MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Project MAC

Machine Structures Group

Memo No. 6

Memorandum MAC-[illegible]

October [illegible], 19[illegible]

NESTING AND RECURSION OF PROCEDURES IN A SEGMENTED MEMORY

J. B. Dennis
E. C. Van Horn

In this note we discuss the hardware and programming mechanisms necessary to implement the generalized nesting of procedures, where a procedure at one level may call another procedure, which may call others, and so on -- perhaps invoking a call on itself (recursion). We start from the following set of assumptions.

1) Memory is addressed fundamentally by word names each consisting of a segment name and a word address. (See MAC [illegible] for an exposition of segmentation.)

2) A processor has a small finite number of operand registers for performing arithmetic and logical operations on quantities, and a small finite number of attachment registers that contain segment names for making the addressing of memory more efficient.

3) Procedure segments are in pure procedure form; that is their execution does not result in their modification.

4) We desire principles that will permit nesting to an arbitrary depth without alteration of conventions or technique.

[illegible]) Conventions are to be established so a procedure does not have to know at what level in the hierarchy it will be called or what procedure it might itself call (names of procedures might be supplied as parameters).

An Example

As an example of what can happen, suppose we are given the procedures Ff(), Gg(), and Hh(), having the structure shown in Figure 1. For simplicity we suppose that these procedures are not used as functions. That is, the names of the procedures do not appear in the left half of assignment statements within their bodies.

We restrict the parameters of a procedure to be any objects representable as a single word - e.g., a single quantity, or the word name of a word that is either single quantity, the entry point of a procedure or the word name of another word. A parameter used by one procedure in calling another may be one of the parameters by which it itself was called.

In our example, procedure Ff() has three parameters, α, β and γ. It contains a direct reference to α, references private variables p and q, and calls a three parameter procedure Gg() with the following actual parameters: (1) the parameter α, (2) the name q of a private variable, and (3) the parameter γ. Procedure Gg(α, β, γ) uses its parameter γ as the name of a procedure that it calls with the following actual parameters: (1) the parameter β, and (2) the parameter α. A third procedure Hh (α, β) calls β as a two parameter procedure with the actual parameters α and β. In our illustration of mechanisms for handling communication among procedures we consider the consequences of a call on Ff() with the following as actual parameters: (1) the name of the word Xx; (2) the name of the procedure Hh; and (3) the name of the procedure Hh.

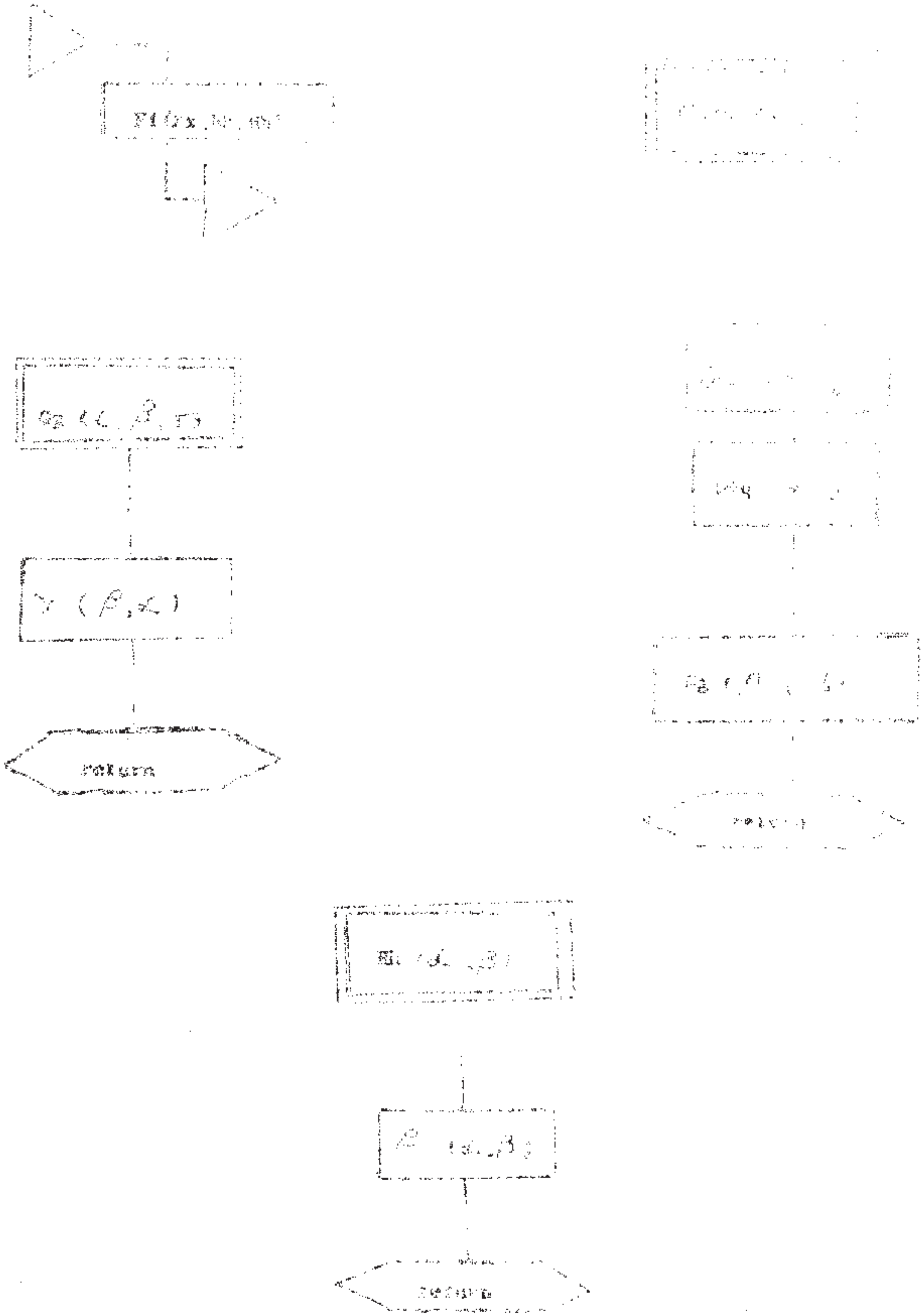

Figure 1 - An example of recursive process

Segmentation

To keep the discussion [illegible], we shall assume [illegible] procedures is coded into a subroutine which consists of a single pure procedure segment. We further suppose that another segment is used as a push down stack for holding return addresses and other data that is private to a particular call upon a procedure. The [illegible] is assumed to be in yet another segment.

The Stack Pointer

The push down stack must be private to the particular process executing the nested procedure. A stack pointer must be associated with the process to indicate which quantities in the stack are pertinent to the procedure whose instructions are currently being executed. We choose to have the stack pointer indicate the first quantity in the stack which is pertinent to the current procedure. Each time a new subroutine is entered, the stack pointer is increased to point to the first location in the stack that is not used by the calling procedure. Each time a subroutine returns, the stack pointer is restored to the value which it had immediately prior to that subroutine's call.

Conveyance of Parameter Information

In the remainder of this note, we distinguish between a subroutine's actual parameter information, and a subroutine's actual parameters themselves. Generally a subroutine must execute some sort of addressing algorithm in order to obtain its actual parameters. This addressing algorithm generally requires the subroutine to fetch certain intermediate linkage information in order to locate its actual parameters. The first of these linkage words that a subroutine examines which contains information unique to a particular parameter is said to contain the parameter information for that parameter.

We consider three methods by which parameter information may be conveyed betwee a calling pure procedure subroutine and a called pure procedure subroutine.

1) Stacking - The calling subroutine places parameter information in the stack segment following the word that the stack pointer will point to upon entry to the called subroutine.

2) Followers* - Parameter information is placed within the calling procedure segment immediately after the procedure call instruction.

3) Operand registers - Parameter information is placed in the operand registers of the processor by the calling subroutine.

It is possible to use one, two, or all of these methods during a given calculation. Our object here is not to describe an optimum method of parameter linkage, but to characterize some of the possible methods. We will therefore restrict our attention to schemes which rely solely on one of the three methods mentioned above.

The stacking technique is straightforward and well known from its use in interpreters for algebraic languages. The techniques of followers and operand registers are in general use in conventional machine language coding. To our knowledge, the provisions necessary for the general

---

* After A.A. Smith

[illegible] [illegible] of the follower technique to pure procedure coding [illegible] until now, been formulated.

Certain characteristics of the stacking and follower techniques are apparent without much consideration. If the parameter information is stacked, then all forms of information can be transmitted in [illegible] quantities modified by the operation of the procedure. Whereas if the information is imbedded in pure procedure coding, then the information cannot be so modified. In particular, the information in the followers cannot be segment names as these are discovered at execution time. On the other hand the stacking of parameter information requires the calling procedure to move information into the stack prior to the call. This leads to extra procedure steps that increase execution times and occupy memory. Followers offer the possibility of reducing the memory space required to transmit parameter information, provided a suitable addressing technique can be formulated.

The method of passing parameter information through operand registers is conceptually the simplest of the three methods. Each subroutine can be written to pick up information about its first parameter in a particular operand register, information about its second parameter in another particular operand register, etc. If a certain subroutine's particular linkage registers are being used to hold other quantities at the time when it is called, then the calling subroutine must save, in other operand registers or in the stack, the contents of these linkage registers before loading them with appropriate parameter information. After the called subroutine returns, the calling subroutine usually reloads these quantities from their former registers. Since the number of operand registers is generally on the order of 10, these save-restore operations occur quite frequently in a procedure base of moderate complexity. This fact makes the method of operand registers not much more efficient than the stacking method. Moreover, information about only a small number of parameters can be passed through operand registers.

In the remainder of this note we will examine the problems involved in the implementation of the stacking and follower techniques in the context of segmentation.

Addressing of Actual Parameters

Having listed three methods of transmitting parameter information, we now turn our attention to the nature of the information which is transmitted. We again consider, by coincidence, three addressing alternatives.

1) Immediate - the transmitted information is the parameter itself.
2) Direct - the transmitted information points to the parameter.
3) Indirect - the transmitted information points to a chain of indirect words which leads to the parameter.

In general, each of these three addressing techniques may be used with either the stacking or the follower transmission methods. However, immediate addressing and the follower method are incompatible in a pure procedure context, because under these circumstances the actual parameters of each procedure call cannot be changed by the calling procedure in accordance with its own actual parameters.

Addressing Segments Within a Procedure

We consider now a sub-problem of the problem of addressing actual parameters, namely the problem of enabling a procedure to access segments whose names are unknown to it prior to a call by another procedure.

This may be done by either of the following two methods

1) Name passing - Segment names are passed as parameters

2) Tag passing - Attachment tags are passed as parameters.

The name passing technique is perfectly general. In a procedure having n attachment registers, the tag passing technique is subject to the following restrictions.

1) The actual parameters of each procedure may reside in no more than n different segments.

2) In a computation involving more than n segments, subroutines may have to save and restore the contents of some of the attachment registers before calling other subroutines.

Summary

Figure 2 summarizes the various combinations of techniques which have been discussed thus far. The rest of this note will be devoted to detailed examples of three implementations of the procedure nest diagrammed in figure 1. These three implementations are: (1) stacking with direct addressing and name passing, (2) stacking with indirect addressing and tag passing, and (3) followers with direct addressing and name passing.

		Stacking	Followers	Operand Registers
Name Passing	Immediate		impossible for pure procedure	
	Direct	first detailed example	third detailed example	
	Indirect			
Tag Passing	Immediate		impossible for pure procedure	
	Direct			
	Indirect	second detailed example		

Figure 2: Summary of methods discussed

Operand Register Conventions

For generality we assume that each procedure is to be coded independently and without regard to the coding of the other procedures except according to well defined conventions. Conventions regarding the use of operand registers and attachment registers, therefore, must apply uniformly over all procedures that might be nested. The following conventions are made here and are examples of what is permissible.

1) Attachment register 0 contains the name of the stack segment.

2) Index register 0 contains the word address which is the stack pointer.

A further convention is made (although it is not essential as will be seen) regarding the use of processor registers for transmitting the return address (the word name of the procedure entry instruction) to the called procedure.

3) Attachment register 1 conveys the segment name of the calling procedure.

4) Index register 1 conveys the word address of the calling instruction in the calling procedure.

In the third detailed example, the following additional convention will be introduced.

5) Index register 3 conveys the word address which is the base stack address (defined below) of the calling procedure.

As with conventions 3) and 4), conventions 5) is not essential, but is introduced for convenience only

Addressing Notation

We use upper case letters to represent segment names as follows:

F, G, H - procedures F(), G(), and H()

S - push down stack

M - main procedure

X - segment containing the public variable x

Lower case letters such as f, g, x, p represent the word address of objects. The notation

A/b

represents the word name consisting of the segment name A and the word address b. The attachment tags are written as a0, a1, a2, ..., while x0, x1, x2, ..., represent the *content* of index registers. An *address word*

a1/b

consists of an attachment tag a1 and a word address b. The notation

{a1/b}

represents the word addressed by the address word. For convenience we will sometimes write <F> to mean the attachment tag of an attachment register containing the segment name F. We presume the existence of an instruction (or macro) of the form

attach S, a0

that causes the segment name S to be placed in attachment register a0 if S is valid in the sphere of protection of the current process. For names of private variables of a procedure (such as p and q in our example) the letters represent the words themselves.

Certain processor registers must be identified, namely those that contain the address word of the current procedure step (program counter), [illegible] that the [illegible]

and [illegible] to change the [illegible]

5. [illegible] Detailed Example [illegible]

Organisation of the Stack [illegible]

Let the base stack address for a procedure [illegible]

[illegible]

Suppose that, each time a procedure is called, the following are stored at the indicated addresses relative to the base stack address:

0. the stack pointer just prior to the procedure call.

1. the segment name of the calling procedure

2. the word address of the calling instruction in the calling procedure

Words 3, 4, ... relative to the base stack address contain the parameter information for the actual parameters of the call. Subsequent [illegible] are used for private variables required by the called procedure. The (non-relative) word address of the first free word in the stack [illegible] to the base stack address for any procedure called by the called procedure.

Subroutine Entry

The subroutine entry [illegible] for [illegible] takes the following [illegible]:

M/n_1 [illegible]

Here M/m and W[illegible] are the word names of the [illegible] new procedure and the entry point of the procedure [illegible] and [illegible] is the increment to be applied to the stack pointer [illegible] execution of the [illegible] function is as:

i) Save the stack pointer in the [illegible] register

$x0 \rightarrow [a0/x0 \quad [illegible]]$

ii) Increment the stack pointer

$x0 + \Delta \rightarrow x0$

iii) Save the return word name

register name

$[PAR] \rightarrow [a1]$

word address

$PWA \rightarrow x1$

iv) Establish execution of the subroutine

$\langle Y \rangle \rightarrow PAR$

$f \rightarrow PWA$

Before calling another procedure, F(x) must store the return word name in the stack as private data, for example in:

$[a1] \rightarrow [a0/x0 + 1]$

$x1 \rightarrow [a0/x0 + 2]$ (1)

The reverse must be performed before return to the main program

$[a0/x0 + 1] \rightarrow [a1]$

$[a0/x0 + 2] \rightarrow x1$ (2)

The <return> instruction performs the following steps

i) Back up stack pointer

$[a0/x0] \rightarrow x0$

ii) Resume execution of the calling procedure

$[a1] \rightarrow [PAR]$

$x1 + 1 \rightarrow PWA$

[illegible]

[illegible]

[illegible]

[illegible]

Main Procedure

```
M/r, 1 → a0
      2 → [a0/x0 → [illegible]]
      [illegible] → [a0/x0 → [illegible]]
      [illegible] → [a0/x0 → [illegible]]
      [illegible] → [a0/x0 → [illegible]]
      [illegible] → [a0/x0 → [illegible]]
      [illegible] → [a0/x0 → [illegible]]
      attach   f, a2
M/r, enter   a2/f, 0
      quit
```

Figure 5. [illegible]

[illegible]

```
F/7,            .
                .
                attach [a0/(x0 + [illegible]
                fail [a0/(x0 + [illegible]

                [a0/(x0 + 9)]    +    [illegible]

                .
                .

                [a0/(x0 + 5)] → [illegible]
                [a0/(x0 + 5)] → [illegible]
                [a0] → [a0/(x0 + 10)]
                x0 + 9) → [a0/(x0 + 1)]
                [a0/(x0 + 7)] → [a0/(x0 + 8)]
                [a0/(x0 + 8)] → [a0/(x0 + 9)]

                attach 6, a1
F/a,            enter a1/s, 11
                .
                .
                return
```

Figure 3. - continued

```
Procedure Gr (i)

G/s
                  - - -

                  - - -

      [a0/(x0 + 5)] --> [a0/(x0 + [illegible])]
      [a0/(x1 + 5)] --> [a0/(x0 + [illegible])]

      [a0/(x0 + 8)] --> [a0/(x0 + [illegible])]
      [a0/(x0 + 6)] --> [a0/(x0 + 15)]
      attach [a0/(x0 + 7)] . a1

G/t,  enter  a11[a0/(x0 + 8)]  .

      - - -

      - - -

      return
```

Figure 3 - (continued).

```
Procedure Sh.1

R/h.

            [a0/(x0 + 3)]    ->   [a0/(x0 + [illegible]
            [a0/(x0 + 4)]    ->   [a0/(x0 + [illegible]
            [a0/(x0 + 5)]    ->   [a0/(x0 + [illegible]
            [a0/(x0 + 6)]    ->   [a0/(x0 + 2[illegible]

            attach [a0/(x0 + 5)], a2
R/u.        enter  a2/(x0 + 6); 7

            return
```

Figure 3 - (continued)

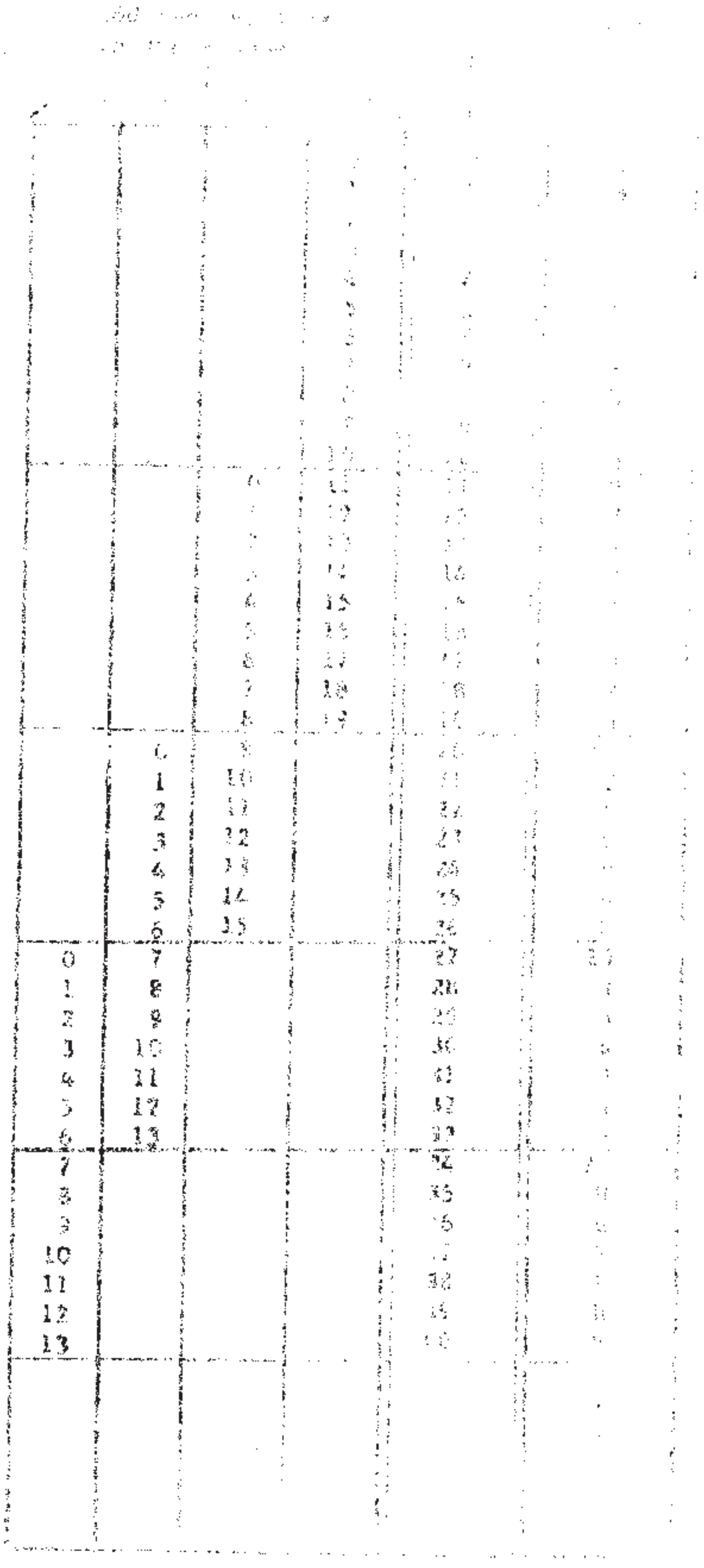

Figure 4 - Condition of the stack in the first [illegible]

The Second Detailed Example - Starting [illegible]

Tag Allocation

We suppose that attachment tags a0, a1, [illegible] for embedding in procedures. All other tags used by [illegible] obtained by use of the meta-instruction

obtain S → a

that loads an available attachment register with the segment name S and has the corresponding attachment tag as its value. The meta-instruction

release S

releases the attachment register associated with the segment named S.

We introduce one additional notation convention. Marking an address word by an asterisk

a0/(x0 + 4)*

means that the addressed word is itself an address word - starred or unstarred - that is to be used to fetch an operand (indirect addressing).

Organization of the Stack Segment

With tag passing the stack is somewhat simplified. Instead of storing the word name of the entry instruction in registers 1 and 2 above the base stack address, we simply store the address word of the entry point in register 1 above the base. The ⟨enter⟩ and ⟨return⟩ macros now become

```
M/m,        enter an/b,

               i)  x0 → [a0/(x0 + 4)]
              ii)  x0 + 4 → x0
             iii)  PAT/PWA → [a0/(x0 + 1)]
              iv)  an/b → PAT/PWA
```

[illegible]

(i) [illegible]

(ii) PN2 [illegible] PN3

(iii) [illegible]

Coding

The coding of the example is given in figure [illegible] condition of the stack just after the [illegible] time.

```
main procedure

    M/w1          0 → x0

                  (obtain X) (x → [x0//x1 + [illegible]]

                  (obtain B) (b → [x0//x0 + [illegible]]

                 <B> (b → [a0//x0 + [illegible]]


    M/w2          enter (obtain F)(F,0)


                  release X

                  release B

                  release F

                  quit
```

Figure 5 - Second detailed example

```
Procedure [illegible]

    [illegible]

        [illegible]

        [illegible]

        aU/[illegible]

        [illegible] aU [illegible]

        aU/[illegible]
        aU/[illegible]
        aU/[illegible]

    F/s    enter [illegible]

        [illegible]

        return

Procedure G[illegible]

    G/s

        [illegible]

        [illegible]

        aU/[illegible]
        aU/[illegible]

    G/t.   enter [illegible]

        [illegible]

        [illegible]

        return

Procedure H[illegible]

    H/b.

        [illegible]

        [illegible]

        aU/[illegible]
        aU/[illegible]

    H/u.   enter [illegible]

        [illegible]

        [illegible]
        return
```

Figure 5 (continued)

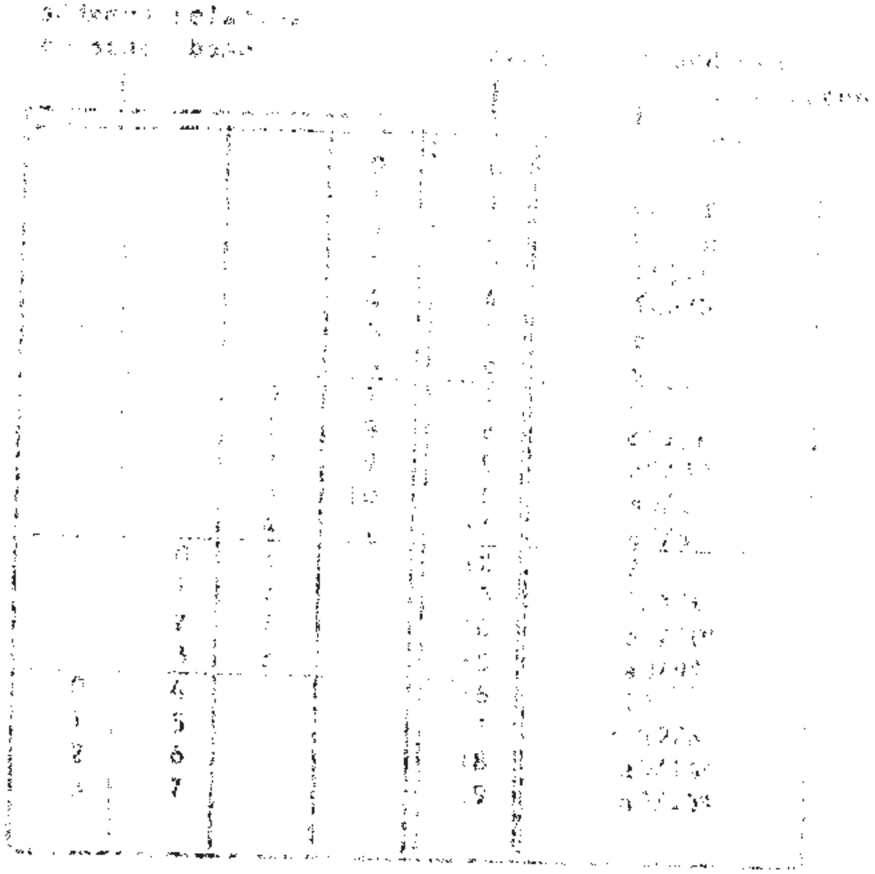

Figure 6 - Condition of [illegible]

The Third Detailed Example – [illegible]

Organization of the Stack Segment

The stack segment is organized in this example as it was in the first example.

Subroutine Entry

In this example we will introduce, for convenience, the additional convention that upon subroutine entry x1 contains the base stack address of the calling subroutine. Also, in this example, it is convenient to use a return instruction which specifies a number, ξ, to be added to the return word address so that control can be returned to the [illegible] the follower.

To accomplish these two objectives, we let the entry instruction be identical to that used in the first example, except for the following additional step after step i):

$x0 \rightarrow x1$

We postulate a new return instruction,

return ξ

which performs the following steps

i) unload the return item from the stack

$[s0/(s0 - 1)] \rightarrow [x1]$

$[s0/(s0 - 2)] + \xi \rightarrow$ [illegible]

ii) back up the stack pointer

$[s0/s0] - 2 \rightarrow s0$

iii) restore the previous general register,

$\{e0+x0\} \longrightarrow x1$

iv) resume execution of the calling subroutine

$\{a1\} \longrightarrow \{PAS\}$

$x1 \longrightarrow PWA$

Addressing of Parameters

In this example, the followers give each function the [illegible] locations in the stack segment of the called subroutine's actual parameters relative to the base stack address of the calling subroutine.

Comparison with the First Detailed Example

The first and third examples both show parameter information as being moved from one location in the stack to another at each call. However in the third example, the coding which does this moving is in the called subroutine, and hence exists only once in storage for each subroutine. In the first example, the coding to do this movement must exist in storage for as many times as the subroutine is called. Looking at it another way, if a subroutine calls several other subroutines, it need move data from its callers' region of the stack only once. It then delivers this information to other routines by means of placing pointers in the followers of the calls. Of course these followers take up memory, but not as much as the coding necessary to perform the data movements that would be necessary on each call in the first example. Hence we see that the method of the third example consumes less storage than the method of the first example is [illegible] [illegible] [illegible] other procedures.

Figure 7 compares the number of memory cycles [illegible] parameters under various criteria of [illegible] transmission [illegible]. We see that the number of [illegible] register case.

		Position of parameter in calling routine	
		Stack	Operand Register
Position of parameter in called subroutine	Stack	ex. 1 : 2 ex. [illegible] 3	ex. [illegible] ex. [illegible]
	Operand Register	ex. 1 : [illegible] ex. [illegible]	ex. [illegible] 2 ex. 3 : [illegible]

Figure 7 - Number of memory cycles for parameter transmission in example 1 and 3.

Coding

The coding of this example is given in Figure 8. Figure 9 shows the condition of the stack just after [illegible] has been entered for the second time. Notice the similarity to the stack configurations of examples 1 and 2.

Main Procedure

```
M/e,      0 → a0
          X → [a0/a0]
          x → [a0/(a0 + 1)]
          X → [a0/(a0 + 2)]
          6 → [a0/(a0 + 3)]
          attach F, a2
M/e,      enter a2/F, 0
            0
            1
            2
            3
            [illegible]
            [illegible]
          halt
```

Figure 8 - Third detailed example

```
Procedure YF()

F/E.
          [a0,x0 → [a1/(x1 → 1)] [illegible] → [a0/x0 + 3]]
                   .
                   .
                   .
          [a0/(x0 → [a1/(x1 → 0)]) [illegible] → [a0/x0 + 4]]
          attach [a0/(x0 + 3)] , a2
          [a2/[a0/x0 + 3] → [a0/x0 [illegible]]]
          [a0/(x0 + 3)] → [a0/(x0 + 1)] → [a0/x0 [illegible]]
          [a0 ? → [a0/(x0 + 1)]]
          x0 → 9 → [a0/(x0 + 1)]
          attach 0 , a2
F/a.      enter a2/g, 11
           5
           6
           11
           12
           7
           8
           .
           .
          return
```

Figure 8 - (continued)

```
[illegible]
   [illegible]  [illegible]
                [illegible]
                [illegible]

                [illegible]
                [illegible]
   [illegible]  [illegible]
                 5
                 6
                 7
                 8
                [illegible]
                [illegible]
                [illegible]
[illegible]
   [illegible]  [illegible]
                [illegible]
                [illegible]

                [illegible]
                attach [illegible]
```

Figure 2 (continued)

```
H/u,       enter [a2 {a0/(x0 + 6)} , 7
             3
             4
             5
             6
           - - -
           - - -
           return 5
```

Figure 8 - (continued)

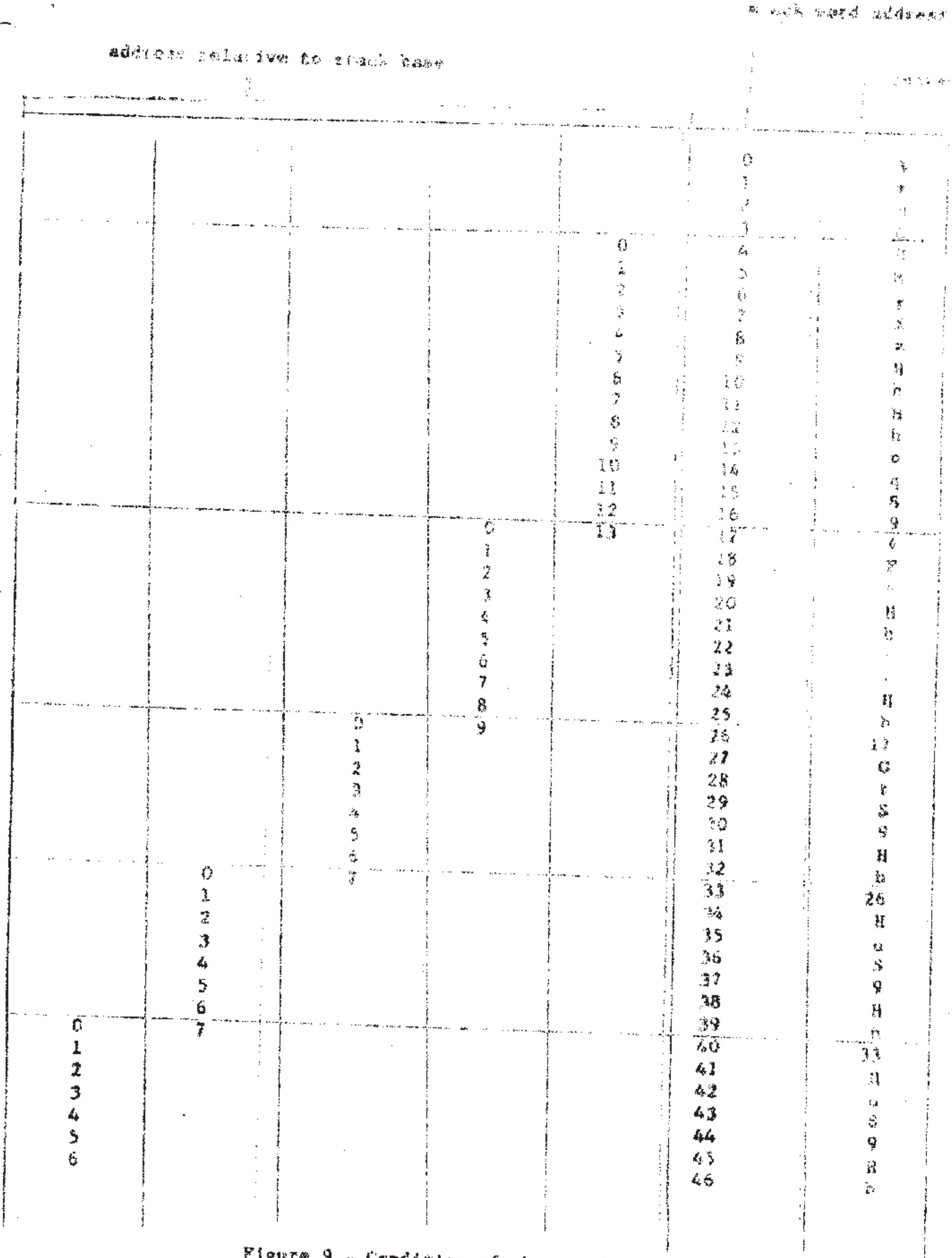

Figure 9 - Condition of the stack in the third detailed example