $\perp$ACC>0
	100	skip if FF=0
	101	skip if FF=1
	110	skip if link=0
	111	skip if link=1

important during the execution of a program by CM_1, it is necessary to discuss them in detail: how can they be performed by the control unit.

Each virtual processor must be in one of the four states: awake, asleep, terminated, or invalid. The address of a valid virtual processor can be retrieved from the virtual processor table. A virtual processor can be created through two steps: allocation and initialization by the system service routine. Since each virtual processor occupies a fixed memory location (18 bytes excluding the display registers and the general purpose registers), CM_1 reserves a fixed storage area in which a linked list of nodes is organized, and each node contains 20 bytes: 18 bytes plus a 2-byte forward link which points to the next available node. A 16-bit register "TOP" always points to the first available node. When a virtual processor is allocated from the memory location pointed by "TOP," "TOP" is modified by the appropriate forward link. The technique to detect the underflow is necessary. During the execution of a program, only two kinds of data need be allocated. They are virtual processors and record contours. The instruction to achieve this goal is ALLOC followed by the data code. The effect of executing such an instruction is to allocate the memory locations needed by the data type specified by the data code and return the address of that first memory location to the pointer register. The initialization of a virtual processor can be accomplished through the initialize-processor instruction. Its format is INIT followed by the state code. The effect of executing this instruction is to initialize an invalid virtual processor which is located from the memory location pointed by PR_0 (the hardware pointer register, see Chapter II) to be either asleep or awake fully depend on the state code. Also, the site of activity and the label register of the newly initialized virtual processor are copies of the label register (LR) of the executing central processing unit.

In the snapshot 4 of example 2 in Chapter I, the two virtual processors P_1 and P_2 were allocated and initialized by P_0, the real situation happened to that snapshot described in AHPL, if the executing processor pair was (P_0,CPU_0) at that time, should be:

```
    SAR_0 ←—— 11;
    LR_0 ←—— SUB(M'(F)).ep;
    LR_1 ←—— SUB(M'(F)).ip;
    ALLOC 110;
    INIT 11;
    ALLOC 110;
    INIT 11;
```

where first ALLOC instruction is to allocate P_1, while the second ALLOC instruction is the allocate P_2.

During the execution of multiprogramming or coroutine, the states of some virtual processors need to be modified from awake to asleep or vice versa. When a virtual processor finishes executing a process, its state is changed to be terminated immediately and its reference count is decreased by one. A terminated virtual processor will not be deallocated until the next garbage collection. In this way, a terminated virtual processor can be reinitialized to be either awake or asleep for executing a new process later. When a new processor pair (VP_i,CPU_j) is created, if the state of VP_i is either terminated or asleep originally, its state has to be changed to be awake through the wake-processor instruction. Its format is WAKE followed by the state code. The effect of executing this instruction is to change the state of the target virtual processor, which is pointed by PR_0 of the executing processor pair, to be awake. If the target virtual processor is terminated first, three more things must be done:

(1) The allocation of its general purpose registers is necessary[1]

(2) Its site of activity is a copy of the executing processor's label register LR, and

(3) Its reference count is increased by one.

After this instruction, the state of the executing virtual processor is changed to be a state specified by the state code. Again use the snapshot 4 of Example 2 as an example, if there was a terminated virtual processor in the virtual processor table when P_1 and P_2 are to be allocated, the system service routine would get the address of that terminated virtual processor from the virtual processor table and store it into the PR_0 of the executing processor pair, then through the WAKE instruction to wake P_1. The details described in AHPL are as follows:

---

[1]The display register will not be constructed until the first instruction of the process to be executed by the new processor pair (VP_i,CPU_j) is fetched.

P_1.sta $\longleftarrow 11_2$;

P_1.soa.ep $\longleftarrow LR_0$;

P_1.soa.ip $\longleftarrow LR_1$;

P_1.lab.ep $\longleftarrow SAR_0$

P_1.lab.ip $\longleftarrow SAR_1$

P_1.gpr.ptr $\longleftarrow$ ADDR (the first byte of 256 bits allocated for P_1's 16 GPRs).

If there was another terminated virtual processor in the virtual processor table available for the initialization of P_2, the same procedure as mentioned above would be performed except the state code was 10 which would cause P_0 to be asleep after P_2 became awake. If there was not, then the instruction ALLOC 110 followed by INIT 11 would create an awake P_2.

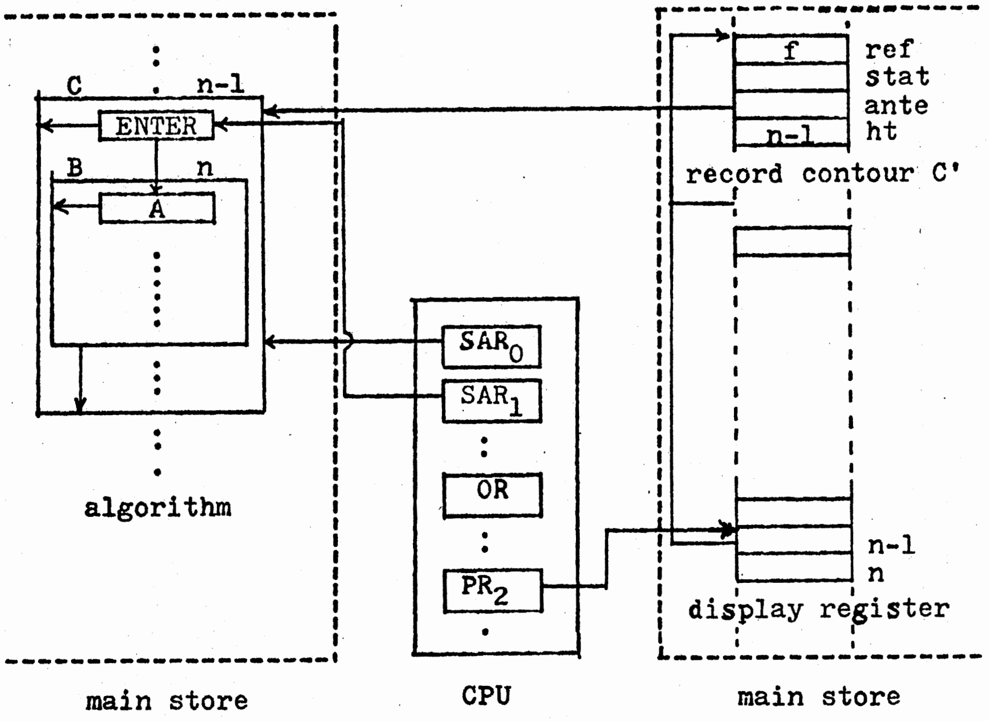
If the CPU_j of an active processor pair (VP_i,CPU_j) is asked to combine with some other virtual processor to form a new processor pair, the state of VP_i is changed to be asleep through the sleep processor instruction. Its format is SLEEP. The effect of executing such instruction is to store CPU_j into VP_i and change VP_i's state into sleep. Its details were discussed in Chapter III.

During the execution of a process, if that process is canceled for some reason, the executing virtual processor's state has to be changed through the terminate processor instruction. Its format is TERM. The effect of executing such instruction is to change the state of the target virtual processor to terminated. The target virtual processor is pointed by PR_0, and its reference count is decreased by one.

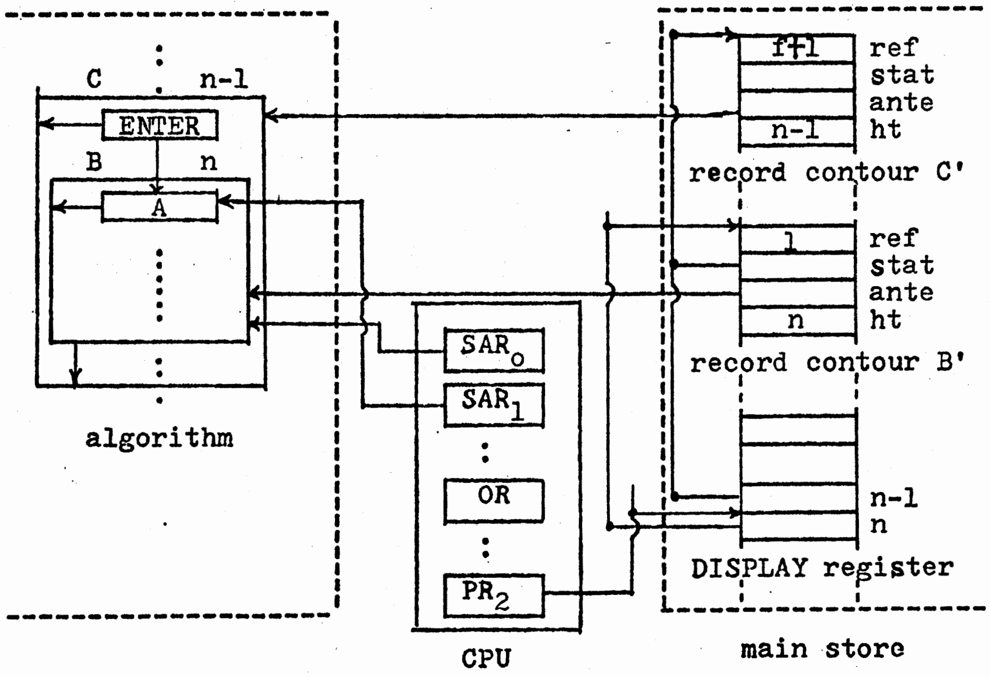
The algorithm contours of a program are allocated at assembly time and remain invariant during the life of that program. Its record contours are allocated or deallocated at runtime when ENTER or EXIT instructions are executed. When an ENTER instruction is being executed, the copy of an algorithm contour with its static link identical to that of the ENTER instruction is made. A newly allocated record contour has its reference count equal to one, which is the consequence of the pointer in PR_0. As mentioned earlier, the reference count of a record contour is maintained equal to the number of pointers point to that record contour. However, those pointers do not include those environment pointers of labels residing in that record contour, but include those environment pointers of labels residing in other record contours. That is, when a record contour is deallocated, the reference counts of those record contours to which pointed by the environment pointers of labels residing in that deallocating record contour must be decreased. The situations when an ENTER or an EXIT instruction is being executed is drawn in Figure 36, with the assumption that procedure B contains no labels. The corresponding control sequencers are shown in Figure 37.

Summary

There are four basic types of data in the contour model, each has its own storage organization, how they can be retrieved in a proper way is a problem. CM_1 solves this by always loading the first two bytes of the target data into the organization register OR, decoding the data code (first three bits of OR), and initiating proper service routine to finish accessing that data. The storage organization and the data code were shown in Figure 3. A successful instruction sequencing, as shown

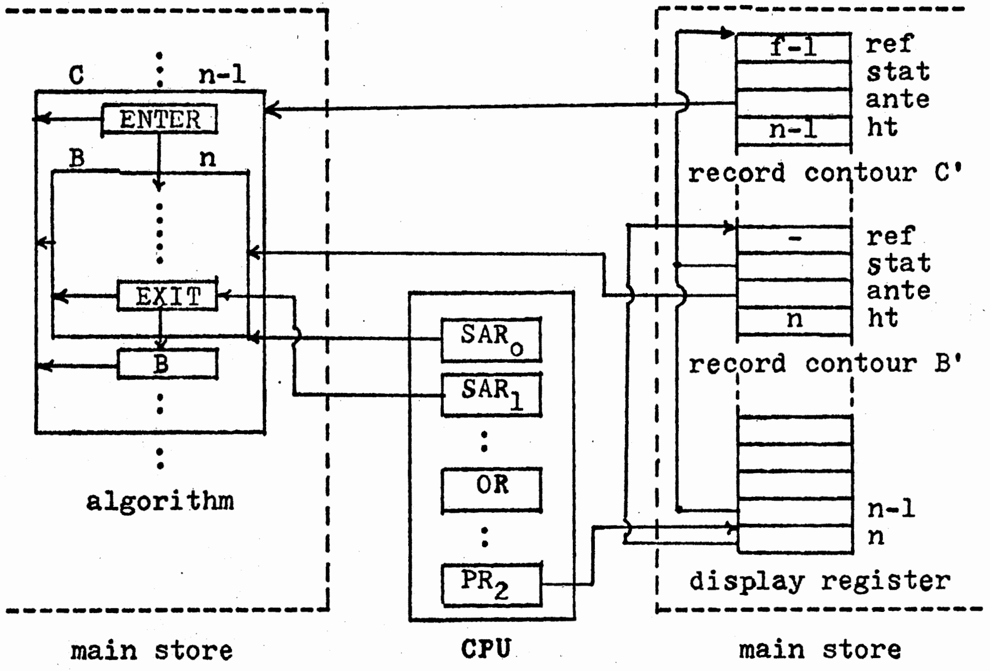


a). Just Before Entering B

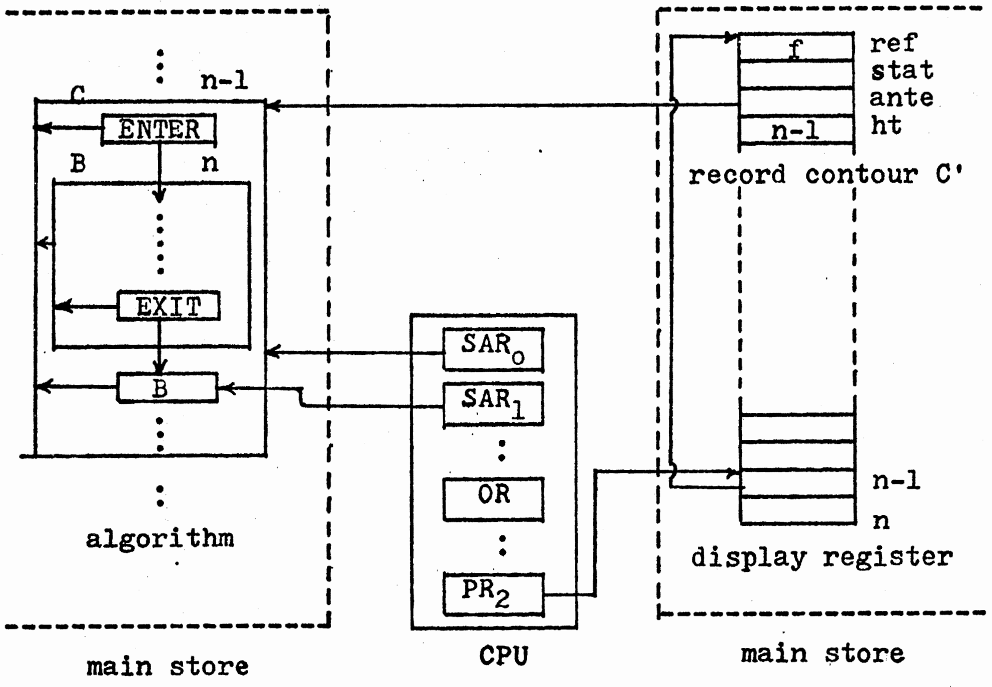


b). Just After Entering B

Figure 36. ENTER and EXIT

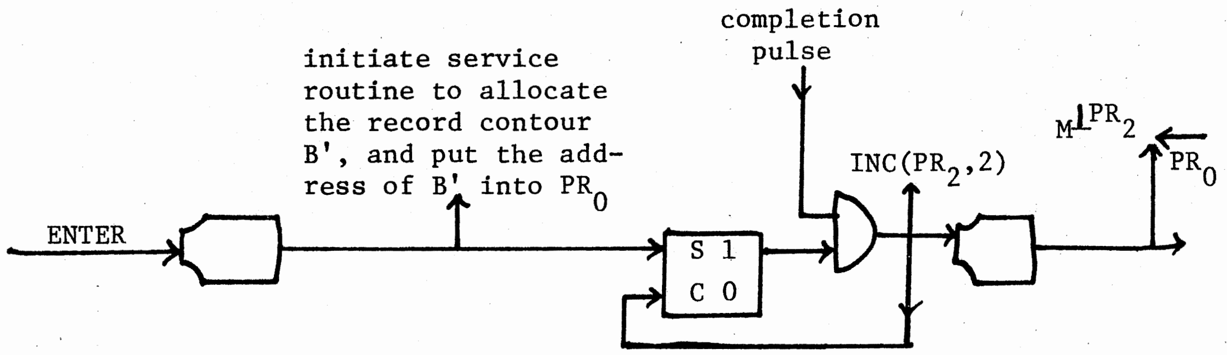


c). Just Before Exiting B

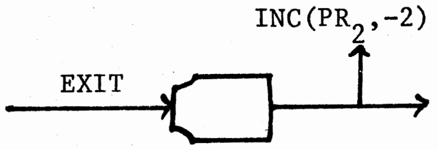


d). Just After Exiting B

Figure 36. (Continued)


initiate service routine to allocate the record contour B', and put the address of B' into PR_0
completion pulse
ENTER
S 1
C 0
$INC(PR_2,2)$
$M\downarrow PR_2 \leftarrow PR_0$

a. ENTER B



b. EXIT B

Figure 37. Control Sequencer of ENTER and EXIT

in Figure 35, needs to use the information supported by its format and its tag. That is, after the first two bytes of an instruction are loaded into the organization register OR, as shown in box 1 of Figure 35, the second and the third three bits contain the format and the tag separately, by decoding the format, the job done in box 2 of Figure 35, then how many bytes must be fetched from the main store to the instruction register becomes known, and by decoding the tag, the job done in box 9 of Figure 35, then the effective address of its operand can be found. Recall the operand of a CMAL instruction can only be an integer, a label, or a pointer.

During the execution of an ENTER instruction, a copy of some algorithm contour is made. The problem is which algorithm contour will be chosen. This can be solved by putting the address of the algorithm contour to be copied into the operand field of the ENTER instruction at assembly time. Of course, its tag must specify the operand is a pointer. In this way, the template of the record contour to be allocated can be found through hardware support.

CHAPTER V

FUTURE WORK

The block diagram shown in the Appendix shows the logical structure of CM_1. It is left hypothetical in this paper for neither the contour model assembler language nor its assembler has been developed (10, 11), and each of them does affect the hardware design of CM_1. In this paper, we have presented how the basic types of data of the contour model can be generated, stored, and retrieved; how the algorithm contours and record contours of a program can be allocated and deallocated; how to implement a processor pair to handle the multiprogramming, coroutines, and reentrant procedures; how CM_1 passes the procedure's name to the called procedure as actual parameter; how CM_1 performs the recursive programs; and finally, how its control unit fetches, executes, sequences a set of instructions of a program as it is designed. Those topics enable CM_1 to make some hardware implementation. But, there is still lots of work to do before CM_1 comes out to be a practical machine.

As pointed out in Chapter I, the contour model can be directly applicable to ALGOL 60. With respect to other high level programming languages, there is need for some modifications to the contour model before they can become formal programming languages for CM_1. This does not mean CM_1, after being modified somehow, cannot be applicable to most of them. That is, as long as those programming languages which can be analyzed by using BEGIN-END pair or equivalent symbols to identify each procedure or block can be implemented by the contour model. This important requirement allows CM_1 to tell when or where algorithm contours should be created at assembly time.

The four basic types of data which can be implemented in the contour model are: integers, pointers, labels, and instructions, any other type of data than these four cannot be handled by the contour model. In order to remove this restriction on the attributes of data, the efforts to develop suitable techniques are necessary. Recall that there is a 3-bit type field and several unused bits reserved for future use are contained in each data's storage area. If more types of data are allowed in CM_1, more bits are needed to code them. This can be done by expanding the type field to the length needed without changing the rest of the storage organization of each type of data.

In this paper, CM_1 is presented as a traditional machine. How to manipulate the microprocessor in the contour model and what its microprogramming language looks like are important work to do in the future. It does not make too much sense to design CM_1 by implementing a microprocessor at the time when even the CM_1, designed in a traditional way, has to be left hypothetical due to insufficient information such as the system service routines (i.e., the operating system) and the trade-off between the software and the hardware.

In order to increase the flexibility of the design in the future, many things, such as the number of central processing units in CM_1 and the unused bits in the storage organization of each type of data, are left variant. Only those working registers having connection with the execution of those instructions associated with the storage organization, and perhaps the hardware support of the system service routines mentioned in this paper are presented. Finally, the design in this paper is on the basis of hardware necessities, not from the economic point of view.

BIBLIOGRAPHY

1. Abd-alla, Abd-elfattah M. and A. C. Meltzer. Principles of Digital Computer Design. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1976.

2. Foster, Caxton C. Computer Architecture. New Yrok: Van Nostrand Reinhold, 1970.

3. Gries, David. Compiler Construction for Digital Computers. New York: Wiley, 1971.

4. Gschwind, Hans W. and Edward J. McCluskey. Design of Digital Computers. New York: Springer Verlag, 1967.

5. Hedrick, G. E. "An Adaptation of the Contour Model as a Run-Time Environment for ALGOL 68 Particular Programs." SOKEN KIYO, Vol. VI, 3 (1977), pp. 19-26.

6. Hedrick, G. E. and B. R. Alexander. "COROUTINE Programming in FORTRAN." The Australian Computer Journal, Vol. IV, 2 (1972), pp. 73-78.

7. Hewlett Packard Company. A Pocket Guide to the Hewlett-Packard 2100 Computer. Palo Alto, California: Hewlett-Packard, 1972.

8. Hill, F. J. and G. R. Peterson. Digital System: Hardware Organization and Design. New York: Wiley, 1973.

9. Horowitz, Ellis and Sarataj Sahni. Fundamentals of Data Structures. Woodland Hills, California: Computer Science Press, 1976.

10. Johnston, J. B. "The Contour Model of Block Structured Processes." SIGPLAN Notices, Vol. VI, 2 (1971), pp. 55-82.

11. Johnston, J. B. "What is to Become a Monograph of Contour Model." (Unpublished paper being written, March, 1977.) Las Cruces, New Mexico: New Mexico State University, Computer Science Department, 1977.

12. Knuth, D. E. The Art of Computer Programming, Vol. I: Fundamental Algorithms. Berkeley, California: Addison Wesley Publishing Co., 1968.

13. Knuth, D. E. *The Art of Computer Programming, Vol. III: Sorting and Searching*. Addison Wesley Publishing Co., 1975.

14. Passen, B. J. *Introduction to IBM System/360 Assembler Language Programming*. Wm. C. Brown Company Publishers, 1973.

APPENDIX

FUNCTIONAL BLOCK DIAGRAM

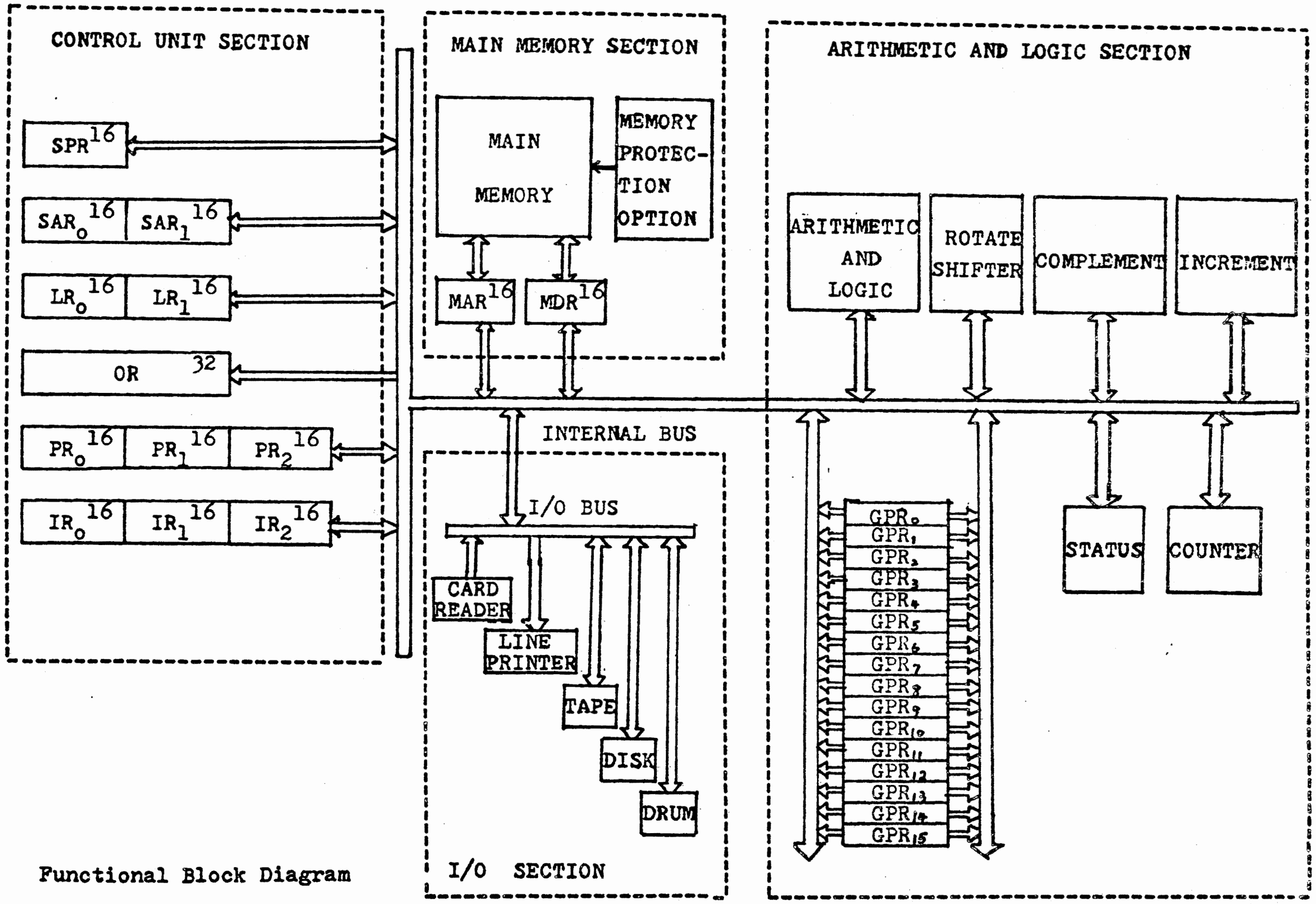


Figure 38. Functional Block Diagram of CM_1

VITA

Pu-Koung Philip Tu

Candidate for the Degree of

Master of Science

Thesis: TOWARD A HARDWARE IMPLEMENTATION OF THE CONTOUR MODEL

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in the province of Canton, China, July 26, 1949, the son of Chi-Lian and Lee-Ing Tu.

Education: Graduated from Chen-Kou High School, Taipei, Taiwan, China, in May, 1968; received Bachelor of Engineering degree in Electronic Engineering from Chung Yuan Christian College of Science and Engineering in May, 1973; completed requirements for Master of Science degree at Oklahoma State University in May, 1978.

Professional Experience: Teaching assistant and laboratory instructor, Chung Yuan Christian College of Science and Engineering, Taiwan, China, from November, 1973 to January, 1975; graduate teaching assistant, Department of Mathematics, Oklahoma State University, from September, 1977 to May, 1978.