

50310

2.11.12

1469



# PRELIMINARY DESIGN OF A LINKED LIST COMPUTER

Caxton C. Foster

## I. Introduction

Several symbol manipulating languages have already been written in both interpreter and compiler forms. Among these are IPL-V, COMMIT, and LISP; each with many attractive features. IPL-V, in particular, is concerned with the use of a linked-list structure.

### A. Nature of a List Structure

As an example of a "list structure," consider the following:

#### Things to do today:

- Shopping
- Clean house
- Pack for trip
- Pay bills
- Wash clothes

Each of these may or may not require an explanatory "sub-list." On a separate page I might have written:

#### Shopping:

- Groceries
- Meat
- Gas station
- Present for Aunt Jane
- Dry cleaning

The note to buy a present for my Aunt may be sufficient in itself, but

it is likely that I will have noted down on yet another page:

Groceries:

Fruit

Vegetables

Potatoes

Onions

Ice Cream

Soap for washing machine

The number of lists is usually not specifiable in advance, nor is the number of items on a given list. Furthermore, one may wish to add items as they are thought of or to delete them after they are accomplished. Note also that the order of the entries may be important, for I must go shopping before I wash the clothes, since one of the items to be purchased at the grocery is soap for the washing machine.

B. Linked-Lists

As Newel, Simon, and Shaw have pointed out, the decision to allot a fixed number of storage locations to each list is both wasteful and restrictive. It is wasteful in that many of the lists will not use up their allotment, and restrictive because some lists will require "special treatment" if they exceed the allotment. This "special treatment" might consist of a "jump" to a new location, at which the list is continued. Newel, Simon, and Shaw's solution to this problem is to make the exception the rule, and in each memory cell write an entry on the list and where to find the next entry. This address of the "successor" is called the "link." In the cell, which is named in the link of the first entry, will be found the second entry and a link to the third entry. This, in effect, reduces the allotment to one, thus eliminating

the waste of unused but allotted cells.

In addition, the use of linked lists facilitates insertion and deletion of entries. Suppose one wishes to insert an item between the second and third entries on a list. The process is simple. Obtain an unused cell in memory. Write the address of this cell in the link portion of the second entry. Write the desired addition to the list in this new cell and as its link, write the address where the third entry is to be found, (the "old" link of the second entry; see Fig. 1).

### C. Hardware Implementation of a Linked-List

The fact that perhaps one-third of each word is used up "merely" to specify a successor may appear somewhat profligate. There should be, it would seem, some less expensive method of performing this linking function in hardware. If a cell does not specify its successor, then the successor must be specified by some ordering of the information in the memory. For example, the next entry will be found in the cell with an address one greater than that of the present cell. To perform this ordering, either the cells containing the data must be moved about physically (difficult, if the cores are wired into planes), or else the data must be transferred to new cells when a change in a list is made. One way to accomplish this might be to make the entire memory a pushdown stack that could be "pushed" at any point, in order to open up a gap for the insertion of an entry. Ignoring for the moment the problem of keeping track of where one's data has been moved to (after several insertions and deletions), we may still note that a pushdown stack or bi-directional shift register probably requires three cores per bit. Thus,  $n$  bits of information require  $3n$  cores for storage in a pushdown stack, as opposed to  $n + \log_2$  (memory size) cores for a linked-list. Numerous other

schemes exist for organizing a list structure, but the linked-list method appears to be among the best and will be employed in this paper.

#### D. Advantages of Linked-Lists

There are three major factors which make linked-lists valuable:

1. The lack of rigid boundaries to lists enable them to grow as required; some short, some long, without advance specification of limits. This is flexibility.
2. The ease with which items can be added to or removed from lists. This is modifiability.
3. The ability to define procedures in terms of themselves. This is recursiveness.

It should not be assumed that these properties are peculiar to linked-lists. There are many other ways that memories could be organized that would produce these behaviors, but they are all easy to achieve with linked-lists.

There is an hypothesis current in linguistics for which credit is given to Benjamin Lee Whorf, to the effect that, "Language modifies culture," that is, those concepts for which there is a name or ready handle will be easier to use than those it takes a paragraph to specify each time they are referred to. The concepts that are easy to use will be the ones that are used. This is usually called the "Whorfian Hypothesis" and, while of great intuitive appeal, is almost impossible to operationalize.

Whether Whorf is right or wrong, it will be clear to anyone who has programmed that higher order languages greatly facilitate the writing of programs. As Shaw et al (1958) have said, "The feasibility of synthesizing complex processes hinges on the feasibility of writing programs of the complexity needed to specify these processes for a computer. Hence, a limit is imposed by the limit of complexity that the human programmer can handle. The measure of this complexity is not absolute, for it depends on the pro-

gramming language he uses." This paper, then, will be concerned with a Language for Linked-Lists and a computer designed so that the basic statements of Triple-L are machine operations of high speed.

## II. Design of a Linked-List Computer (LLC)

### A. Specifications

After the preceding discussion, we are in a position to write down some of the things an LLC must be able to do "gracefully."

1. An LLC must handle the linking process swiftly and without any intervention by the programmer. He should, of course, be permitted to look at the link of a cell when he so desires; but under normal conditions, linking should be automatic. Thus, adding to the head or tail ends of a list or removal from either, the change or deletion of an entry or the insertion of an addition at a specified place on a list should be rapid and simple.
2. There should be several rapid access cells of a pushdown nature between which data may be shuffled at a high rate for sorting or other processing.
3. There must be automatic housekeeping of discarded memory locations that become available for other uses after an old list is erased.
4. Indirect addressing must be available, for often we will not know what operand we want but only where to find it, or even just where its address is stored.
5. "Off street parking" should be provided for use when two programs alternately call on each other.
6. Input and output operations should be buffered and an automatic interrupt for handling the I/O buffer over-and under-flow should be included.
7. All high speed registers must be addressable and part word operations on them made simple.

### B. Machine Organization

#### 1. Overview

An explanation of the design of this computer will be simpler if

reference is made to Figure 2.

There are six major sections of the machine, three of which - the core memory, the control circuitry, and the arithmetic unit, are quite similar in design to standard von Neuman machines.

The memory is a word organized, 45 bits per word, 32 words per plane, 1024 plane, core memory with external selection and a 3.5  $\mu$ sec half-cycle (read or write) time. Reading is destructive. There is an additional 32-bit word on each plane which contains the "occupancy bits" of the other words and is readable with separate external selection circuitry (see AVSPA Control).

The memory is accessed under control of a 15-bit Memory Address Register (MAR) and all transfers take place via a 45-bit distributor.

The Arithmetic Unit performs both fixed and floating point operations with a 29-bit plus sign mantissa and a 14-bit plus sign exponent.

The operation of the control circuitry and the Instruction Register will be examined in more detail in the section on instruction codes. The remaining three major sections, being somewhat novel, will be examined in more detail below.

## 2. Available Space List (AVSPA Control)

### a. The problem:

One of the most frequent operations in any list modification will be the acquisition of an unused cell for an addition to a list, or the disposal of a cell no longer needed in a list. IPL-V has handled this problem by linking together all the unused cells in memory on a list called the "Available Space List." If one wishes to know the "name" of an unused cell, something like the following steps must be carried out:

- 1) Read the head cell of "available space" and store its

contents (the name of the cell we are to use) in the Memory Address Register (MAR).

2) Read the memory under control of the MAR.

3) Store the contents of the cell just read in the head cell of "available space." (This updates the available space list for the next acquisition cycle.) The name of the cell acquired is in the MAR.

This requires at least one memory access time in step 2, and three access times if the head of the available space list is not a high speed register.

Disposal of a surplus cell requires much the same sort of operations:

1) Read the head cell of available space.

2) Store this information in the surplus cell.

3) Store the name of the surplus cell in the head cell of the available space list.

Again, we find either one or perhaps three memory accesses. In our linked-list computer, we must consider the mechanization of the available space list most carefully. Since the acquisition or disposal of a cell will occur with great frequency, it is a process which should be of the highest speed possible. If one keeps track of available space in a linked-list as IPL-V does, then, even if the head of this list is kept in a high-speed register, the additional memory access cycle required to update the list for every transfer operation will almost halve the effective operating speed. It seems reasonable to assume that if, by some other method of keeping track of the unused cells, we can double the effective speed of memory with only a moderate increase in cost, we will be well justified.

b. The proposed solution:

To each word in memory, we shall add one additional bit, to be called the "busy bit." Since the memory is to be conventional in every



other way, reading is destructive and this bit will be set to "zero" whenever the word is read out. When and if the information is returned to storage, writing into a cell will automatically set its busy bit to "one."

Assuming a word-organized memory with 32 words per plane, it will be convenient to read out the busy bits of all the words in a plane as a unit to a high speed 32-bit shift register. The information read out in this step must be re-written at once, since it would not do to mark these words as "unbusy" indiscriminately. The AVSPA control unit will consist of a 32-bit 5-megacycle shift register, a 15-stage divide-by-two chain, and assorted gates, sense amplifiers, and selection circuitry, as shown in Fig. 3.

Under control of the upper ten bits of the divide-by-two chain, the busy bits of a memory plane are selected and stored in the shift register. The selection circuitry and sense amplifiers used for this operation are not the same as those used for normal memory access and so may be used concurrently, provided only that one set is not trying to read from the same plane as the other is.

Immediately, the information has been entered in the shift register, a 5-megacycle clock signal begins to shift left until the first "zero" appears in the "sensing flipflop." A count of the number of shifts is kept in the low order bits of the divide-by-two chain. Once a "free" word has been detected (busy bit equal to "zero"), shifting is inhibited and AVSPA control waits for a request for a free cell. Since the upper 10 bits of the divide-by-two chain specify the core plane containing the "free" word and the lower 5 bits specify a count indicating its position in the plane, the divide-by-two chain holds the address of a "free" word.

When a request for a free cell comes along, gates are opened which permit transfer of the information in the divide-by-two chain onto the main bus. After

a short delay to permit the transfer to take place, shifting and counting is re-initiated and proceeds until either another "zero" is found or until the count reaches 32, causing a transfer into the high-order portion of the divide-by-two chain. This transfer pulse sets a flip-flop which, if the memory is quiescent, initiates the selection of a new set of busy bits from the next higher numbered core plane. If memory is busy and the 10 high order bits of the MAR and the divide-by-two chain are the same, access is delayed until it is possible to read without interference. Selection of the busy bits of a plane will, in turn, inhibit normal access to that plane for a full read-and-write cycle.

Let us examine in some detail the behavior of this method of obtaining unused cells. To begin with, there will be many requests for unused cells during the execution of a program, so that the use of averages will be justified. If a fraction  $f$  of the memory is not occupied by program or data (free), we expect that on the average, there will be  $32f$  zeros in any "busy bit word." We may expect to find one zero about every  $1/f$  bits, so we have an average shifting time of  $1/5(1/f)$   $\mu$ sec if a 5 mc shift rate is used. If the half-cycle time of the memory is 3.5  $\mu$ sec., then the average elapsed time until a new empty cell is found will be  $T = (1/5f) + 7.0/32f$   $\mu$ sec for shifting and reading.

Now, after obtaining a free cell, the program will most probably write something into it and will then read and rewrite a new instruction. This will require three half-cycles so that, if as much as 4% of the memory is free, approximately one-half of the time AVSPA control will have found a new cell already and be waiting for another request. The other half of the time a new cell will not yet have been found, and, since for lack of evidence to the contrary we may assume the free cells to be randomly dis-

tributed throughout the memory, we effectively start the search over again and must wait another  $1\frac{1}{2}$  cycles or about  $10\ \mu\text{sec}$ . Thus, the average AVSPA delay time for two successive discoveries will be:  $T = \frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 10 = 5\ \mu\text{sec}$  if 4% of the memory is idle. If one other operation intervenes, then this time it will be cut in half to  $2.5\ \mu\text{sec}$ . If  $n$  other operations intervene, then the expected delay time will be  $T = \frac{5}{2^n}\ \mu\text{sec}$ . The worst case will arise when a list is being duplicated by a program. But still, the mean access time for successive discoveries of unused cells is better than would be obtained with a regular linked-list.

It is of interest to examine the case when only one free cell remains in the computer. On the average, AVSPA control will have to search half of the  $1024$  planes before finding it. This gives an access time of:  $(7\ \mu\text{sec per word} + 6.4\ \mu\text{sec count time}) \times 512 = 6.88\ \text{msec}$ . Somewhat before this stage is reached, it will probably be of advantage to move some part of the contents of core onto tapes or other backing store.

Using this scheme for the control of available space has the further advantage that, when a memory cell is discarded after being read out of, it automatically becomes available to AVSPA control without any further processing, and there is no possibility of tying available space in a knot by erasing the same cell twice.

To insure against endless hunting for a free cell when none is available, let us add a flip-flop to store the output of the high order bit of the divide-by-two chain. This flip-flop will be reset by any "zero" that is found. If, however, a second set pulse appears before reset, this indicates that no free cells have been found in any part of memory and a halt is executed and a "memory overflow" light turned on.

### 3. Hysteretic Pushdown Stacks

In order to avoid having two (or more) address instructions, all programmed transfer of information uses a "common" cell. In IPL-V, this "communication cell" is a linked-list called H-0 and is in all respects similar to any other list. The use of a genuine high-speed (1  $\mu$ sec) pushdown stack, rather than a linked-list, will reduce the access time from 7  $\mu$ sec to 1  $\mu$ sec for each reference to H-0, and that will be just about every instruction.

Fig. 4 shows one possible configuration using three cores per bit and a three-phase shift cycle that shifts right for the sequence ABC and left for the sequence ACB; but many other arrangements are possible to perform the same function. Shifting speeds of perhaps one  $\mu$ sec per full (3-stage) shift should be possible. Relative simplicity of wiring should hold the cost down to, say, \$2.00 per bit.

Studies conducted during the design of the KDF.9 computer at English Electric indicate that for a conventional computer, a pushdown accumulator of 16 cells should suffice for most needs. It is not immediately obvious that this result will apply to a linked-list computer, but even if it should, the restriction of this list, of all lists, to a maximum length would throw an unacceptable burden on the programmer. Yet, clearly, at \$2.00 per bit, we must keep the length of the pushdown stack as short as is reasonably, since this mode of storage costs about four times as much as main memory storage. The solution adopted here is to make the pushdown stack "reasonably" long and handle overflows with other hardware.

The pushdown stack is divided into two sections that can "push" or "pop" independently, as well as in synchrony. In addition to the 30 bits required to store the word, an additional "occupancy" or "busy bit" is

added to each word, making a total of 31 bits per word (see Fig. 5).

a. Drain cycle

As information is added to the stack, old information is pushed down until the occupancy bit of the bottom word comes on. At this point, an interrupt occurs and normal operation halts for a short time. The symbol in the bottom cell is transferred to the I and S portions of the distributor. At the same time, the address stored in another cell called "O-link" is transferred to the L portion of the distributor. The address of an unused cell is obtained from AVSPA control and is loaded into the MAR and into "O-link."

A write operation is initiated and the lower half of the stack is pushed down to clear out the bottommost word. If the occupancy bit of the bottom word is still "one," the operation is repeated. If, at the end of a cycle, the occupancy bit of the bottom cell is zero, the interrupt is terminated and the regular program continues.

b. Refill cycle

As words are read out of the top of the pushdown stack in the normal course of the program, a time will come when the words that were stored away above will be required. At this time, the pushdown stack will be empty and the occupancy bit of the top word will be zero. Again, an interrupt occurs. The contents of the "O-link" register specifies the location of the last word stored away so this address is sent to the MAR and a read operation is started. When the read cycle is complete, the contents of the I and S portions of the distributor are loaded into the bottom word of the top half of the pushdown stack and that half of the stack is popped. The L-portion of the distributor contents is transferred

to O-link and to the MAR. If the occupancy bit of the topmost word is still zero, another read cycle is initiated. If it is one, the interrupt is terminated.

c. Rationale

Since data is not available concerning the way in which words are added to and removed from the communication cell, and since it seems clear that, for most programs, words put into H-O will eventually be removed, let us assume that the loading and unloading of H-O is a random walk with symmetric probabilities. Actually, the situation will be better than this in all likelihood because of the high proportion of "in and out" operations which use the communication cell only for moving things about.

Feller (1950) considers random walks in detail and a careful study of optimum length for such a push-down stack should be made before any componentry is assembled. Assuming an 8-cell stack (4 cells per half), there will, on the average, be several operations (load or unload) before a random walk makes a 4-unit excursion causing a "drain" or "fill" cycle to begin. At this point, the walk may be considered to start over.

Using Feller's equation of 3.5 on page 287, we find that the expected number of references to the communication cell after a drain or fill operation has taken place and before it is necessary to drain or fill again will be:

$$Dz = Z(a-z) = 4(8-4) = 16$$

for equal probability of entering or withdrawing. These 16 operations do not include replacement (overwriting) or copying out, but only pushdown or pop-up operations.

Each drain of fill requires four half-cycle memory access times or 14  $\mu$ sec. Thus, the average access time for push or pop of a high speed stack will be:

$$\overline{t}_a = 14/16 = .875 \mu\text{sec.}$$

Such a technique might also be applied to the B-5000 or the Kdf-9 with profit using, probably, a reserved section of main memory and a pointer, rather than a linked list.

In our Linked-List Computer, we propose 19 pushdown stacks:

- 1 communication cell - 8 deep (30 + 1 bits)
  - 1 instruction register link - 8 deep (15 + 1 bits)
  - 7 working cells - 8 deep apiece (30 + 1 bits)
  - 1 I/O buffer - 30 deep (30 + 1 bits)
  - 9 attics - 4 deep (30 + 1 bits)
- Total number of bits involved here is 4158.

#### 4. Working Cells

While most symbol transfer will occur via the communication cell, there will be many times when high speed "working lists" will be useful for sorting or collecting items. Furthermore, once a collection of items has been entered on a working list, it would be desirable to be able to treat that list directly, without having to transfer it into the communication cell.

In the COMMIT language, there is one "working space" (our CC) and 128 "shelves" (our working cells). Data can be transferred back and forth between working space and any shelf and in addition, it is possible to exchange the entire contents of a shelf and the working space in one machine operation. This is accomplished by exchanging the "names", not the data.

Since our communication cell and our working cells are all identical except for their names, it seemed desirable to add an 8-register crossbar-type switch which would permit "name exchange" at high speed. This is instruction no. 117, which appears to the programmer to exchange the contents of the two units. We provide only 7 "shelves" and not the 128 provided by COMMIT.

### 5. The Attics

If the operation of two programs is interlocked such that, for example, program A finds "eligible cells" on some list, and program B performs some operation on the cells found by A, it is convenient in IPL-V to consider program A to be a "generator," in that it generates names for B to manipulate. There are primitives (J-processes) defined which make generator "set up" and "put away" simple to program.

To execute this type of function, we have included an "off-street parking facility" called the "attic," where information pertinent to a generator (or any program at a level higher than the current operating program) may leave partially processed operations for later recall. There will be an attic over each working cell, over H-0, and over the instruction register link.<sup>1</sup>

The attics will, at least in the first design, be high speed pushdown stacks linked into main memory but of a depth of only 4 units and not 8. This measure of economy is indicated by the assumption that they will be

1. It is not clear, without more experience with this machine, whether an attic over L(HI) is required. It seems as if it might be replaced by relatively simple usage of the regular link stack, various branching instructions, and the "looping" instruction.



used less often than the "regular" storage. Their design will be hysteretic and "drain" and "fill" operations will involve only two cells at a time. Their effective access time will be, therefore:

$$\bar{T}_a = 3.5 \times 2/2(4-2) = 1.75 \text{ } \mu\text{sec.}$$

In a sense,

these attics are just another group of working cells, but the nature of the proposed design allows parallel transfer of information from the first to the second group in one word time. When the contents of a working cell and the contents of (0) are exchanged using a 177 instruction, the contents of the respective Attics are also exchanged.

#### 6. I/O Buffer

The operation of the I/O buffer differs from that of the other pushdown stacks in some ways. It may be viewed as a "tube", one end of which is permanently attached to the I/O selector and the other end of which is connected, when required, to the distributor. The I/O selector determines which peripheral unit is being operated and in which direction information is flowing. The I/O selector in turn is controlled by the word at the head of the I/O queue.

Let us suppose that we wish to output a list L-1 to tape unit 3. Assume that the tape is sitting at the end of record mark. Using instruction 33, we add the following word to the tail of the I/O queue (15):

P	Q	S
18	0	L-1

When this word reaches the head of the I/O queue, the first thing that happens is that peripheral unit 18 (tape unit 3) is interrogated to find out if it is busy. If it is busy, no further action takes place until it is free.

Once this happens, q is examined. Since  $q = 0$ , an output is called for and an interrupt of normal processing begins. The S part of the I/O-queue-head is transferred to the MAR and the I/O buffer is connected to the distributor. The contents of L-1 are read and the I and S portion, transferred to the I/O buffer, the link of L-1 is transferred to the MAR<sup>and to S(15)</sup> and another read cycle begins. This process continues until either the I/O buffer is full, or else a link of zero is discovered. In either event, the interrupt is terminated and normal programming resumes. Meanwhile, back at tape unit 3, a start signal has been received and it is beginning to roll. When writing speed has been achieved, the I/O buffer begins to drain its contents out onto the tape. At an appropriate time before the last words are drained out, S(15) is examined. If it is zero, the list was completely transferred to the buffer already and is popped to bring up the next I/O command. If S(15) is not zero, another interrupt is initiated and the I/O buffer refilled as required, beginning where it left off in the cell named S(15). The I/O buffer thus acts as a FIFO list coupling the peripheral units into the main computer. Note that output of a list removes it from main memory so if one wants it saved, he must make a copy.

The other I/O commands shown in II-7-d function in a similar fashion.

The computer is designed to accept up to 3 I/O buffers and queues: 15, 16, and 17. With more than one I/O unit, "queue discipline" (the order of operations) becomes more complicated and it will be possible to inadvertently attempt to output a list before it is input. It will be the programmer's responsibility to keep track, perhaps on a list somewhere, of what is where.

## 7. Word Format

Because they had to fit it inside an existing computer, the authors of IPL-V were limited as to the word format they could choose. We, however, are free to choose as we see fit. There are three sections to a word in Triple-L: the instruction, the symbol, and the link to the next word. If there are 32K words in main store, then the link must have 15 bits if it is to be capable of pointing to any one of them. By a similar argument, the symbol must also be 15 bits long. The instruction portion of a word must allow for  $2^7 = 128$  instructions and  $2^3 = 8$  possible modes of "indirect address." In addition to these 10 bits, we must store information about the kind of symbol (local, regional, etc.) the word contains and perhaps if it has been processed or not. It seems convenient to allow 5 bits for these markers, bringing the total for the I-portion up to 15, equal to the other two. Thus, our words will consist of three equal fields of 15 bits each, plus one "busy bit" to denote occupancy of the memory cell. Fig. 7 shows three possible formats.

### a. Instructions

If a cell contains an instruction when its contents are loaded into the IR, the first field will be treated in three sections:

- m - marker bits
- p - process to be carried out
- q - "quality" of indirectness

The other two fields are treated in units as the SYMBOL to be processed and the LINK to the next instruction.

#### i. Marker bits

There are five marker bits in each instruction labeled  $m_1, m_2, m_3, m_4, m_5$ . They are interpreted as follows:

- $m_1$  - sign - if 0 - word is positive  
                  if 1 - word is negative
- $m_2$  - local - if 0 - SYMBOL is a regional or an internal symbol  
                  if 1 - SYMBOL is local and is the name of a sublist  
                          of the list which contains this cell
- $m_3$  - responsible - if 0 - SYMBOL is to be treated merely as  
                          an entry on this list  
                  if 1 - SYMBOL names a cell or list whose  
                          contents are to be erased when and  
                          if this cell is erased.
- $m_4$  - processed - if 0 - not processed  
                  if 1 - already processed
- $m_5$  - terminal cell - if 0 - not terminal  
                  if 1 - is terminal

ii. Process code

There are 128 possible instructions. They are divided into 16 groups of 8 instructions each and a detailed description of each one may be found in the appendix.

iii. Quality bits

The quality bits determine how to find the effective symbol on which the processing should be performed.

- q
- 0 -- the effective symbol may be found in the S portion of the IR.  
      It is (S) itself.
- 1 -- S(IR) holds the address of where the effective symbol may be found.
- 2 -- S(IR) holds the name of a cell which holds the address of the  
      effective symbol (indirect).
- 3 -- S(IR) holds the location of a cell which holds the name of  
      another cell which holds the address of the effective symbol  
      (second-order indirect).
- 4 -- Use the q and s of the word stored in the cell named in S(IR).
- 5 }  
6 } Not assigned, may be used for monitor or compiler or programmer.  
7 }

b. Data

All numeric data is stored in floating point format, as a 29-bit plus sign integer mantissa in the I and S portions of a word and a 14-bit plus sign exponent. Genuine integers will have an exponent of zero. Two groups of arithmetic operations are provided: Fixed point and floating point, as well as float and unfloat instructions. Integer mode operations will set the flag  $\ominus$  on overflow.

c. Head of list

The head cell of a list is reserved to store information about that list. In the I-portion of the word is stored the address of the last cell on the list - the tail. In the S-portion is stored the name of the description list (if any) and in the L-portion the address of the first cell.

d. I/O commands

Words on the I/O queue (lists 15, 16, or 17) have an interpretation which differs from those of normal instructions. The marker bits are ignored completely. The process bits specify which peripheral unit is involved in the transfer and the quality bits indicate the nature of the operation to be performed.

For an average machine installation, p might have the following interpretations:

- p
- 0 - console typewriter
- 1 - key-lights bank 1      always available
- 2 - key-lights bank 2      optional
- 3 - key-lights bank 3      optional

- 4 - key-lights bank 4      optional
- 5 - key-lights bank 5      optional
- 6 - on line printer
- 7 - card reader-punch
- 8-15 - up to 8 disc-units
- 16-31 - up to 16 tape units
- 32-127 - not assigned or remote keyboards or more tape or discs.

The actual configuration will, of course, depend on the desired applications and finances.

The quality bits have the following meaning:

- q
- 0 - output list S
- 1 - input list to S
- 2 - scan forward for S
- 3 - scan backward (tape) for S
- 4 - rewind tape
- 5 - proceed to end of record
- 6 - select disc track S
- 7 - select disc sector S

### III. Machine Language Operations

From the user's point of view, the most important aspect of a computer is its command code, be this hard or soft-ware implemented. The repertoire of available operations and the speed at which they are carried out will determine the usefulness of the machine.

From a compiler writer's or a machine designer's point of view, the machine

language or hardware commands are all important. The problem is not one of what commands are necessary and sufficient; "subtract the contents of x from the accumulator and store the result in x", together with a branch on negative accumulator, can play the Sheffer-strokes of the computing world. The problem is to design an "efficient" set of commands that, without excessive cost, can perform most of the operations one is likely to need, and at high speed. In general, the larger the hardware repertoire, the happier the compiler writer. But, we don't want to go overboard. "Branch if the contents of the accumulator is exactly 21" might be a very convenient instruction at times, but the relative merits of installing the required hardware compared with the cost of writing "subtract a cell holding 21, Branch on zero" seems to be all on one side.

Only experience can really decide if a command structure is a good one or not, and, unfortunately, by the time enough experience has been accumulated, the programmer is well acquainted with the tricky ways of getting around the designer's oversights and like a tree root penetrating a crack, is just properly shaped to fill the available space.

The command structure outlined in the appendix was arrived at in the following way:

1. An initial list of all the operations and the J processes of IPL-V and IPL-VI and a selected subset of the commands of COMIT was made up since these represented the considered thought of earlier authors.

2. Those items on the list which could be replaced by a simple concatenation of two or three other entries were removed, but those whose lack would require extensive "programming around" were retained, as well as those operations for which a significant improvement in speed could be obtained by suitable hardware.

3. Operations required to make the hardware design arrived at above more flexible were added.

4. Other operations that seemed "free" (utilized existing hardware in a slightly different manner) were added.

5. The resultant set was discussed with Prof. Norman Scott of the Electrical Engineering Department, and Dr. Philip Benkard of the Mental Health Research Institute, who has had considerable experience in IPL-V. Prof. Scott urged me, among other things, to expand the arithmetic and logical operations, and Dr. Benkard convinced me of the usefulness of "generators" in IPL-V, which resulted in the "Attics," and suggested the inclusion of group 20 - indirect symbol movement.

Thus, the credit belongs to others and the mistakes to the author alone. The reader will notice that several instructions and numbers are left blank. He is urged to make suggestions regarding their useful employment.



APPENDIX I

Command Structure

The following notation will be used:

- (0) or CC - the communication cell. I + S, pushdown stack.  
L-head of overflow list in core. T-end of list.
- (1)-(7) - working cells - I + S = pushdown stack. L-head of overflow  
list in core. T-end of list.
- (10) or IR - the instruction register. I + S control register.  
L-pushdown stack. T-head of overflow list in core.
- (11) - the distributor = ISL = 45 bits
- I(12) - the available space register (AVSPA) - 15 bits
- S(12) or COMP - the comparator - 15 bits
- L(12) or MAR - the MAR - memory address register - 15 bits
- (13) or  $\lambda$  - the lambda - the logical operation register.  
ISLT = 60 bits.
- (14) - the counter or accumulator. IS = mantissa L = exponent
- (15)-(17) - I/O buffer queue's I + S = head of queue L address of  
next command. T-address of last command.

When a part word operation is indicated, the relevant section will be designated as  $\phi(x)$ , meaning the contents of the  $\phi$  th part of register x.

$\phi$  may take on the following values:

- O(x) - whole word - often omitted as (x)
- m(x) - the contents of the marker bits of register x. If mis-  
subscripted (1-5), only the indicated bits are affected.
- P(x) - the contents of the process bits of register x
- q(x) - the contents of the quality bits of register x
- I(x) - the contents of the first 15 bits of x, (I = m + p + q)
- S(x) - the contents of the symbol or S-portion of register x

- L(x) - the contents of the link or L-portion of register x
- T(x) - the contents of the tail or T-portion of register x (for certain high speed registers only).
- S<sup>q</sup> - designates the "effective symbol."








Detailed Description of Instructions

(Lists end with a link of 77777. Any attempt to address this location will set flag ⊖ and terminate instruction.)

- 0 - Clear - set the bits of the  $\Phi$  part of high speed register n to zero.
- 1 - Copy part - the bits of the  $\Phi_2$  part of high speed register n<sub>2</sub> are forced equal to the bits of the  $\Phi_1$  part of n<sub>1</sub>.
- 2 - Exchange - the left end of  $\Phi_1(n_1)$  is connected to the right end of  $\Phi_2(n_2)$  and vice versa. A left shift of as many places as there are in  $\Phi_1$  is carried out.
- 3 - Find Eff. Symb. - clear I (11) to zero\*If q(10) = 0, copy S(10) into S(12), and S(11). Terminate. If q(10) ≠ 0, decrement q by 1. Copy S(10) into L(12) the MAR and Read and Rewrite. Copy S(11) into S(10). Return to \*.
- 4 - Write in S<sup>q</sup> - Find S<sup>q</sup> (as in 3). Copy into MAR. Copy (13) into (11). Pulse write wires.
- 5 - Read from S<sup>q</sup> - Find S<sup>q</sup>. Copy into MAR. Pulse read wires.
- 6 - Pushdown S<sup>q</sup> - Find S<sup>q</sup>. Copy S(12) into L(12) Read. Copy I(12) (AVSPA) into L(12) (MAR). [Note: This copy will be held up until AVSPA does have a cell available.] Write (information into new cell).\* Copy L(12) into L(11), Copy S(12) into L(12). Set responsibility bit (11) to zero. Write (info with new link back into old cell).
- \*7 - Pop up S<sup>q</sup> - Find S<sup>q</sup>. Copy S(12) into L(12) . Read. If L(11) is 77777, clear (I + S) (11) to zero. Set m<sub>3</sub>(11) (termination bit) to one. Rewrite. Set flag ⊖. Terminate. If L(11) ≠ 77777 copy (11) into (13). Copy L(11) into L(12). Read. Copy S(12) into L(12). Write. Test responsibility bit M (13); if zero, terminate. If one copy S(13) into S(0). Erase list structure [execute 4<sup>l</sup>].

Symbol Movement

- 10 - Copy S<sup>q</sup> into (0) - Find S<sup>q</sup>. Set responsibility bit of (11) to zero. Copy IS(11) into (0). (→)

- 11 - Copy  $S^q$  onto (0) - Find  $S^q$ . Set responsibility bit of (11) to zero. Push down (0). Copy IS(11) onto (0). (  )
- \*12 - Move  $S^q$  into (0). Find  $S^q$  \* Copy IS(11) into (0). Copy L(11) into L(12). Read. Copy S(12) into L(12). Write. (  )
- \*13 - Move  $S^q$  onto (0) - Same as 12 but push down (0) at \*. (  )
- 14 - Copy (0) into  $S^q$  - Find  $S^q$ . Copy S(12) into L(12). Read. Copy (0) into IS(11). Set responsibility bit of (11) to zero. Write. (  )
- 15 - Copy (0) onto  $S^q$  - Proceed as in no. 6 until \*, then copy (0) onto IS(11). Go on with no. 6. (  )
- \*16 - Move (0) into  $S^q$  - as in 14 but do not erase responsibility bit. Then pop-up (0). (  )
- \*17 - Move (0) onto  $S^q$  - As in 15, but do not erase responsibility bit. Then pop up (0). (  )

INDIRECT Symbol Movement

Note l(x) the cell named in cell x.

- 20 - Copy  $S^q$  into l(0) - Find  $S^q$ . Set  $m_4(11)$  to zero. Copy (11) into (13). Copy (0) into L(12). Read. Copy IS(13) into IS(11). Write.
- 21 - Copy  $S^q$  onto l(0) - Push down l(0), then proceed as in 20.
- \*22 - Move  $S^q$  into l(0) - Execute 20, then pop up  $S^q$ .
- \*23 - Move  $S^q$  onto l(0) - Pushdown l(0), then execute 22.
- 24 - Copy l(0) into  $S^q$  - as in 14 but use contents of cell named in (0) not contents of (0).
- 25 - Copy l(0) onto  $S^q$  - First pushdown  $S^q$ , then execute 24.
- 26 - Move l(0) into  $S^q$  - Execute 24, then pop up l(0).
- 27 - Move l(0) onto  $S^q$  - Pushdown  $S^q$ , then execute 26.

List Scanning

- \*30 - Scan list (0) for  $S^q$  - Find  $S^q$ . Copy S(0) into L(12) (MAR) and S(13). Read and rewrite. Copy L(11) into L(13) and into L(12). \*Read and rewrite. Shift (13) left 15 places.

Copy L(11) into L(13) and L(12). Compare S(11) with S(12). If equal, terminate. If not equal, test termination bit of (11). If 0, return to \*. If 1, copy S(13) into L(12). Read. Copy I(13) into L(12). Read. Set L(11) = 77777. Rewrite. Set flag  $\ominus$ , terminate.

Note: No. 30 leaves the address of the cell which precedes the cell containing  $S^q$  in I(13). The address of the cell containing  $S^q$  in S(13) and the link of the cell containing  $S^q$  in L(13). No. 30 also cleans up private termination cells left by an attempt to pop up the last cell of a list, if  $S^q$  is not on the list.

- \*31 - Address of "parent" of  $S^q$  on list (0) replaces (0) - Execute no. 30. Copy I(13) onto S(0). Clear I(0) to zero.
- \*32 - Address of  $S^q$  on list (0) replaces (0) - Execute no. 30. Copy S(13) onto S(0). Clear I(0) to zero.
- \*33 - Address of "son" of  $S^q$  on list (0) replaces (0) - Execute no. 30. Copy L(13) onto S(0). Clear I(0) to zero.
- \*34 - Value of attribute (0) of list  $S^q$  replaces (0) - Find  $S^q$ . Copy S(12) onto L(12). Read and rewrite. Copy S(0) into S(12). Copy S(11) [name of description list] into S(0). Execute no. 30.\* Copy L(13) into L(12). Read and rewrite. Copy IS(11) into IS(0).
- \*35 - Address of value of attribute (0) of list  $S^q$  replaces (0) - As in 34 until \*. Then copy L(13) into S(0). Set I(0) to zero.
- 36 - no op.
- 37 - no op.

#### List Modification

- 40 - Insert  $S^q$  before l(0) - Find  $S^q$ . Copy IS(11) into IS(13). Push down cell named in (0) as in no. 6 until \*. Then copy IS(13) into IS(11) and proceed with no. 6 from \*.
- 41 - Insert  $S^q$  after l(0) - Find  $S^q$ . Copy IS(11) into IS(13). Copy (0) into L(12). Read. Copy L(11) into L(13). Copy I(12) (new cell) into L(11). Write. Copy L(11) into L(12). Copy (13) into (11). Write.

[Note: if (0) contains the name of a list, no. 41 adds  $S^q$  to head of list].

42 - no op.

43 - Add  $S^q$  to tail of list (0) - Find  $S^q$ . Copy I(11) into T(13). Copy (0) into L(12). Read. Copy ISL(11) into ISL(13). Copy I(11) into L(12). Read. If termination bit of (11) is one; copy S(12) into S(11). Copy T(13) into I(11). Set L(11) to 77777. Write. Copy ISL(13) into (11). Move (0) to L(12). Write. Terminate. If termination bit of (11) is zero; Copy I(12) into L(11). Write. Copy L(11) into L(12) and I(13). Copy S(12) into S(11). Copy T(13) into I(11). Set L(11) = 77777. Write. Copy ISL(13) into (11). Move (0) into I(12). Write.

[Note: No. 43 assumes that the address of the tail of a list is in the I portion of the head of the list.]

44 - Push aside cell (0) - Copy (0) into L(12). Read. Copy IS(11) into IS(13). Clear I(11) to zero. Set m(11) to "local and responsible." Copy I(12) into S(11). Write. Clear responsible bit (n) to zero. Copy IS(11) into IS(0). Copy S(11) into L(12). Copy I(12) into I(11) and L(11). Set S(11) to 77777. Write. Copy I(11) into L(12). Copy IS(13) into IS(11). Set L(11) = 77777. Write.

[Note: No. 44 creates a new, describable local list and puts its name in (0) and into the cell named in (0). The former contents of the cell named in (0) become the first (and only) entry on the new list.]

45 - Erase list structure  $S^q$  - Find  $S^q$ . Clear counter (14) to zero. Copy S(12) into L(12) \*. Read. If S(11)  $\neq$  77777 copy S(11) onto (0). Augment (14) by one; then (or otherwise) \*\*. Test L(11). If = 77777 or L(11) = 0, decrement (14) by one. If (14) negative, terminate. If (14)  $\geq$  0, Copy S(0) into L(12), return to \*. If L(11)  $\neq$  77777 or 0, copy L(11) into L(12). Read. Test responsibility bit of (11). If equal to zero, return to \*\*. If equal to one, copy S(11) onto (0). Augment (14) by one. Return to \*\*.

[Note: No. 45 will erase a list and then all the lists mentioned (including the description list) for which there is responsibility. Any kind of list structure (re-entrant, knotted, threaded, tree, or branching) may be erased without worry using no. 45, since the test on L(11) at \*\* for zero prevents trying to erase an already erased cell.]

46 - Assign (0)' as value of attribute (0) of list  $S^q$  - execute no. 35 (find address of value),

then move (0) into S(12), then execute no. 16, 1, S(12).  
(Move (0) into cell named in S(12).

- 47 - Add (0)' as additional value of attribute (0) of list  $S^q$  - execute no. 34 (get name of value). Test if local symbol. If yes, execute no. 41. If no, execute no. 44 (create a list) and then no. 41.

#### Fixed Point Arithmetic

- 40 - Add  $1(S^q)$  - add contents of cell  $S^q$  to contents of IS(14).  
42 - Subtract  $1(S^q)$  - subtract contents of cell  $S^q$  from contents of IS(14).  
43 - Low multiply by  $1(S^q)$  - low order product of contents of cell  $S^q$  and (14).  
44 - Integer divide by  $1(S^q)$  - contents of IS(14) are divided by contents of cell  $S^q$ . Only integer part retained.  
45 - Copy counter into  $S^q$  - contents of IS(14) are stored in IS( $S^q$ ). L( $S^q$ ) is not disturbed.  
46 - Fractional divide by  $1(S^q)$  - IS(14) divided by contents of  $S^q$ . Remainder to IS(14).  
47 - Set flag  $\ominus$  if exponent zero - If L(14)  $\neq$  0, set flag  $\oplus$   
if L(14) = 0, set flag  $\ominus$

#### Floating Point Arithmetic

- 50 - Floating add  $1(S^q)$   
51 - Floating subtract  $1(S^q)$   
52 - Floating multiply by  $1(S^q)$   
53 - Float  $1(S^q)$  - contents of cell  $S^q$  copied into IS(14) and right shift of IS(14) with count in L(14) until rightmost bit of IS(14) is a one.  
54 - Floating divide by  $1(S^q)$   
55 - Ground  $1(S^q)$  - contents of cell  $S^q$  copied into ISL(14). Shift IS(14) and count until L(14) is zero.

56 - no op.

57 - no op.

Branch and Test

60 - Branch to  $S^q$  (do not save L(10))

61 - Execute  $S^q$  (save L(10) for return)

62 - no op.

63 - Branch to S if sense switch  $q$  is set

64 - Branch to S if  $q$ th part of (11) equals S(12)

65 - Execute S if I/O buffer queue  $q$  is busy

66 - Repeat execute  $S^q$  (0) times - Find  $S^q$ . Copy S(12) into L(12).  
Read. Copy S(11) into S(13). Copy L(11) onto L(10).  
Copy I(12) into S(11). Write. Copy S(11) into L(12).  
Copy IS(0) into IS(11). Copy S(13) into L(11). Write.

67. Continue at S if not finished - Exchange S(10) with L(12). Read.  
Copy S(11) into IS(14) decrement by one. If counter not  
zero, copy IS(14) into S(11). Write. Terminate.  
If counter is zero, copy L(11) into S(11), copy I(10)  
into I(11), Copy S(10) into L(12). Move L(10) into  
L(11). Write.

[Note: If no. 57 is in the head of a threaded list  
S (in cell S), then a 56, 0, S will cause the list S  
of instructions to be repeated (0) times. Since the  
loop count is kept on a push down list named in S(S),  
these instructions may be nested as desired.]

70 - Branch to  $S^q$  if flag  $\ominus$  . Do not save L.

71 - Execute  $S^q$  if flag  $\ominus$  , do save L.

72 - Set flag to S

73 - Complement flag.

74 - Preserve flag and set to S.

75 - Pop up flag.

76 - If counter  $\geq$  0, Branch to S.

77 - If counter = 0, branch to S.

Test and Set Marker

P

- 100 - Execute S if  $m_1(0) = 1$  data
- 101 - Execute S if  $m_2(0) = 1$  local
- 102 - Execute S if  $m_3(0) = 1$  responsible
- 103 - Execute S if  $m_4(0) = 1$  processed
- 104 - Execute S if  $m_5(0) = 1$  terminal
- 105 - Set  $m(0)$  to 0 whenever  $S_1-S_5$  has ones
- 106 - Set  $m(0)$  to 1 whenever  $S_1-S_5$  has ones
- 107 - Mark list structure  $S^q$  processed.

Stack Control

- 110 - Transfer q symbols from (0) to stack S
- 111 - Distribute q symbols from (0) to q stacks starting up from S
- 112 - Distribute q symbols from (0) to q stacks starting down from S
- 113 - Collect q symbols, one from each of q stacks starting up from S
- 114 - Transfer q symbols from stack S to (0)
- 115 - Pushdown q stacks starting up from S
- 116 - Pop-up q stacks starting up from S
- 117 - Exchange names of stack (0) and stack S

Logical Operations

- 120 - AND (0) and  $C(S^q)$  to (0)
- 121 - OR
- 122 - EXOR



- 123 - Complement
- 124 - Blend - insert bits of  $C(S^q)$  into (0) wherever  $\lambda$  has ones
- 125 - Cyclic shift left (end around) of (0) S places
- 126 - Left shift of (0) S places
- 127 - Right shift of (0) S places

Miscellaneous Operations

- 130 - Count available space in counter (takes about 10 msec).
- 131 - Push-up stacks into Attics.
- 132 - Pull down Attics into stacks.

References

1. Feller, W. Probability Theory, Vol. I (first edition), New York, Wiley, 1950.
2. Haley, A. C. D. The KDF.9 Computer System, Proceedings of the Fall Joint Computer Conference, 108-120, 1962.
3. Newell, A. (Editor). Information Processing Language-V Manual, New Jersey, Prentice-Hall, 1961.
4. Shaw, J. C., Newell, A., Simon, H. A., and Ellis, T. Q. A Command Structure for Complex Information Processing, Proceedings of the Western Joint Computer Conference, May, 1958.
5. Whorf, B. L. Language, Thought, and Reality, Technology Press, Cambridge, 1956.
6. Yagva, V. H. An Introduction to COMIT Programming, MIT, 1961.

Fig. 1

Insertion of an Item in a List.

The name "2576" is assumed to be in AVSPA as the next free cell.

OLD LIST

Machine Address	Item	Link
1000	MILK	1001
1001	BREAD	1002
1002	CHEESE	1003
etc.		

NEW LIST

Machine Address	Item	Link
1000	MILK	1001
1001	BREAD	2576
2576	BUTTER	1002
1002	CHEESE	1003
etc.		

FIG. 2

GENERAL ORGANIZATION

THE NUMBER IN PARENTHESIS INDICATES THE ADDRESS OF THE REGISTER

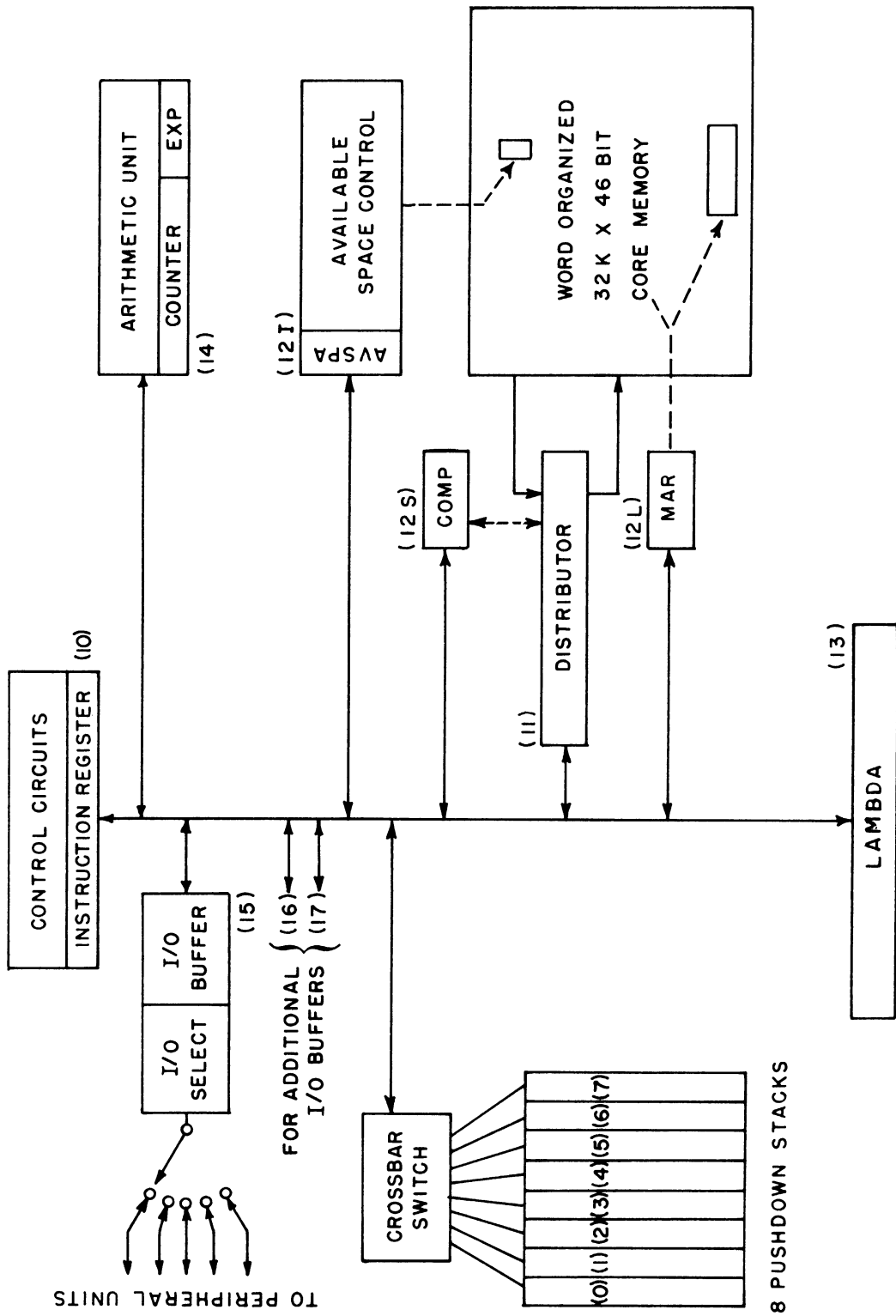


fig 3  
AVAILABLE SPACE CONTROL

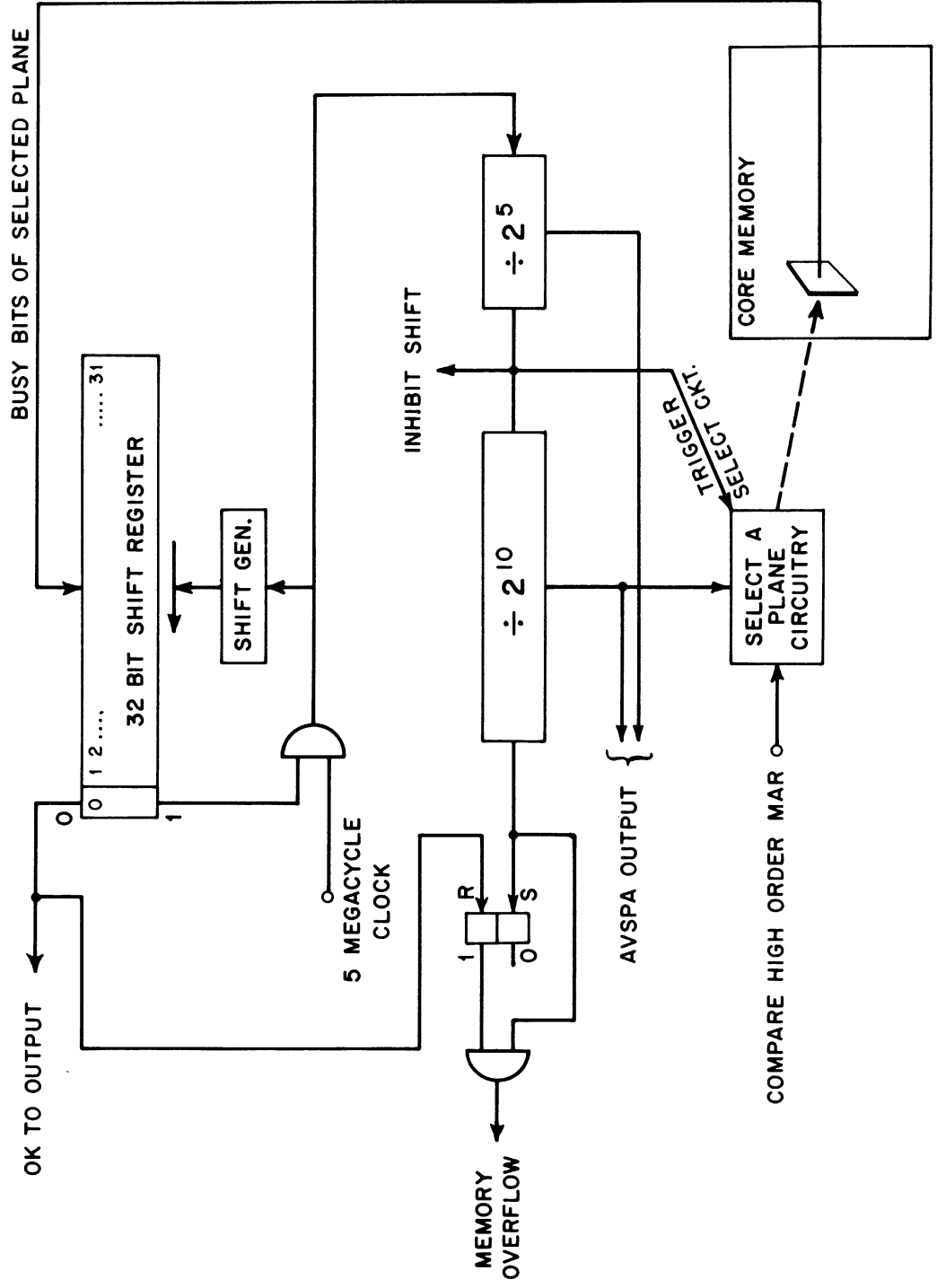
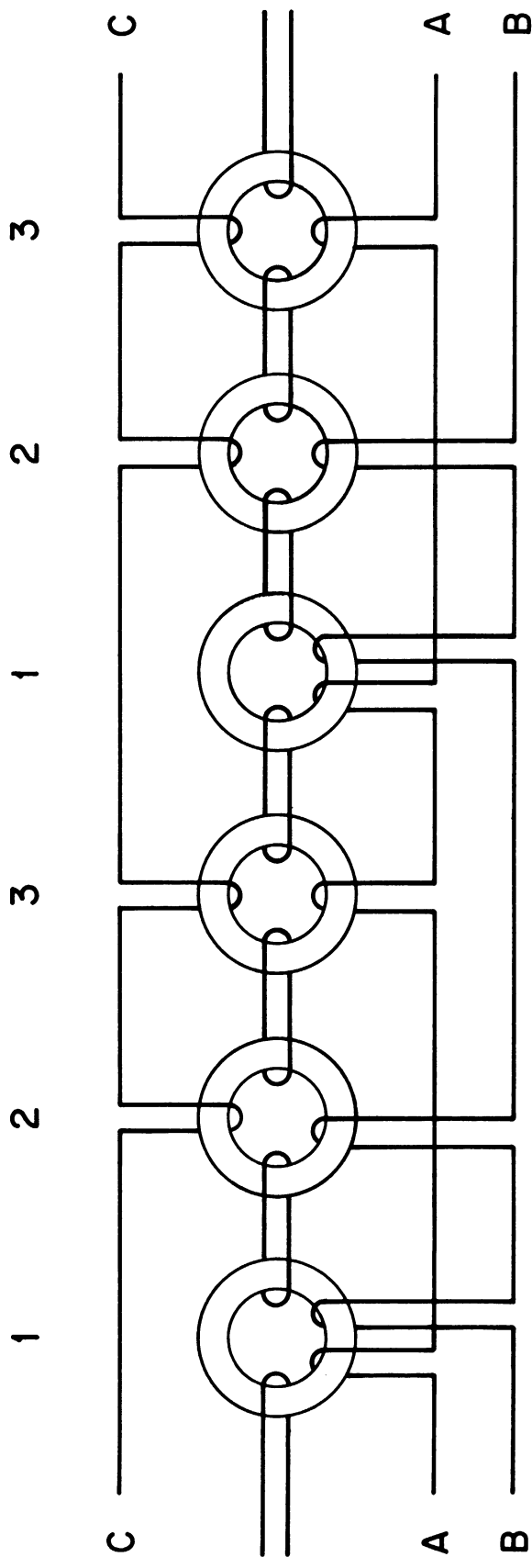


fig 4

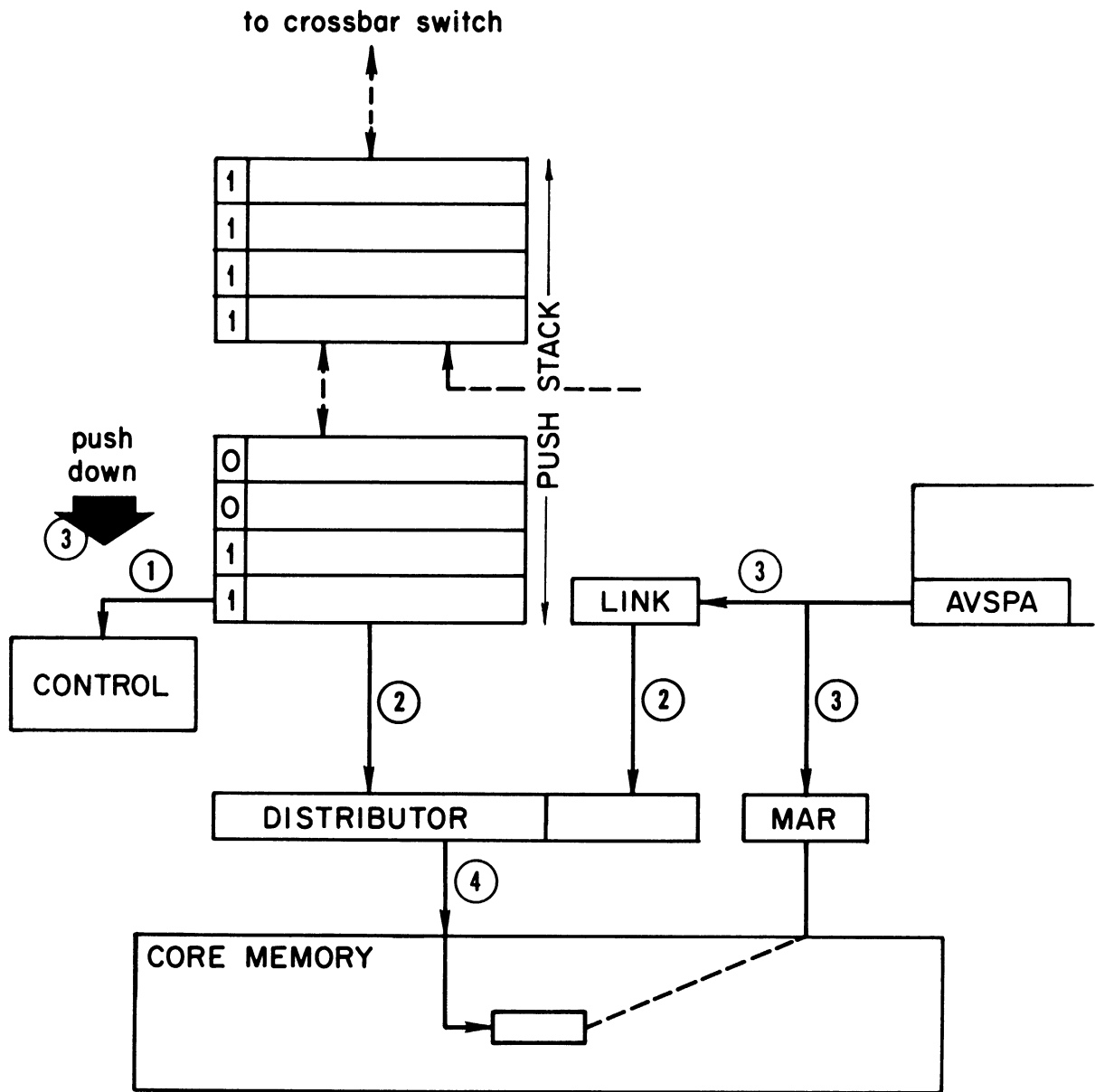
### SHIFT REGISTER



Cycle ABC causes shift of information 1 → 2 → 3 → 1 right.

Cycle BAC causes shift of information 1 → 2 → 3 → 1 left.

fig 5a  
DRAIN CYCLE

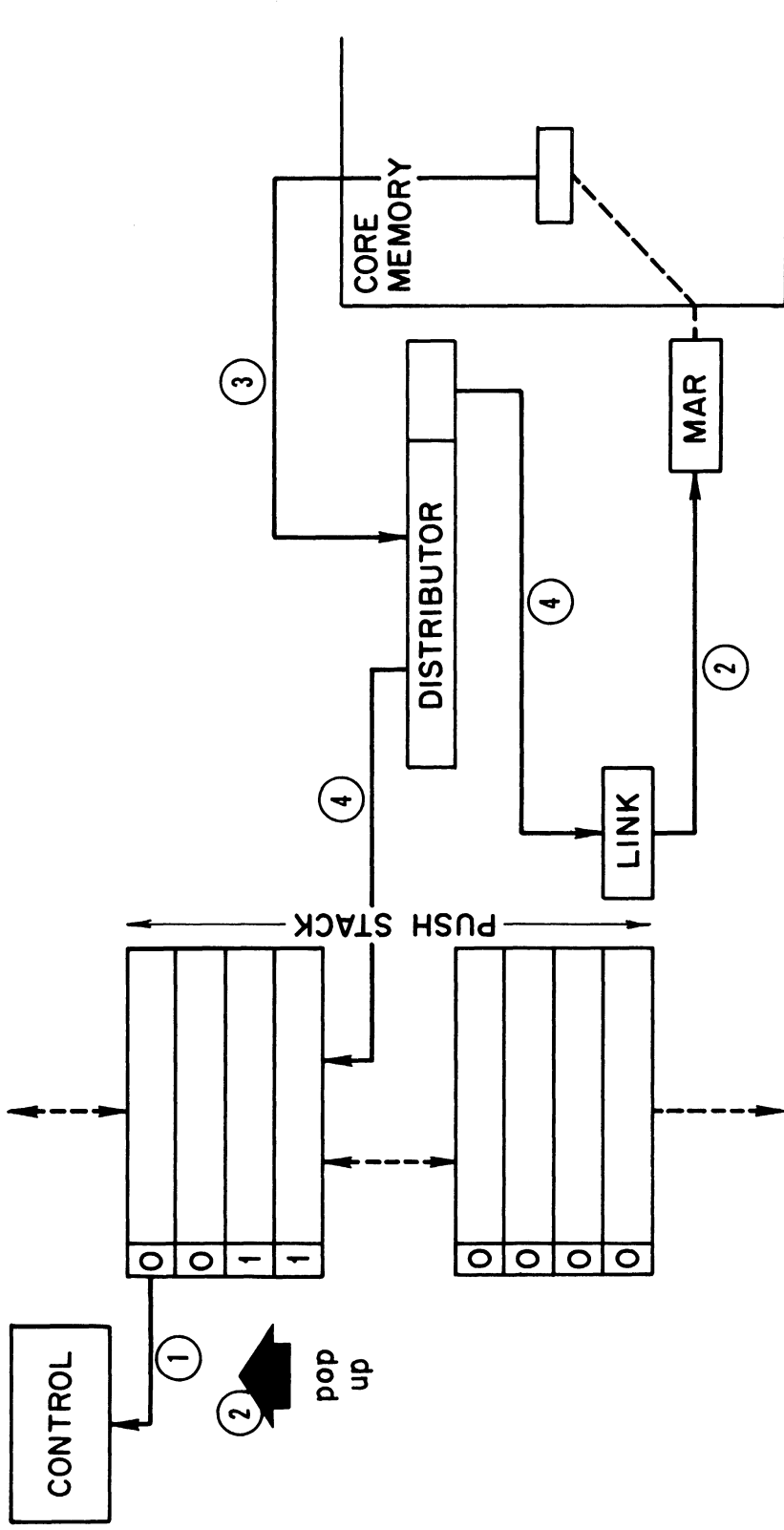


TIME PERIOD.

1. Test busy bit of bottom cell { if 1, interrupt go to 2  
if 0, release
2. Copy bottom cell and link into distributor.
3. Copy (AVSPA) into link and MAR push down bottom half stack.
4. Write.

fig 5b

FILL CYCLE



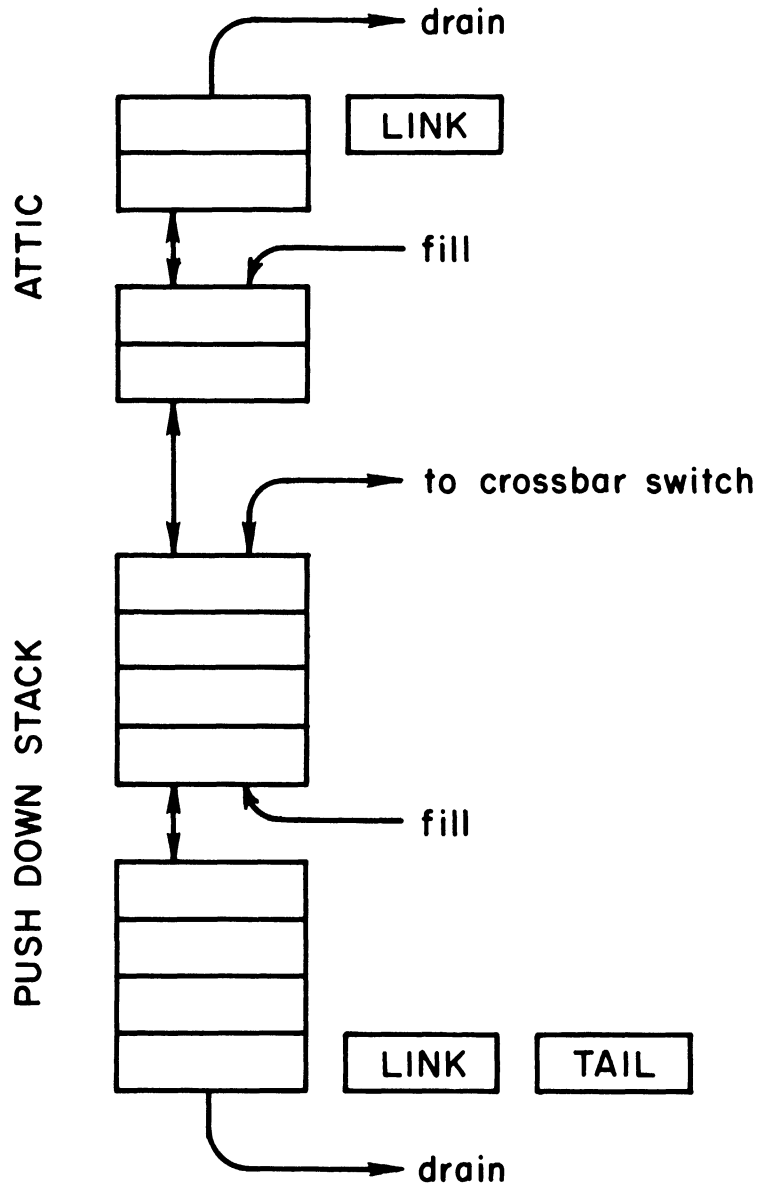
TIME PERIOD

1. Test busy bit of top cell { if 0, interrupt, go to 2  
if 1, release
2. Pop up top half stack and copy link into MAR.
3. READ
4. Copy distributor into bottom cell of top half and link.

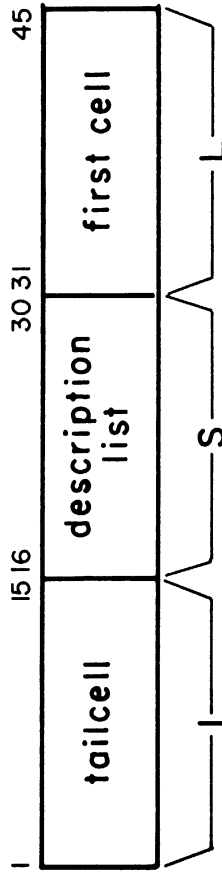
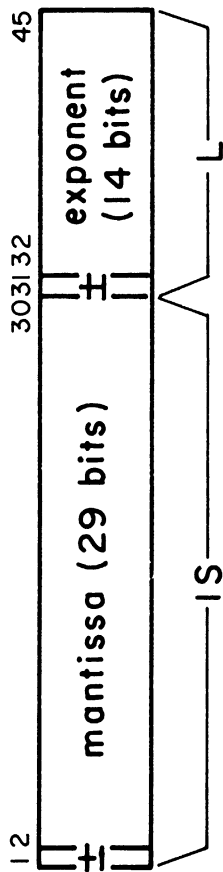
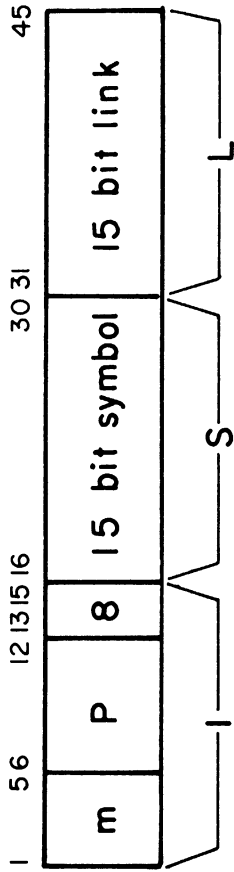


fig. 6

# ATTIC CONSTRUCTION



**Fig. 7**  
**Word Format**



UNIVERSITY OF MICHIGAN



3 9015 02826 5554