A GUIDE TO MULTICS

FOR

SUBSYSTEM WRITERS

Chapter I

A Primer on
Segmentation and Address Formation
in the GE 645

Elliot I. Organick

Draft No. 3

November 1967

Project MAC

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER I

SEGMENTATION AND ADDRESS FORMATION IN THE GE 645

1.1 INTRODUCTION

Multics is a computer and programming system environment which is new in concept, large in scope, and unique in its implementation relative to its predecessor systems. Where does one begin to describe it? One should start with a high-altitude overview so as to see the whole of Multics. One such overview has already been written, and is entitled A New Remote-Accessed Man-Machine System. It contains reprints of papers on the Multics system which were presented at the Fall Joint Computer Conference, Las Vegas, Nevada, on November 30, 1965. No doubt, as Multics becomes a working reality, more up-to-date "top-down" descriptions will appear.† It is presumed that the reader of this guide has already read the above-mentioned overview.

Computer specialists who read introductions of this sort are quick to grasp the aims and objectives of Multics, but some others may be slow to comprehend how it works. They then find a need to dig into the details, attempting to reach an ultimate understanding of the system by starting from the inside core and working outward toward a grasp of the whole.

What constitutes the "inside" of Multics? Until it is understood better, there will probably be differing opinions. Rather than speculate, we have chosen to begin by looking at some of the new features of the GE 645, especially address formation. The purpose is to show how these new hardware features are related to, and permit the effective implementation of, the new software concepts of Multics; chiefly that of process segmentation.

G0029 was used as the prime technical reference for address formation and segmentation hardware in the GE 645.* Much more is covered in G0029 which is of lesser interest to a Multics subsystem writer. You can use two approaches depending on the availability of the reference document:

† As of May 1967 the extensive overview "Multics Operating System" prepared by GE's Cambridge Information Systems Laboratory has become available but limited at least initially in its distribution to those affiliated with Project MAC.

* Since this chapter was written, a new more complete GE reference document, M50EB00107, entitled Engineering Product Specification GE-645 Prototype Processor, became available; dated November 8, 1966.

1) If you have already dug into G0029, you will find the Figures and Tables of this first Guide chapter are helpful summaries, and can ignore the text portion of the chapter;

2) You can use the first chapter as a Primer on segmentation and addressing, and at least temporarily avoid the need to reference the G0029 document.

1.2 SOME DEFINITIONS AND CONCEPTS

1.2.1 Processor

A computer processing unit like the CPU of the IBM 7094.

1.2.2 Process (Lay definition) *

A set of related procedures and data undergoing execution and manipulation, respectively, by one of possibly several processors of a computer.

1.2.3 Segments

A segment of a process is a collection of information important enough to be given a name. Segments are, generally speaking, blocks of code (procedures) or blocks of data ranging in size from zero to 2^{18} words in units of 2^{10}. The segments of a process may be located in discontiguous sections of core memory and still function effectively as a unit.

Each segment can be allowed to grow or shrink during execution of the process. A record of its size is kept in the "descriptor word" associated with the segment.

1.2.3.1 Page

Unseen by the user, hardware mechanisms exist for subdividing a segment into smaller units called pages, each of which may be located in smaller, discontiguous blocks of core memory. All pages of a segment are of the same size and are either 64 words or 1024 words, at the option of the

* Dennis and Van Horn (MAC-TR-23) give the following technical definition: "A process is a locus of control within an instruction sequence. That is, a process is that abstract entity which moves through the instructions of a procedure as the procedure is executed by a processor."

supervisor. If a segment is so subdivided, it is said to be "paged". Every segment which a user will have any control over will be paged.

1.2.3.2 Page Table

For each paged segment, the supervisor creates a table called a page table which, when stored in core memory, will contain pointers to the individual pages of the segment that are also currently stored in memory.

1.2.4 Descriptor Segment

A table of some of the facts concerning the segments of a given process, one entry per segment. (See Figures 1-1, 1-2; pages 1-9, 1-10.) Specifically

1) the location of the page table for the segment;
2) the access rights for use of the segment.

1.2.5 Descriptor Word

A single entry (36 bits) in the descriptor segment. Each entry contains a pointer to the page table of the segment, if the segment is known to be residing in core memory. Otherwise, an indication of the segment's absence from core is provided in the entry. Also contained in the entry is the size (or maximum allowable size) of the segment and a "descriptor" field.

1.2.6 Descriptor Field

This field is sometimes referred to as the access control field. Bits in this field are set by the supervisor and interpreted by the hardware. (See Table 1-1, but only for details.) A segment exists in one of five classes defined by one subfield (bits 33-35). Another subfield defines the access rights to the segment of a currently executing segment; i.e., who may read and who may write in the segment. In the event the segment is not in core, the subfield defines the desired trap (hardware fault) to the supervisor. Other bits relate to details of the segment's further subdivision into pages — the level of detail that is curtained off from the eye of the user.

1.2.7 Supervisor

A collection of segments made part of each user process which perform various process management service functions. Thus, certain supervisor

segments are responsible for core allocation, others are used for searching secondary storage for needed segments and still others are used for the construction, loading and maintaining of segment or page descriptor words. Some supervisor segments operate in slave mode and others operate in master mode.

1.2.8 Mode of Execution – Master/Slave

There are two modes of execution, Master and Slave. A Procedure segment is classified as either master or slave by a bit set in its segment descriptor word. When the processor is executing in a master procedure segment, any one of the entire repertoire of 645 instructions may be executed. When the processor is executing in a slave procedure segment, certain instructions are "off-limits". An attempt to execute one of these will cause a hardware fault which in turn causes a trap to one of the supervisory segments of the process.

1.2.9 Segment Class

There are five classes for a segment (A-000, B-001, C-010, D-011, E-100).

1.2.9.1 Missing Segment (A-000)

Segment is missing; i.e., is not now resident in core memory. Attempted access of any word in this segment will automatically cause a fault; one of eight types depending on the kind of access problem that was encountered. (Bits 30-32 of the descriptor are used for indicating the kind of fault.)

1.2.9.2 Data Segment (B-001)

Segment is data (it may be read or written on only according to the setting of write and read permit bits (30 and 31) of the descriptor field). It may never be executed, i.e., information from this segment cannot be fetched during the instruction cycle of the 645.

1.2.9.3 Ordinary Slave Mode Procedure Segment (C-010)

Segment is a procedure, garden variety. It's called ordinary slave ØS. Is the kind that any user can write. It may contain data. That is, modifications may be made to this procedure by the same or other procedures. See BD.7.02 for details on how to set up call, save and return sequences.

A procedure segment that is not modified in any way as a result of being executed is called a pure procedure. All others are regarded as impure.

1.2.9.4 Execute-Only Procedure Segment (D-011)

Segment is another kind of procedure that operates in slave mode but it's restricted to an ordinary user in the sense that a slave mode procedure can only transfer to it via the very first word of the segment, i.e., word zero. We call such a procedure an execute-only procedure, EØ. It's restricted in order that it can decide whether the caller was a valid caller, i.e., to control the access to this procedure. A fault will occur if an attempt is made by any procedure which is operating in slave mode to transfer into any part (other than word zero) of the EØ procedure. Moreover, no part of this segment may be read as data by another procedure that operates in slave mode. Any user can write one of these procedures, but the call, save and return sequences are slightly different. See BD. 7.03 for details.

An EØ procedure, in spite of its misleading name, can make self references, i.e., read itself and modify itself, i.e., it may be impure.

Writing impure either EØ or ØS procedures is not recommended, since a single copy of an impure procedure cannot effectively be shared by two or more processes.

1.2.9.5 Master Procedure Segment (E-100)

Segment is a master procedure. Meaning, it operates in master rather than slave mode, and hence can execute the "privileged" instructions. A master procedure can also transfer into EØ segments at points other than the top, i.e., word zero. If a slave mode procedure wants to transfer control to a master procedure, the master procedure will appear to the slave as an EØ. That is, a slave calling on a master must call "at the top".

1.2.10 Segment Naming and Segment Numbering (Notation)

Multics has adopted a standard notation to refer to the name of (i.e., symbolic reference to) a segment and to its number (i.e., to the position of its descriptor word within the descriptor segment — see Figure 1-2).

Example: if the segment is the cosine routine. The segment name might be referred to as <cosine> and its number is referred to as cosine#. In general

<seg> means that the name of the segment is seg

and

seg# means the number of the segment whose name is <seg>

1.2.11 Named Locations and Numeric Locations Internal to a Segment

Multics has adopted a standard notation to refer to an "internal address" i.e., to an address relative to the word zero of the segment. If the location is known symbolically, e.g., via a name in the location field of the assembly-language coding, then this name is indicated by enclosing the name in brackets. Thus in <cosine>, if there is an instruction named "loop" then this name is referred to as [loop] and the symbol "<cosine>|[loop]" means location named "loop" within the segment named "cosine".

The vertical bar always separates the segment name or segment number from the local, or internal name.

Example:

<k>|[kloc]

means location "kloc" within segment "k".

To distinguish the internal symbol from its value, we simply remove the brackets when we want to speak of the value assigned to that symbol. Thus if [kloc] were assigned the value 52 within <k>, and if k# were 16, then

k#|kloc

would mean: position 52 from the top of segment number 16. This pair of numbers determines the core memory address of <k>|[kloc] during execution, once we determine the address of

k#|0

which is the address of word zero of <k>. We note that the address of k#|0 is stored as a pointer in the 16th word of the descriptor segment. See Figure 1-2 for details.

1.2.12 Core Address

A full memory address is 24 bits. In forming this address for a fetch or store of a word, or of a pair of adjacent even-odd words, the GE 645 employs

address components, which are stored as 18-bit fields (or less). To form a 24-bit core address, a left shift of 6-bit positions is performed on the value copied from the pointer in the descriptor word. If, for example, the pointer has the value χ, then the core block to which it points is located at $\chi \times 2^6$. In Figure 1-2 and in succeeding representations of descriptor words, the pointer field is represented by the symbol A_B.

1.2.13 Address Bounds Field of the Descriptor

Each descriptor word (See Figure 1-2) contains an 8-bit field called N_B (bits 19-26). This field reflects the size of the segment, measured in blocks of 2^t words. N_B is also the number of words in the segment's page table. Each time a memory address is made to a position within a segment, <k>, say at <k>| kloc, the value of N_B is automatically employed by the hardware as a bounds check to be sure that <k>| kloc lies within <k>.

If the size of the segment in words is size, then $N_B = size/2^t$. Bit 27 of the descriptor word determines t.

t = 10 if bit 27 = 0, i.e., page size is 1024 words

t = 6 if bit 27 = 1, i.e., page size is 64 words

This means that a segment is allocated space in blocks of 1024 words (2^{10}) if bit 27 = 0. Such a segment can never exceed $2^{8+10} = 2^{18}$ words even if core memory should grow beyond 2^{18}. Otherwise, if bit 27 of the descriptor word = 1, the segment is allocated space in blocks of 64 words, in which case, such a segment can never exceed $2^{8+6} = 2^{14}$ words.*

1.2.14 Descriptor Base Register (dbr)

A 29-bit register (one per processor). This register has only two fields of interest to us. They are called "addr" and "bd" in all the figures that depict the contents of the dbr. We pay no attention to the field marked "desc".†

* Some hardware changes are being planned for the GE 645 which will make it possible for a segment whose page size is 64 words to have up to 2^{12} pages, i.e., up to 2^{18} words. I don't know how such a planned change will affect the interpretation of the address bounds field.

† This is a 2-bit field that gives information as to whether and how the descriptor segment, pointed at by addr, is subdivided into pages. We will never, as users have any control over these bits. This is a supervisory function of a process called the core manager (BG.6).

addr is a pointer to the head of the descriptor segment for the process that is currently running on the processor to which this dbr belongs.

bd is a bounds value. It gives the size of the descriptor segment measured in blocks of 2^v descriptor words. v = 10 or 6 in value depending on whether bit 27 of the dbr is set to 0 or 1 respectively. If the addressing mechanism (described in Figure 1-2) is ever used to access a word in the descriptor segment that lies beyond addr + bd x 2^v then an automatic fault will be detected and control would be transferred to a master mode supervisory procedure segment. (Remember that the supervisory procedures are automatically made a part of every user process.)

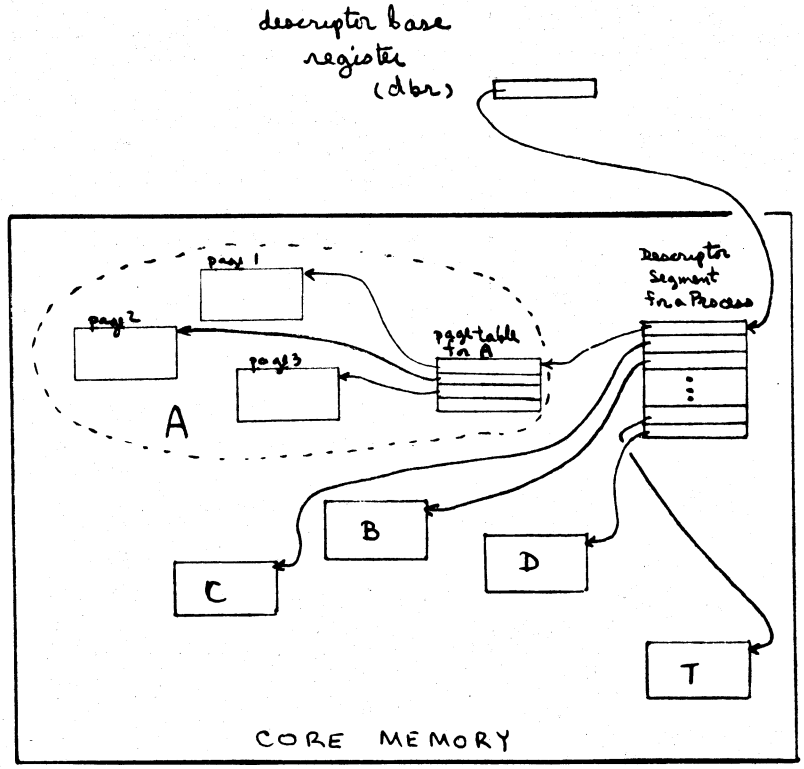
1.2.15 Two or More Processes in Core Memory

In principle, a number of different processes may cohabit memory. (See Figure 1-3.) To switch a processor from process 1, say, to process 2, all that is required is to save the contents of the dbr (i.e. (dbr)), and replace with the addr, bd, and desc fields appropriate to process 2. In simple terms, just make the dbr point at a different descriptor segment.

In actual fact, process switching in Multics is somewhat more complicated. Those interested can read the details in the Traffic Controller sections of MSPM. See for example BJ.5.

1.2.16 Common Segments

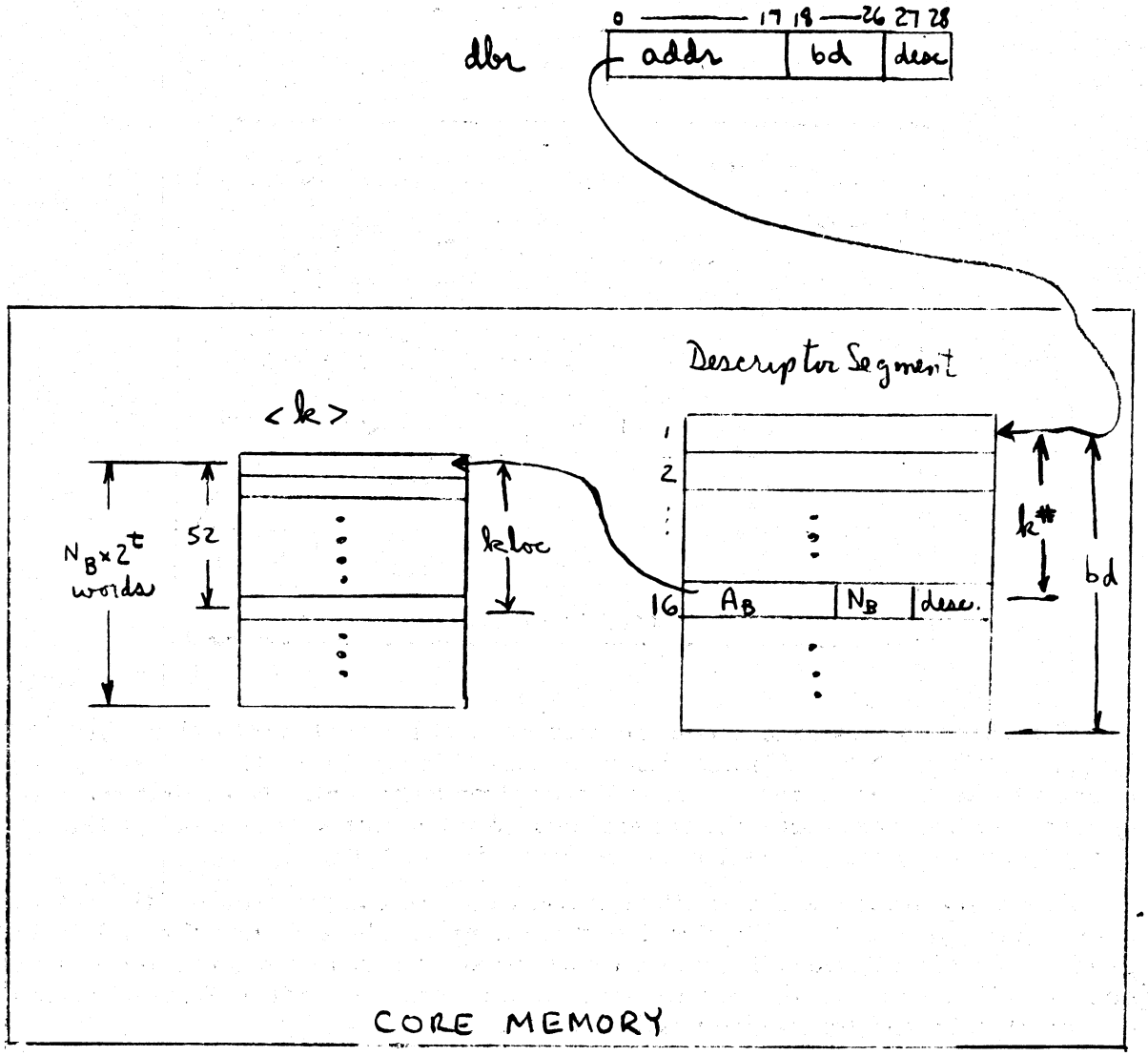
Note in Figure 1-3 that two processes (residing in memory at the same time) can share common data or procedure segments (and their respective page tables). In fact, in Multics this is definitely the rule. Every process has certain key segments which are master supervisory procedures provided by the system. Some are called "hard core" and "administrative" routines. Certain of these procedures are automatically made part of every Multics user process. These are generally the lowest numbered segments in the process. Moreover, a few of the segments which are employed for process switching are given the same numbers respectively in each process (like segments <a>, and <c>). Other segments may be common, but they need not be numbered identically in each process.



The process has a number of segments called (symbolically) "A", "C", "B",..., "T", "D". This is the order in which pointers to these segments happen to be listed in the process descriptor segment. Note that the current contents of the descriptor base register (dbr) points to the head of the descriptor segment. The dbr is explained in 1.2.14.

Each segment should, strictly speaking, be diagrammed with the detail shown for segment A. That is, first the page table and then the individual pages. To simplify our figures, we will ordinarily avoid such detail and hope it will be inferred from the simplified, single block pictures we shall use as illustrated for segments C, B, etc.

Figure 1-1. Showing a Process Resident in Core Memory

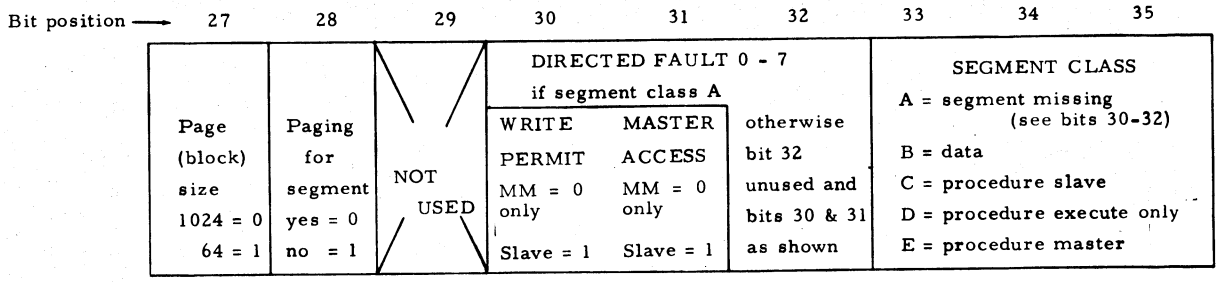


Note: See 1.2.10 and 1.2.11 for an explanation of the segment naming and numbering notation used in this diagram.

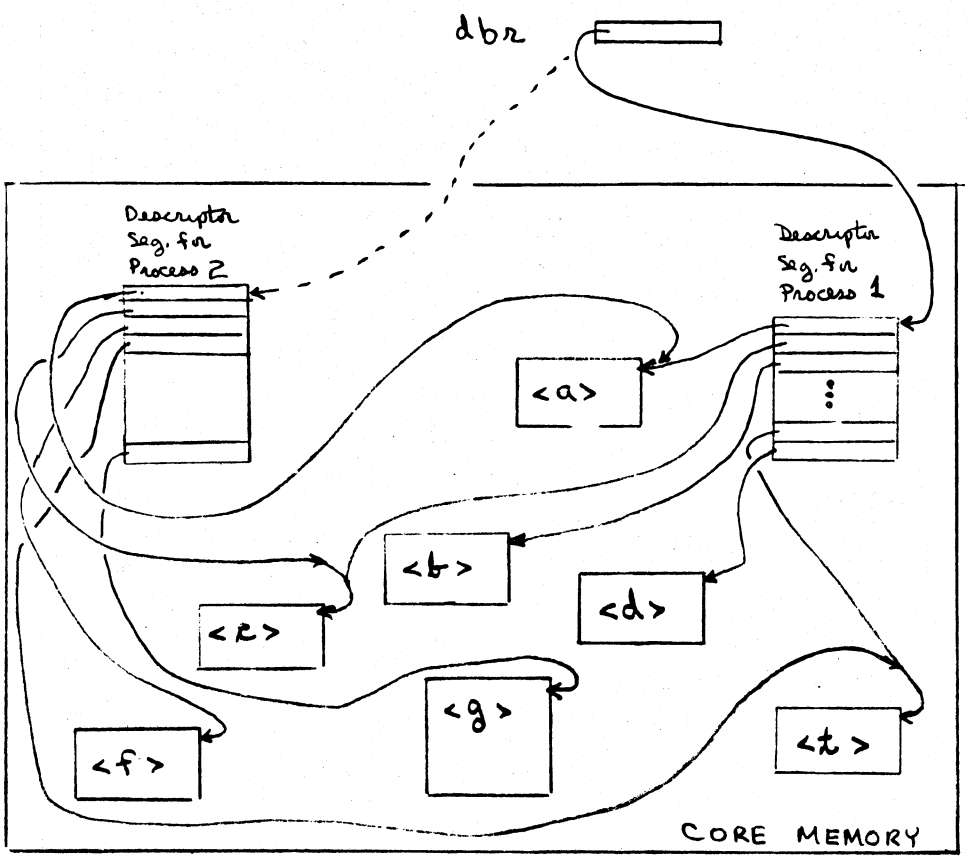
Figure 1-2. System of Pointers to Locate the Core Address of <k> | [kloc]

TABLE 1-1.

Descriptor Field of the Segment Descriptor Word



Note: Binary coding is used for the SEGMENT CLASS subfield (bits 33-35) of the segment descriptor word; i.e., A = 000, B = 001, C = 010, D = 011, and E = 101. Binary coding is also used for the directed faults (bits 30-32). More details on segment classes B, C, D, and E, and on write and access permit bits, can be found in G0029, pages 38-41.



Process 1: consists of segments called <a>, <c>, <d>, . . . , <t>, <d> in the order listed in the descriptor segment.

Process 2: consists of segments called <a>, <c>, <f>, <t>, . . . , <g>, in the order listed in the descriptor segment.

Note: Both processes share certain segments in common. Thus, segment <a> and <c> are identically numbered 1 and 2, respectively, in each process. But segment <t>, which is also a common segment, has a different number in each process.

By the number of a segment in a process we simply mean:

> A segment has the number k if it is pointed to by the k^{th} word of the descriptor segment for the given process.

Figure 1-3. Showing Two Processes Resident in Memory

1.3 CORE ADDRESS FORMATION

We will now see how core addresses are formed for purposes of fetching instructions and data during the execution of a user's process.

Conceptually, every memory reference is to a particular location within a particular segment. This is sometimes referred to as "two dimensional" addressing, because one dimension may be thought of as the segment number, an offset within the descriptor segment which determines the desired segment, and another dimension is the offset within the desired segment. We shall sometimes speak of these as the first and second dimensions, respectively. More often they will be referred to as the external base and internal base or address.

1.3.1 Fetching Instructions

In a familiar computer like the IBM 7094, addressing is one dimensional. The instruction counter, IC, holds the absolute location of the instruction to be fetched. In the GE 645, the IC merely holds the value of the second dimension, i.e., the relative location within the desired procedure segment. The value for the first dimension is the segment number for the currently executing procedure. This number points to the desired descriptor word, which in turn contains the pointer A_B to word zero of the desired segment. Construction of the actual core address that is finally referenced is as follows:

core address for the fetched instruction = address of desired segment + (IC)

If a procedure segment is executing a sequence of instructions that lie entirely within the segment, the value A_B remains stationary. Only the IC changes in value, increasing by one for sequentially executed instructions, or replaced by new values in the case of a successful transfer instruction to some arbitrary point within the same segment.

How, then, is A_B established? If we are talking about segment <k>, then clearly some register could be designated to serve as a pointer to the proper descriptor word that holds A_B for <k>.

In the GE 645, a special 18-bit register has been provided for this purpose, and it's called the "pbr" or procedure base register. The value k# must be established and held in the pbr as long as procedure <k> is executing. Figure 1-4 gives an illustration of this instruction address determination. In this and subsequent figures the first dimension of the address is referred to as the "effective pointer", a term used in G0029.

1.3.2 Fetching or Storing Data

Most procedures in Multics will be pure, so the data associated with the procedure are almost certain to be located in some other segment of the same process. Hence, to form the address of a word in the data segment, it's necessary to point within the descriptor segment to a different descriptor word than the one currently pointed at by the pbr. If we are going to hold a value in the pbr as an "instruction" pointer for a series of instruction fetch cycles, we'll need another register for use as a data pointer during a series of execute (data fetch) cycles. In this way the computer can alternate between instruction and execute cycles without having repeatedly to shift pointer values to the appropriate segment descriptor word. The special 18-bit register used in the GE 645 for the execute cycle address formation is the "tbr", or temporary base register. On each execute cycle the tbr holds the value for the first dimension of the two-dimensional data address.

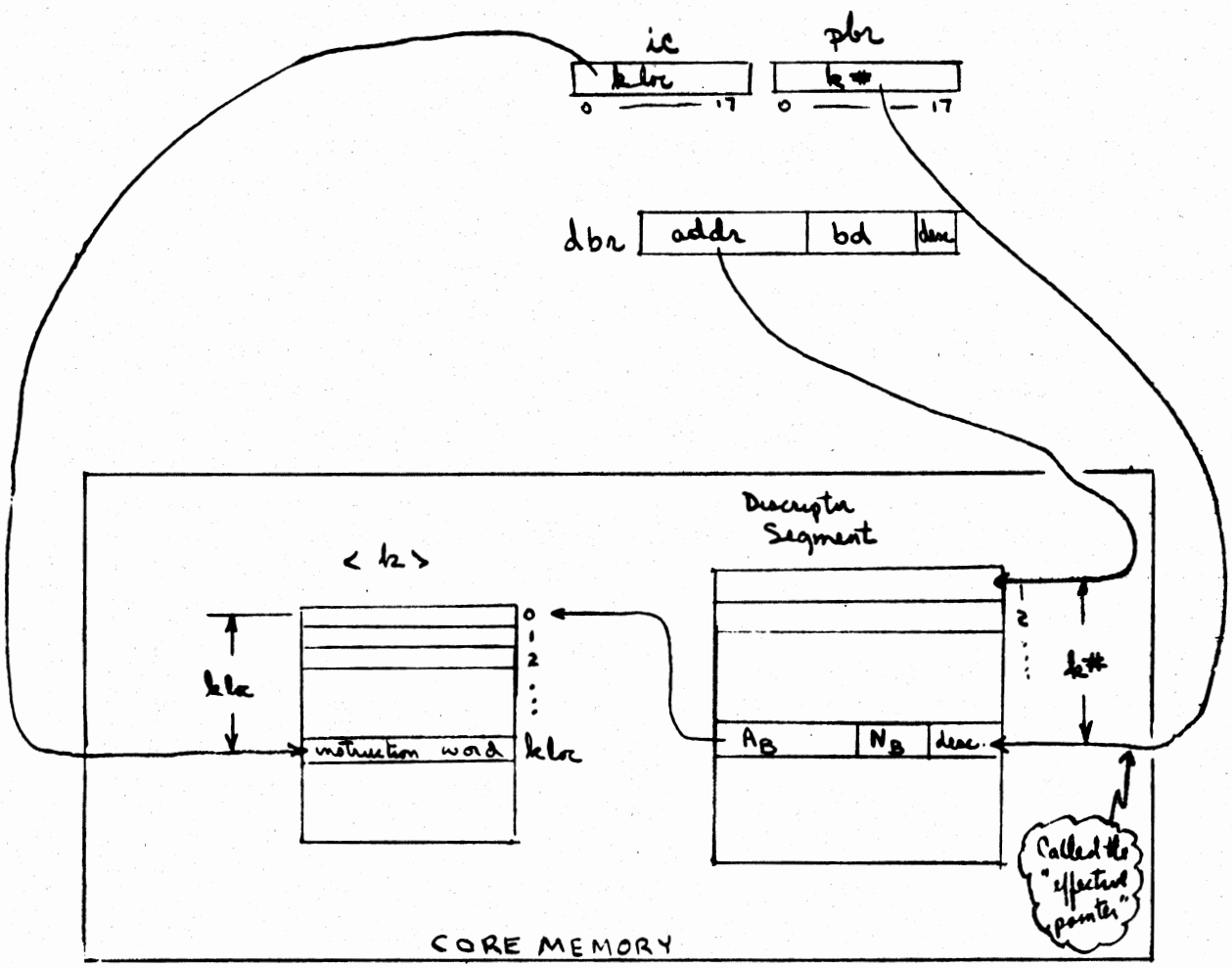
Figure 1-5 illustrates the partial determination of the core address for a data word located symbolically at

<d>|[dloc]

In this figure we show how the address of <d> is determined (although we have not yet shown how we got the value to put in the tbr). It remains now to show how the relative location within <d> i.e., dloc is determined. G0029 calls this second dimension the "effective internal address".

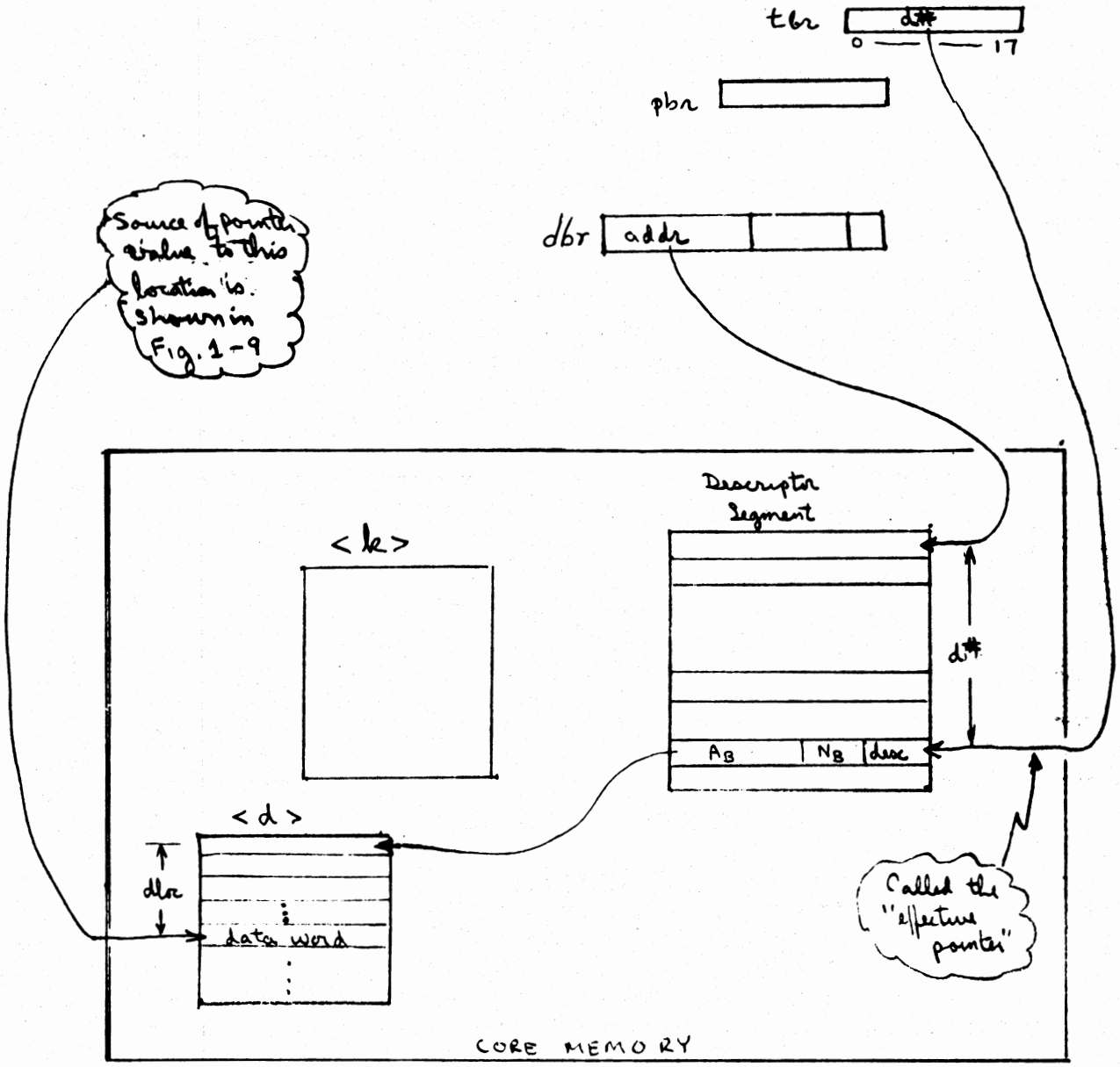
1.3.3 Computing the Effective Internal Address

We have seen how the address of data segment <d> has been determined. Now the question is, how is the effective internal address, dloc, (i.e., the address within <d>) determined? We shall have to approach the full explanation to this question in stages.



The actual core address is $A_B \times 2^6$ + kloc. A_B is found by setting a pointer value, k#, in the pbr. The (dbr) points at the base of the descriptor segment and the (pbr) provides the necessary offset to get at A_B.

Figure 1-4. Address Formation to Fetch an Instruction at <k>|[kloc]



tbr points at the d#th descriptor word which in turn points at <d>.

Figure 1-5. Partial Address Formation During an Execute Cycle to Obtain a Data Word at <d>|[dloc]

There are two characteristic formats for a GE 645 instruction as shown in Figure 1-6. Instructions of type 1 are recognized by virtue of bit position 29 = 1, while instructions of type 0 are recognized by a 0 in bit 29.

Type 0 instructions are used for referencing data or instructions within the segment currently being executed, while type 1 instructions may reference data or instructions in any segment. Because there appears to be something inherently more general about type 1 instructions (and for no other reason) we shall discuss these first, and treat type 0 instructions later (in Section 1.3.8).

For a type 1 instruction, the effective internal address is made up of three components. The three fields y, tag and segment tag of the instruction, either directly or indirectly determine these components.

The address field y is a signed 14-bit quantity, i.e., address ranges over $\pm 2^{14}$ values.

The tag field. There is a fairly complicated six-bit tag field. Some details can be found in Table 1-2. For the present we can make the following simplification:

1. three of these bits designate one of the 8 index registers (0-7);
2. two bits specify the type of indirect addressing, if any, for this instruction.

The segment tag field points to one of 8 so-called "address base registers".

1.3.4 The Eight Address Base Registers

There are eight address base registers (abr's). Any one of these may, in principle at least, be pointed at by the segment tag of a type 1 instruction. Historically, the purpose of the address base register was to be a convenient place to store effective pointers, i.e., segment numbers of data or procedure segments other than the one currently executing. As the hardware developed further another possible use was developed for these registers, namely as a convenient place to store the internal effective address or a component of it. The designers then decided to permit each abr to serve either purpose, that is to say it was allowed either to hold the effective pointer to a descriptor word, in which case it would be referred to as an "external base" or to hold

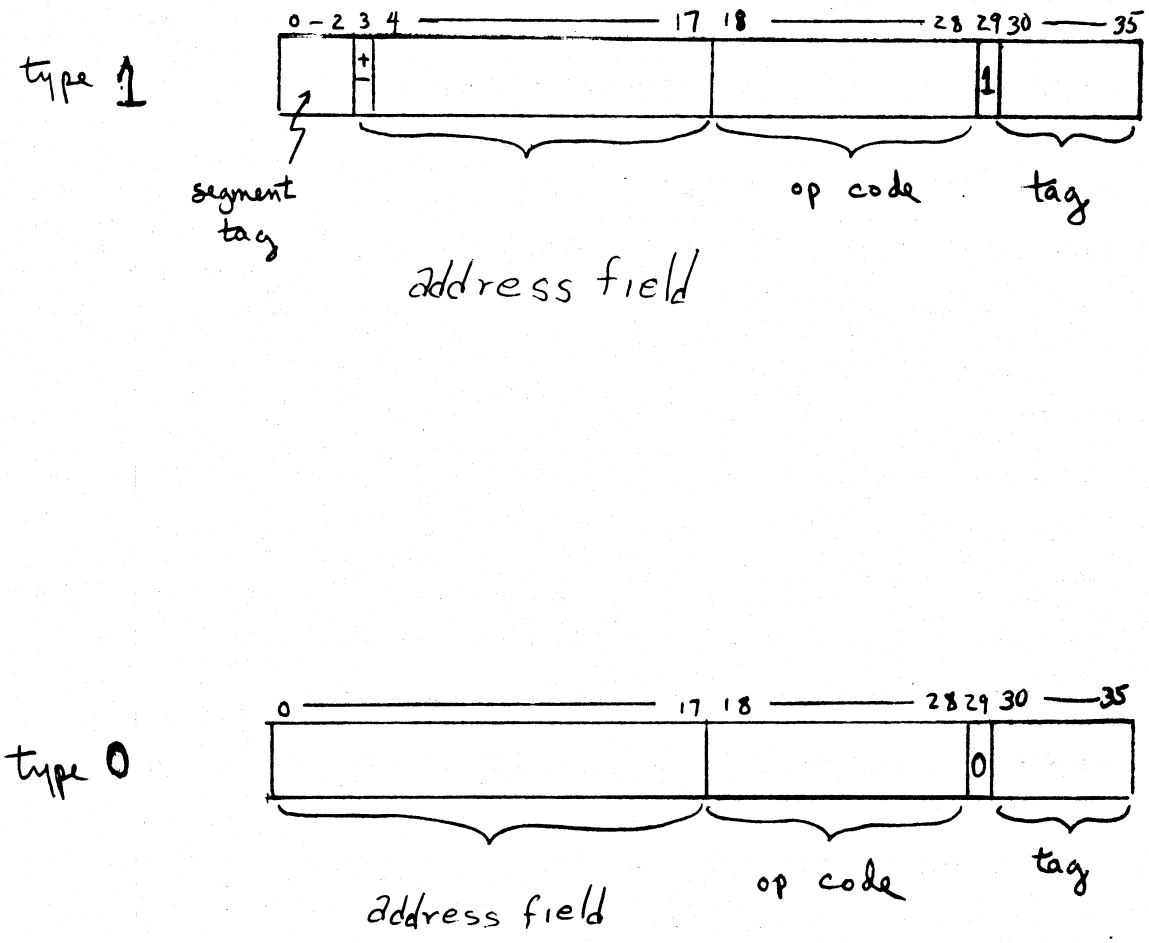
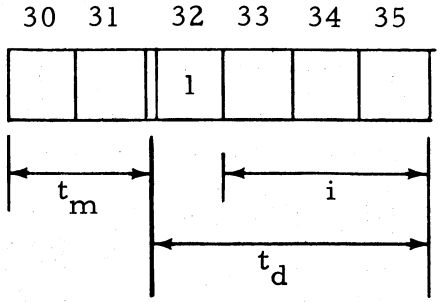


Figure 1-6. Formats of Typical GE 645 Instructions

TABLE 1-2

Tag Field Details for Indexing and Indirect Addressing



There are two main subfields which are described in detail in the GE 625/635 Reference Manual, p. II-24 through II-26. Additional details may be found in G0012, pages 16-19.

t_d, provided bit 32 = 1, designates index register i in bits 33-35.

t_m designates the type of indirect addressing, if any. Several types of indirect addressing are available, however, only one is of immediate interest.

t_m	635 Parlance	Interpretation
00	R-type	no indirect addressing
01	RI-type	multi-level indexed indirect addressing
11	IR	not discussed in this Guide. See reference documents for details.
10	IT	

a component of the effective internal address, in which case it would be called an "internal base". As an internal base the abr can act like a second index register, but if so, it must also point to another abr holding an external base. Each abr has 24 bits, sufficient to hold both an 18 bit address field and a control field with which to identify the function of the first field as external or internal. This is illustrated in Figure 1-7.

There is an interesting capability for pairing the abr's. If bit 21 is a "0", then bits 0-17 serve as an internal base. It serves as the so-called "p" component of the effective internal address. Moreover, three other bits in the same register (bits 18-20) are interpreted as a pointer to another abr whose bits 0-17 are then taken as the external base or effective pointer.

In short, by properly setting the control bits in the abr, the segment tag of a type 1 instruction can point not to one, but to a pair of abr's. The first will contain a component of the internal address and the second will contain the effective pointer.

In Multics operation, the eight registers are paired by presetting the control bits (18 - 21) in the registers so that they may always act effectively as 4 pairs of base registers (18 bits per register), as shown in Figure 1-8. To address a particular pair of these registers in a type 1 instruction, one needs to give as the segment tag the address of the internal base*, i.e., 0, 2, 4, or 6.

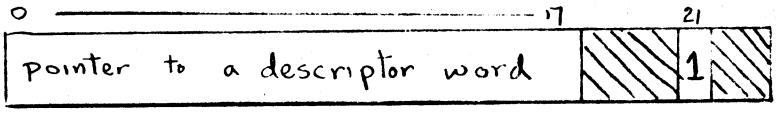
We are now ready to complete our unfinished explanation of how to determine the core address for the data word <d> | [dloc]. The internal effective address, dloc, in this case is formed from three components y, (i), and p

dloc = y + (i) + p

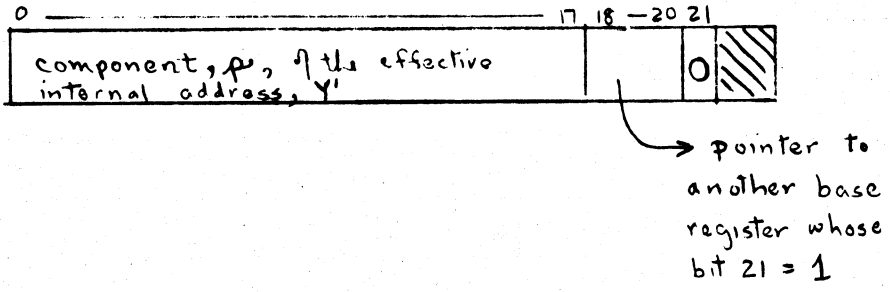
---

* In writing instructions in a Multics assembly language, we use standard two-letter names in place of the numeric values. These are:

ap for address base register 0
bp for address base register 2
lp for address base register 4
sp for address base register 6

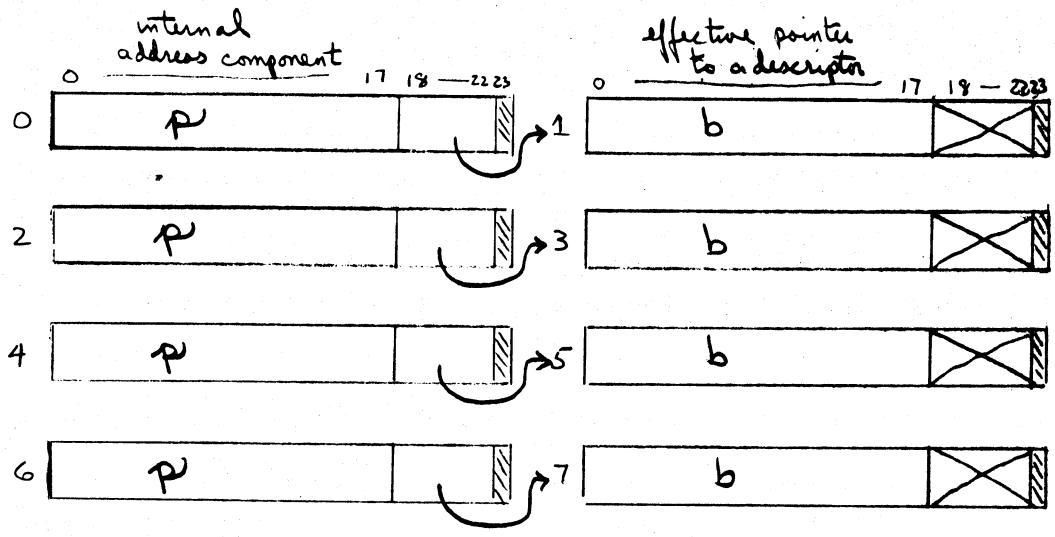


a. as an external base



b. as a component of the effective internal address with a pointer to an external base

Figure 1-7. Two Ways for an Address Base Register to Function



(For more details on bits 18-23 in each register, see Table 1-3.)

Figure 1-8. Multics Standard Pairing of the Eight Address Base Registers

TABLE 1-3

Normal Control Field of the Address Base Registers

Register Multics Name	Register Number	Bit No. 18 19 20 (Specifies register which is external if this one is internal)	Bit No. 21 (Intern = 0 Extern = 1)	Bit No. 22 (Lock* Base)	Bit No. 23	Remarks
ap	0	0 0 1	0	0	Not Used	internal
ab	1	Not Used	1	0	Not Used	external
bp	2	0 1 1	0	0	Not Used	internal
bb	3	Not Used	1	0	Not Used	external
lp	4	1 0 1	0	0	Not Used	internal
lb	5	Not Used	1	0	Not Used	external
sp	6	1 1 1	0	0	Not Used	internal
sb	7	Not Used	1	1	Not Used	external

*Lock Base = 0 means this register may be loaded in slave mode. All abr's except sb are set = 0 by the Multics supervisor

Lock Base = 1 means this register is locked against change in slave mode. This means a user will never be able to destroy the current value in sb.

To alter any of the control field bits in an abr requires a privileged (Master Mode only) instruction, so the user will never be able to alter these bits.

where

(i) is the contents of the index register pointed at by the tag field of the instruction,

p is contents of the internal base address register pointed at by the segment tag field of the instruction.

Moreover, since the internal base register is coupled to an external base register, the contents of the coupled external base serves as the value, d#, destined to be copied into the tbr. All this is illustrated in Figure 1-9.

Before we attempt to summarize our discussions to this point, we offer one further pictorial technique to help us in the condensation of future diagrams. We shall in the future depict and name the group of 8 base address registers as shown in Figure 1-10.

The array representation of the 8 registers suggests that

(1) each row is a coupled base pair.

(2) the register in column 1 is the internal base or p-type and the register in column 2 is the external base or b-type.

The Multics names given to the 4 pairs are:

a for argument list pointer,

b for general base,

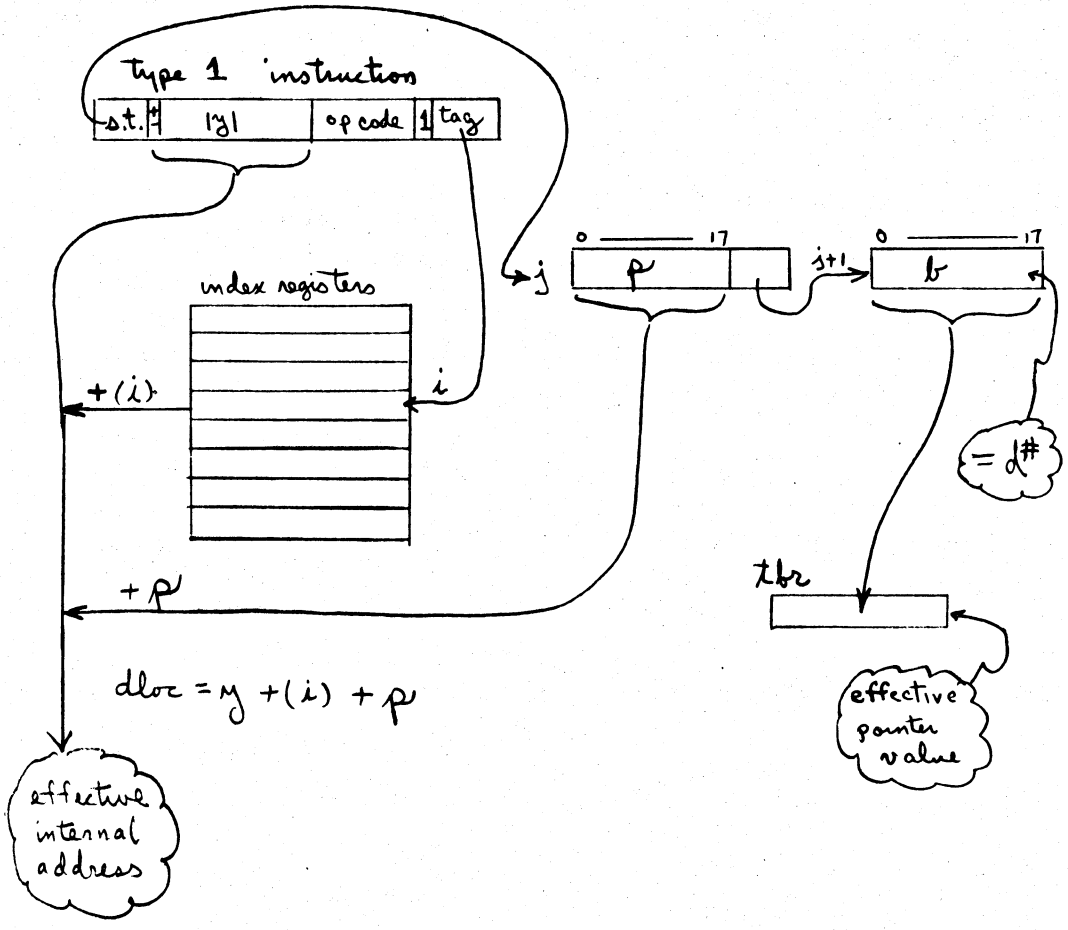
ℓ for linkage segment pointer,

and s for stack segment pointer.

Some preliminary motivation for these names is attempted in Section 1.3.5. You can skip over this for the present if you want to get on with the details of address formation.

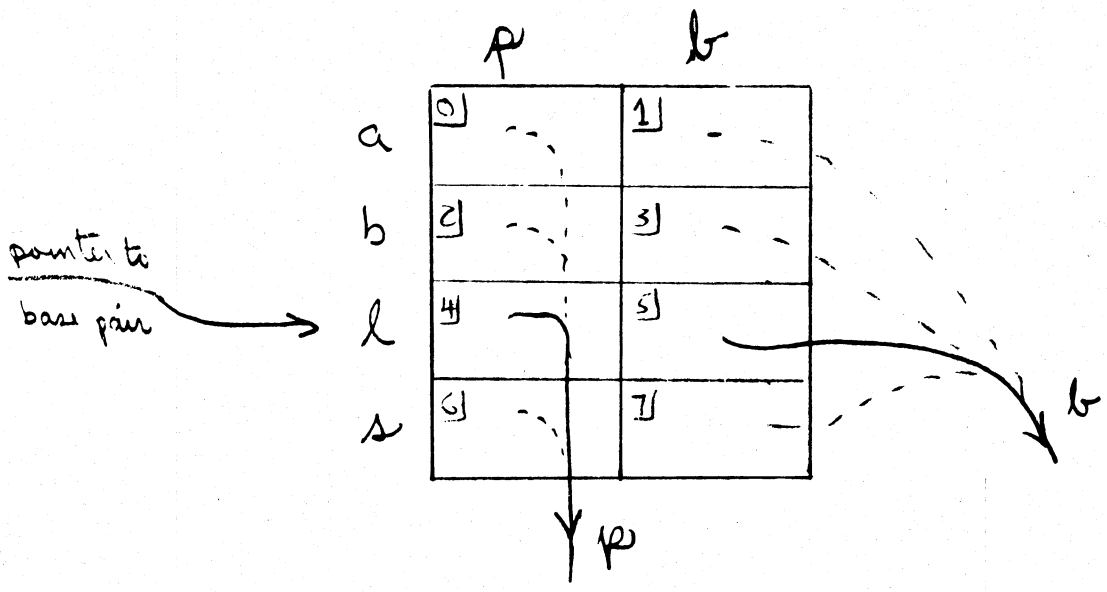
1.3.5 Address Formation Strategy

A typical process will involve a large number of segments. Many of these segments, as you will see, when you read other MSPM documents, are part of the system supervisor. Entries for these segments are automatically added to each process descriptor segment. Apart from these, each process will generally have one or more procedure segments, one or



The tag is assumed to indicate direct addressing rather than indirect addressing and points to index register i. The value of the segment tag is j where j = 0, 2, 4, or 6 in normal Multics operation. The current value of the coupled base register j + 1 is assumed to have been preset to equal d#.

Figure 1-9. Showing Effective Internal Address Formation from a Type 1 Instruction



The four coupled registers or "base pairs" are:

Numbers	Commonly Used Multics Notation
0, 1	ab ← ap
2, 3	bb ← bp
4, 5	lb ← lp
6, 7	sb ← sp

Figure 1-10. Eight Base Registers

more data segments, a stack segment and a linkage segment for each procedure segment. Let's suppose the process involves a typical MAD job. There would be a procedure segment for the main program, one for each separately compiled subprogram, and perhaps one for each library program.

Suppose the MAD compiler is written to structure object code in such a way that variables local to each of the MAD procedures (and for that matter possibly those in PROGRAM COMMON as well) are allocated storage in a single data segment. Let's suppose the name of this segment is <stat_>, for static storage.

A process will also refer to other data banks such as publicly available tables of read-only data, or perhaps an input buffer or an output buffer area. Such data areas, because they might have different access controls associated with them, will normally be stored in independent data segments.

Each time one procedure calls on another (including recursively called procedures) data and machine conditions such as contents of index registers must be saved. These are stored in the segment, <stack>. The planned use of <stack> is described in the BD. 7 and BD. 9 sections of MSPM and is thoroughly discussed in Chapter 3 of this Guide.

The linkage segment contains certain vital symbolic data, descriptive information, pointers, and instructions which are needed for the linking of procedures in each process. More will be said about the linkage segment in Chapter 2 of this Guide (in connection with the commentary on the BD. 7 sections of MSPM). Most of the data segments (other than <stack>) that a procedure needs to refer to are referenced indirectly via special intersegment pointers placed in the linkage segment. These pointers are called "its" or "itb" pairs and are discussed in Section 1.3.9. Basically they permit formation of a final effective address for the desired data element without further need of the base address pairs.

The tentative conclusion we can make here is that a typical process, grinding away on a processor under the plan for the Multics system, will find itself making direct references to data and instructions from a relatively small number of different segments over relatively large time spans.

The segments are:

(1) One or more independent data segments.
(2) The procedure segment currently being executed.
(3) The linkage segment for the procedure currently being executed.
(4) The one stack segment for the process containing the perishable data of the process.

Now we begin to see how the paired address base registers are typically employed. By controlling their contents, pointers may be set to 4 different entries in the descriptor segment besides the currently executing procedure segment which is pointed at by the pbr. During the execution of one procedure, the frequency with which it will be necessary to reset any of these 4 paired "base" pointers may be low.

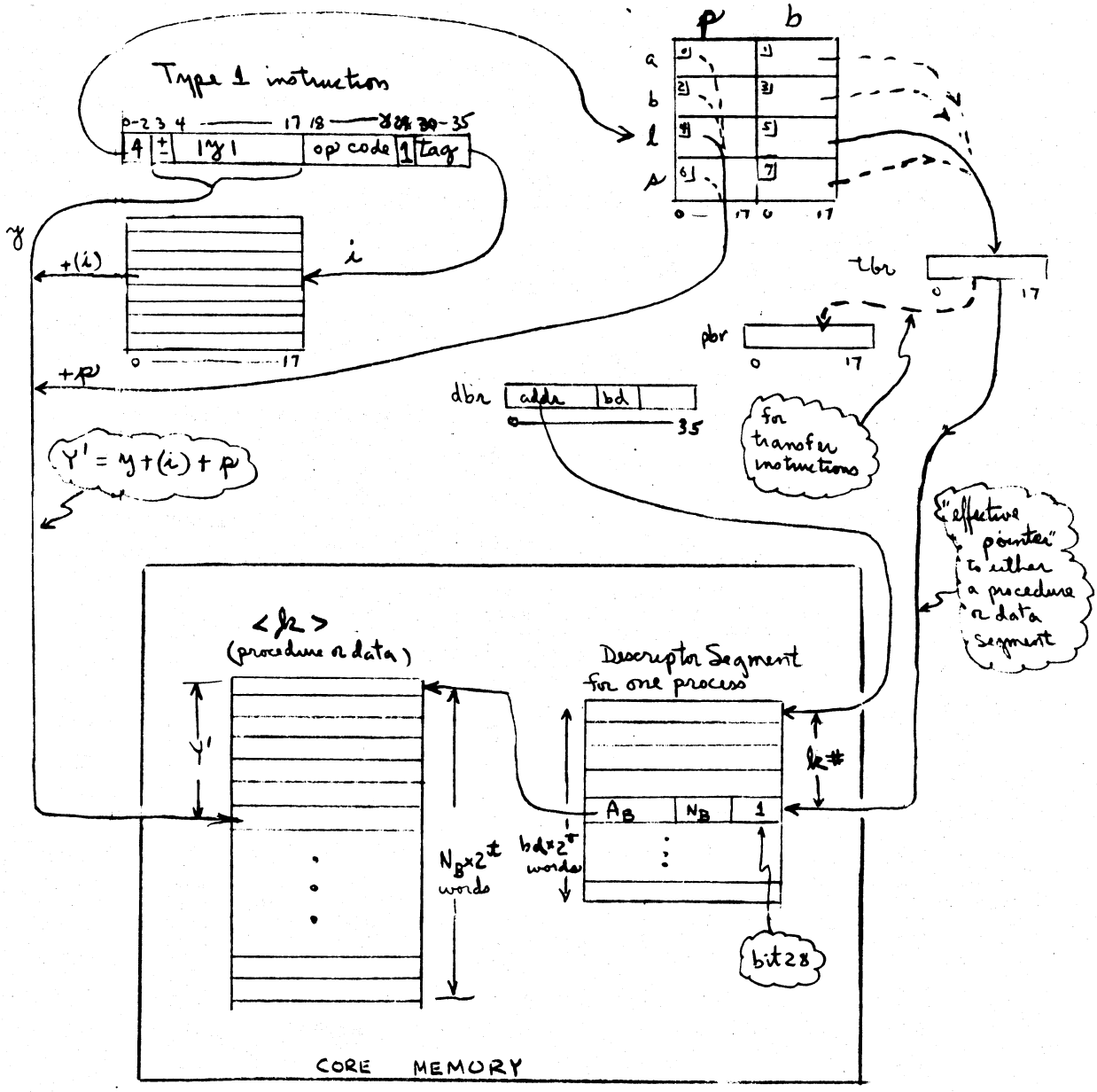
1.3.6 Summarizing Direct Address Formation

We now summarize all of our discussions so far on address formations. Note the following points:

(1) To fetch an instruction (instruction cycle), the (pbr) becomes the effective pointer. From this we obtain the address of the desired procedure segment. To this is added the (ic).
(2) To fetch a data word (execute cycle of a type 1 instruction), we illustrate with Figure 1-11(a). The effective pointer to the required descriptor word is obtained from the (tbr). The value of the tbr is copied from the current contents of one of the base registers ab, bb, ℓb or sb, depending on the value of the segment tag in the current instruction, being 0, 2, 4, or 6 respectively. The effective internal address, hereafter called Y', is formed as

$$Y' = y + (i) + p$$

where p is the current contents of ap, bp, ℓp, or sp, according as the segment tag has the value 0, 2, 4, or 6 respectively.
(3) The dashed line indicating actual data flow from the tbr to the pbr will be explained in the following discussion.



(For simplicity we imagine that segment <k> is not paged. This would be reflected in the fact that bit 28 of the descriptor word for <k> = 1.)

Figure 1-11(a). Address Formation in the Execute Cycle of Type 1 Instructions

1.3.7 Transferring Control to Another Procedure Segment

We consider here how a transfer instruction to a another procedure segment using a type 1 instruction is accomplished. If a new procedure segment is to be executed, there must be a new setting for the pbr so that hereafter it points to the new procedure segment. Let us consider what happens when procedure segment <a> executes any successful transfer instruction (of type 1). The effective address of a tra type instruction in most contemporary computers like the IBM 7094 simply replaces the value currently in the ic. In the GE 645, two things happen. (1) the effective internal address Y' = y + (i) + p replaces the contents of the ic, and, (2) at the same time the effective pointer, b, which is brought to the tbr, is then copied into the pbr. In other words

(1) $\quad ic \leftarrow y + (i) + p$

and

(2a) $\quad tbr \leftarrow b$

(2b) $\quad pbr \leftarrow b$

Now then, if b points to the descriptor word for <a>, a tra within the same segment will be achieved. But, if b points to a different descriptor word, say for segment <t>, then the transfer to <t> is automatically achieved, because t#, which is really what b amounts to, will have been placed in the pbr.

The next instruction executed will then be fetched from <t> at location of word zero of <t> + (ic), by virtue of having altered the contents of the pbr.

1.3.8 Address Formation for Type 0 Instructions

Any time an executing procedure segment attempts to make a self reference, formation of the data address can be greatly simplified, since the effective pointer is already known to be in the pbr. To make it possible

for the computer to recognize this simple case, we use a type 0 instruction*. When such an instruction is being analyzed for execution, recognition is made by virtue of bit 29 = 0.

The effective pointer is copied into the tbr from the pbr, i.e.,

$$tbr \leftarrow (pbr)$$

The effective internal address is made up of only two components. (You should look back at Figure 1-6). The two components are $\tilde{y}$ and (i), where $\tilde{y}$ is the 18 bit address (bits 0-17) of the instruction.

$$Y' = \tilde{y} + (i).$$

This type of address formation is shown in Figure 1-11(b).

If the type 0 instruction is a transfer instruction, the the net effect is:

(1) $ic \leftarrow \tilde{y} + (i)$

and

(2) no change in the contents of pbr.

1.3.9 Multi-Level Indirect Addressing (RI type) and Its Restrictions

The basic form of indirect addressing in the GE 645 is inherited from the GE-635 circuitry. This is indirect addressing via the type RI tag, which

---

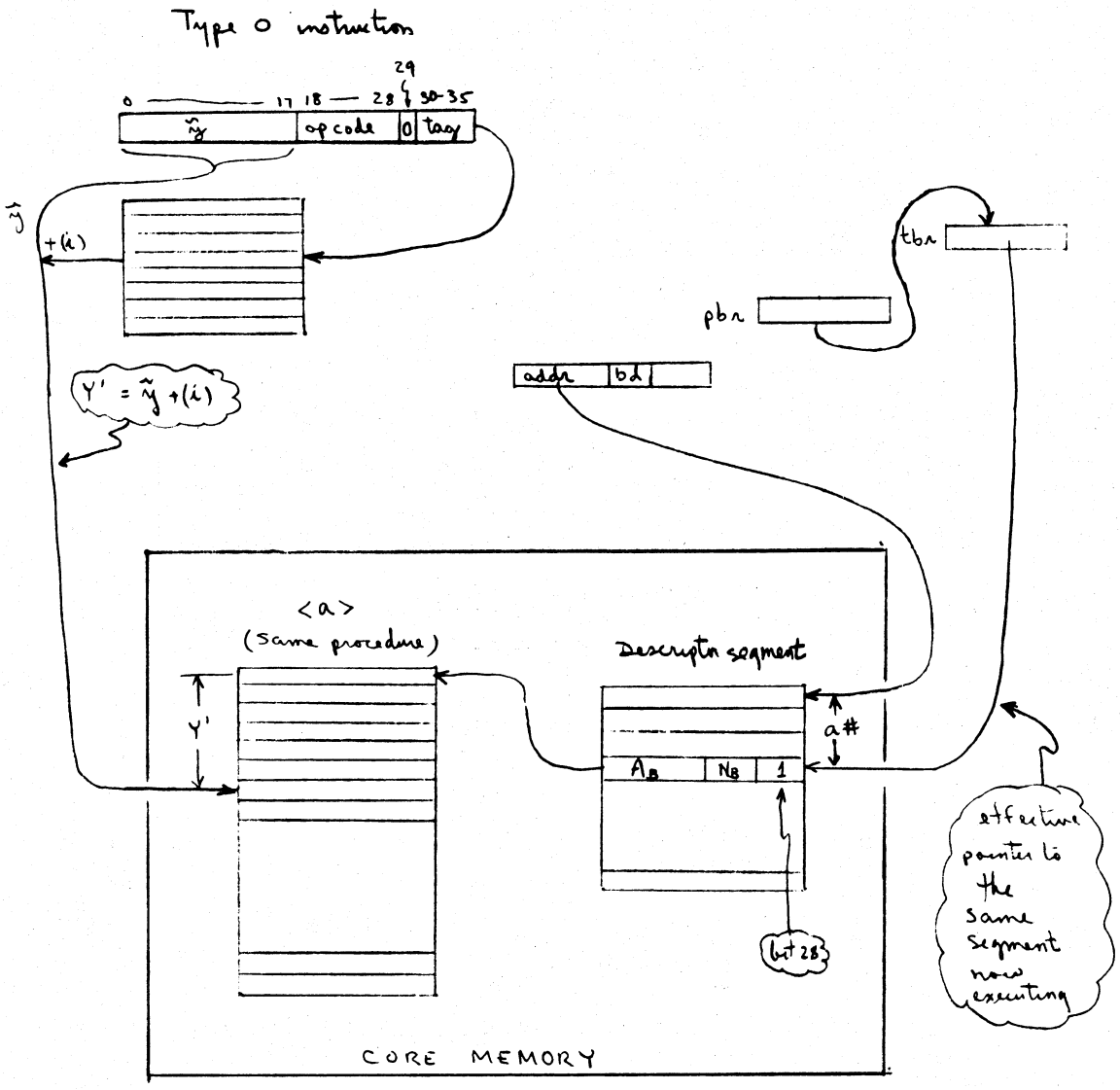
* If you were coding in a symbolic assembly language, the distinction between a type 1 and a type 0 instruction would be purely syntactical. For example, in the epl bsa assembly language, the variable field for a type 1 instruction always begins with a segment name or abr name, followed by a vertical bar. For a type 0 instruction, this component is missing.

Thus,

lda bp|6

is automatically recognized by the assembler as type 1 instruction. It means: load the accumulator with the data word located from a segment whose effective pointer is bb and effective internal address is 6 + bp.

The instruction lda 6, on the other hand, is recognized by the assembler as a type 0 instruction. It means load the accumulator with the data word found in location 6 of this procedure segment.



(For simplicity we imagine that segment <a> is not paged. This would be reflected in the fact bit 28 of the descriptor word for <a> = 1.)

Figure 1-11(b). Address Formation in the Execute Cycle of Type 0 Instructions

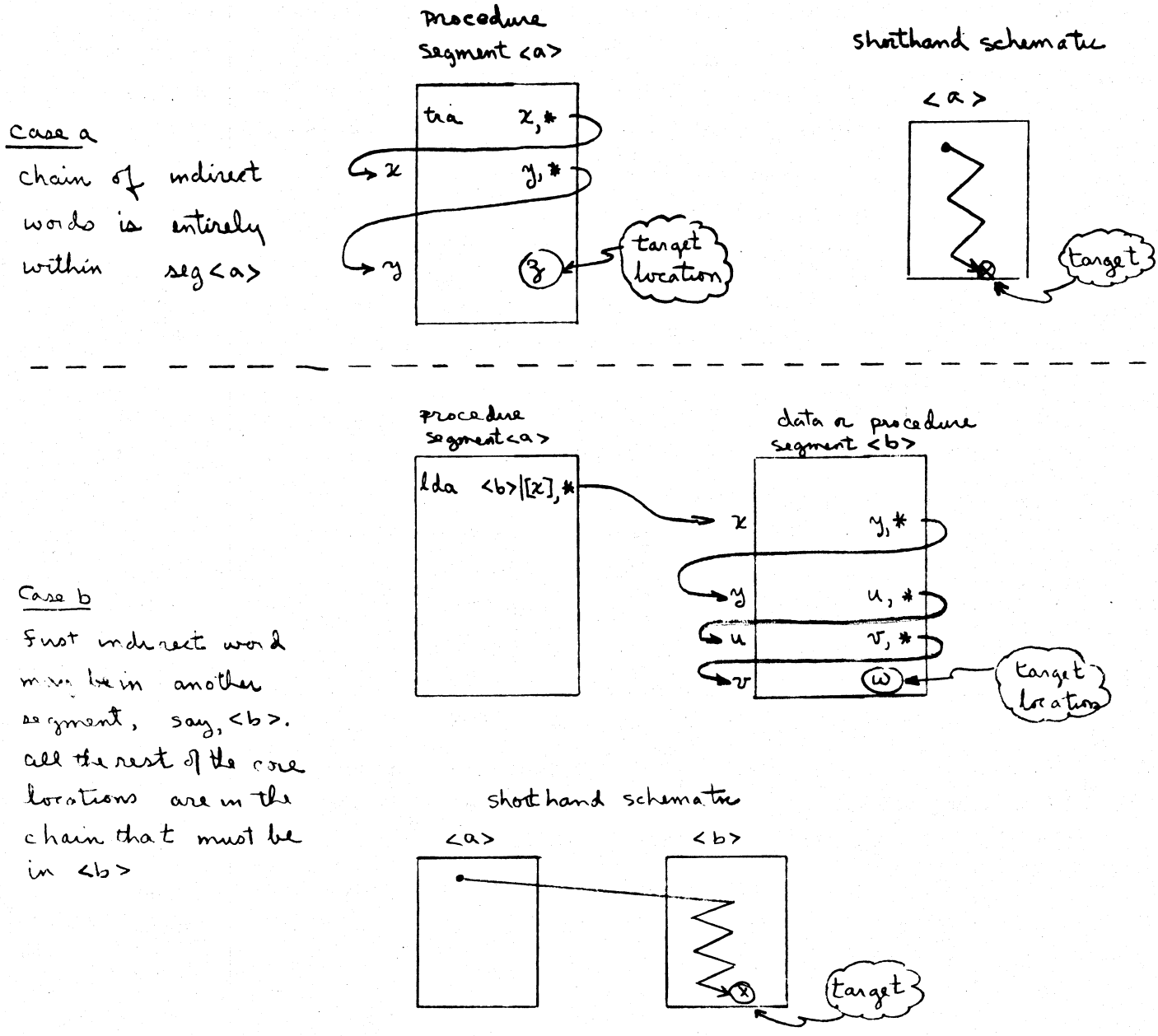
is sometimes called multi-level indexed indirect addressing. This is very similar to that of the IBM 7094 except indirect referencing continues to any number of levels instead of halting after two memory references. However, when a data word or transfer address is being fetched via a chain of one or more indirect words, the core address of each new indirect word, as well as the final effective address, is determined in essentially the same fashion as given in Figure 1-11(b).

In particular, only two fields of each indirect word are brought out of memory to be examined. These are the address field, $\tilde{y}$ (bit 0—17), and the modifier field (bits 30—35). To form the intended two-dimensional address that's coded in this indirect word, the effective pointer held in the tbr remains unchanged from its preceding value. The effective internal address is

$$Y = \tilde{y} + (i)$$

In other words, the address formation for the coding in an indirect word is handled much like that of a type 0 instruction in spite of the fact that the originating instruction might have been a type 1 instruction. The net effect is that no matter how many indexed indirect addresses are formed in one chain, the second and all succeeding addresses are treated as belonging to the segment pointed at by the first address in the chain. For example, if the originating instruction has an RI tag, and if its external base points to segment <b>, then the effective pointer for the next indirect word (and all others, if more than one) in the chain will remain internal to <b>. Figure 1-12 illustrates the two possible cases which can arise.

For fully general intersegment programming, it would be ideal if the indirect addressing mechanism at our disposal were such that we could cross from one segment to another as we go from one indirect word to the next. This capability is provided in GE 645 hardware as described in the next paragraphs.



The notation "tra x, *" and "lda <b>| [x], *", etc., is GE 645 assembly language (EPLBSA) notation. See G0012, or BE 7.04 for more details.

Figure 1-12. Two Cases of Multi-Level Intra-Segment Indirect Addressing (RI Type)

1.3.10 Indirect Word Pairs or Generalized Addresses

This hardware facility allows us to initiate indirect addressing from say seg <a> and arrive at segment <b> to find a special pair of indirect words (called either an "its" or an "itb" pair) whose content designates the core address for a point within an arbitrary segment. Recognition of its or itb pairs comes about as follows:

When the address of an indirect word is even the computer always fetches a pair of words (much like the 7094). Whenever a pair of indirect words are fetched the modifier bits of the first word (bits 30–35) are examined for the possibility of the its type modifier (octal 43) or the itb type (octal 41).

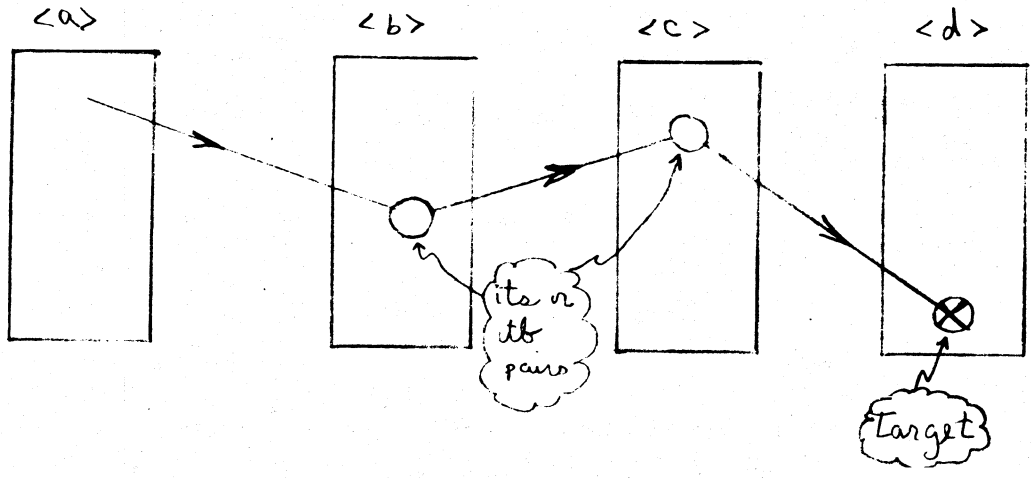
Understanding the role of the its pair will be critically important to most subsystems writers. Occasional but important use will also be found for itb pair. Details of the its or itb pair will be described shortly. The essential point is that such a pair designates for the 645 an arbitrary segment and an internal address for that segment. The word pair may also indicate further indirection. In short, use of an its pair allows us to "travel" from one segment, through a second one, to a third segment, say <c>, and so on. This feature is illustrated schematically in Figure 1-13(a). As special case, an its or itb pair can refer to the containing segment as suggested in Figure 1-13(b).

If the fetched indirect word or word pair is not an its or itb type, but does suggest further indirect addressing, then such addressing continues within the segment established from the preceding fetch. This is illustrated in Figures 1-14(a) and (b).

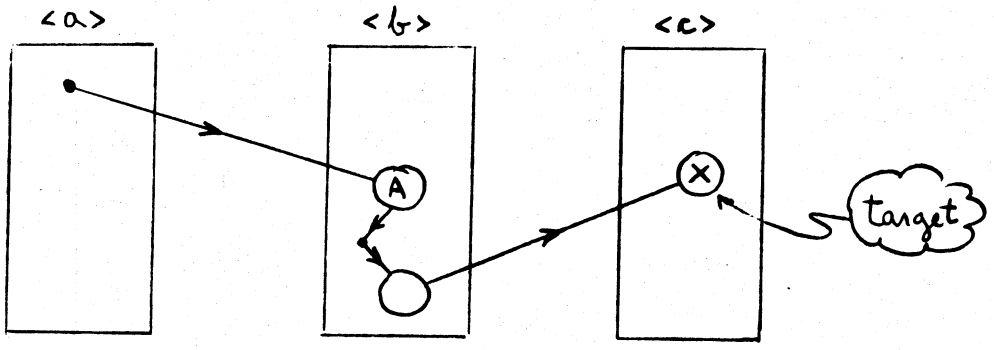
1.3.11 Details of its and itb Pairs

When the hardware has recognized one of these pairs, the format is then interpreted as shown in Figure 1-15. There are 4 fields of interest in each its or itb pair.

The second field of the first word is a six-bit identification code (bits 30–35). When this field has the octal value of 43 (its) or the octal value 41 (itb), the computer recognizes the containing word as the first word of an indirect word pair.

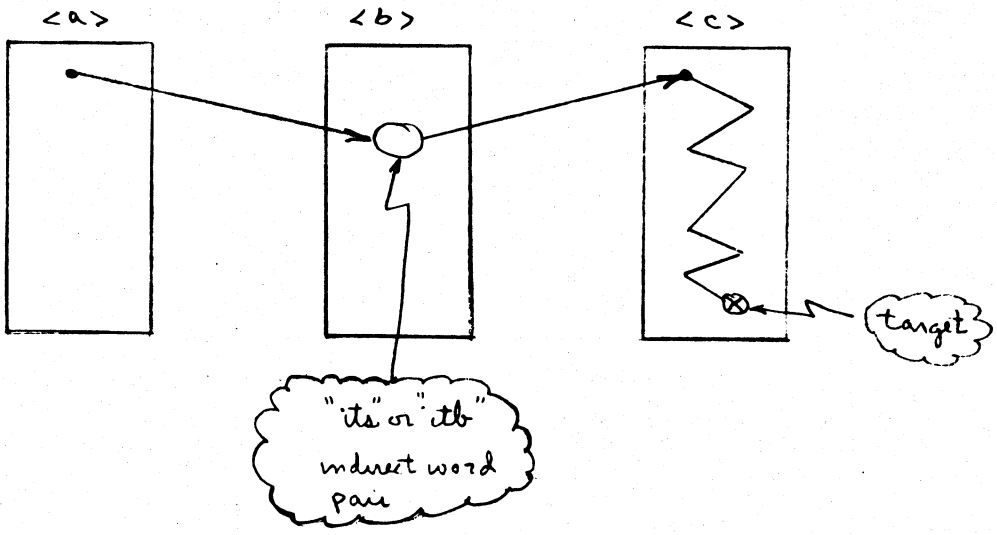


(a) Use of its or itb pairs.

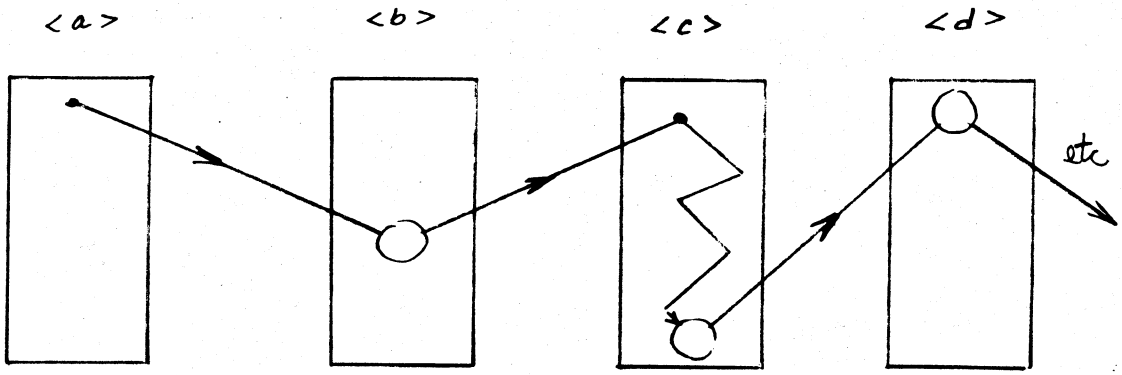


(b) Illustrating how its or itb pairs may refer to the same segment. The its pair marked A is an example.

Figure 1-13. Multi-Level Intersegment Indirect Addressing

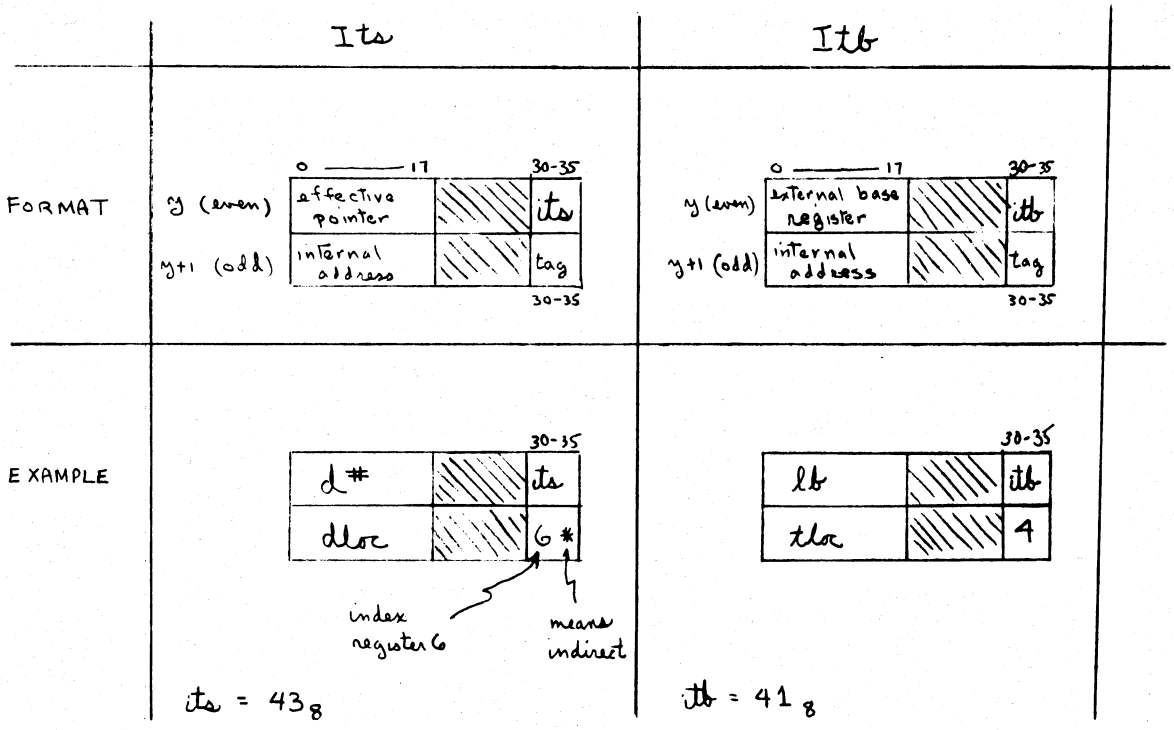


(a) Indirection may continue inside a third segment to the desired target without further use of its or itb pairs.



(b) It is quite permissible to keep the intersegment chain going if each its or itb pair points to an ordinary indirect word of the RI type. One of these can then point to another its or itb within the same segment. This its or itb could then allow a leap out to still another segment, etc.

Figure 1-14. Continuing the Indirect Intersegment Chain



its example:

directs GE 645 to fetch indirect word from d# | dloc + contents of index register 6

itb example:

directs GE 645 to fetch final word from lb | tloc + contents of index register 4

Figure 1-15. Format of its and itb Indirect Word Pairs

The second field of the second word is the standard GE 635/645 modifier or tag field (bits 30–35) as exhibited in Table 1-2. In the examples for this field we show an index register and, if indirect, a "*" following the register number. The full range of possible modifiers are available (see G0012, pages 16–19).

The first field of the second word is the 18–bit internal address within the desired segment that is designated in the first word.

The first field in the first word is either the effective pointer to the desired segment (for an its pair), or the external base register number which has been preset to hold the effective pointer to the desired segment (for an itb pair).

Figure 1-16 illustrates how address formation continues for an indirect its word pair. Figure 1-17 shows address formation for an itb word pair.

By way of summary we list below five types of address formation that have been discussed to this point.

	Type	Illustration
(1)	Instruction cycle	Figure 1-4
	Execute cycle	
(2)	Type 1 instruction	Figure 1-11(a)
(3)	Type 0 instruction	Figure 1-11(b)
	Indirect word pair	
(4)	its	Figure 1-16
(5)	itb	Figure 1-17

Figure 1-18 is an attempt to create a composite view of three of these, (1), (2), and (4), in order to see in one frame how each of the important registers plays its role. It may be worthwhile for the reader to superimpose the details of types (3) and (5) also. Different colors are recommended.

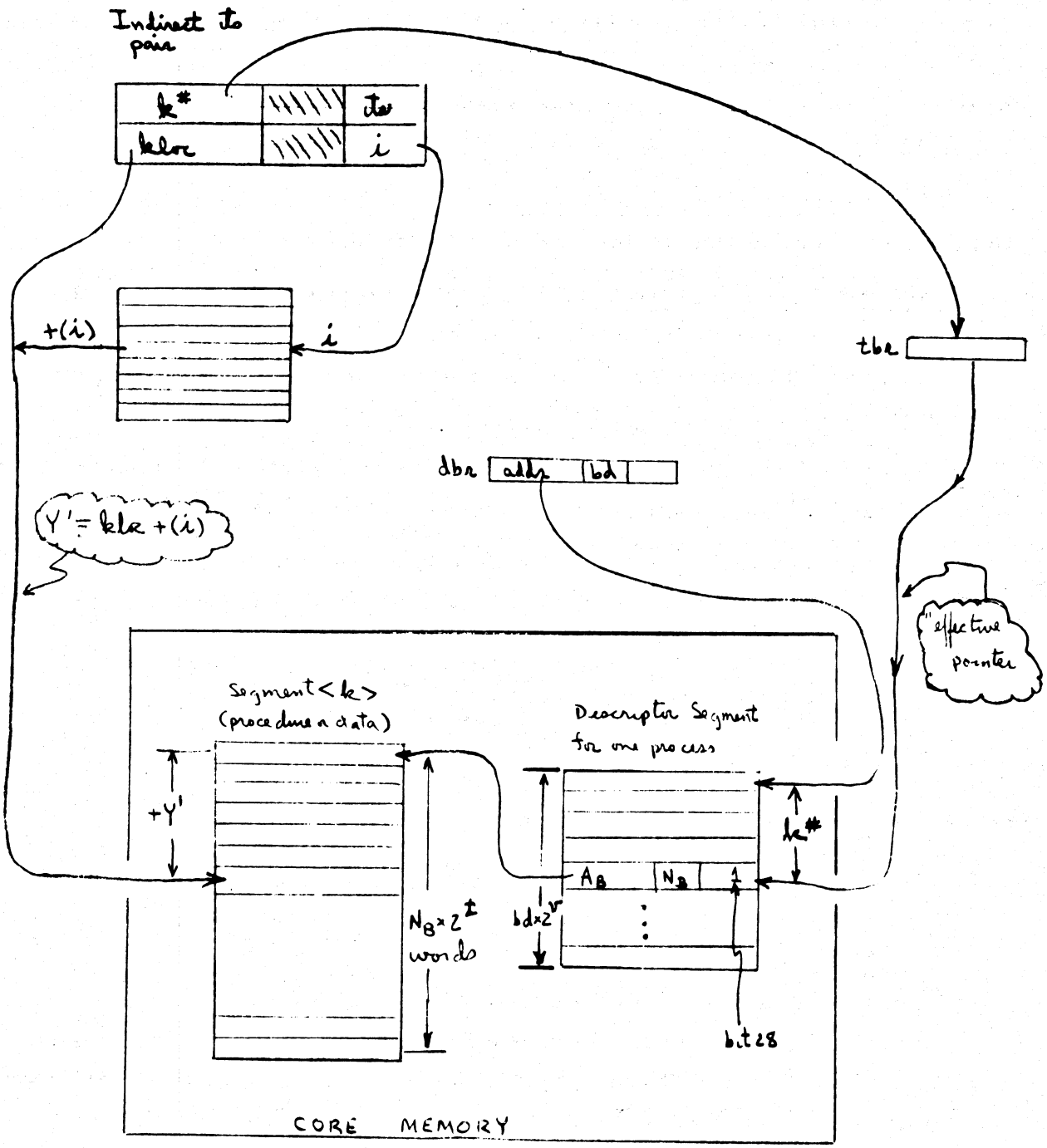


Figure 1-16. Address Formation for its Pairs

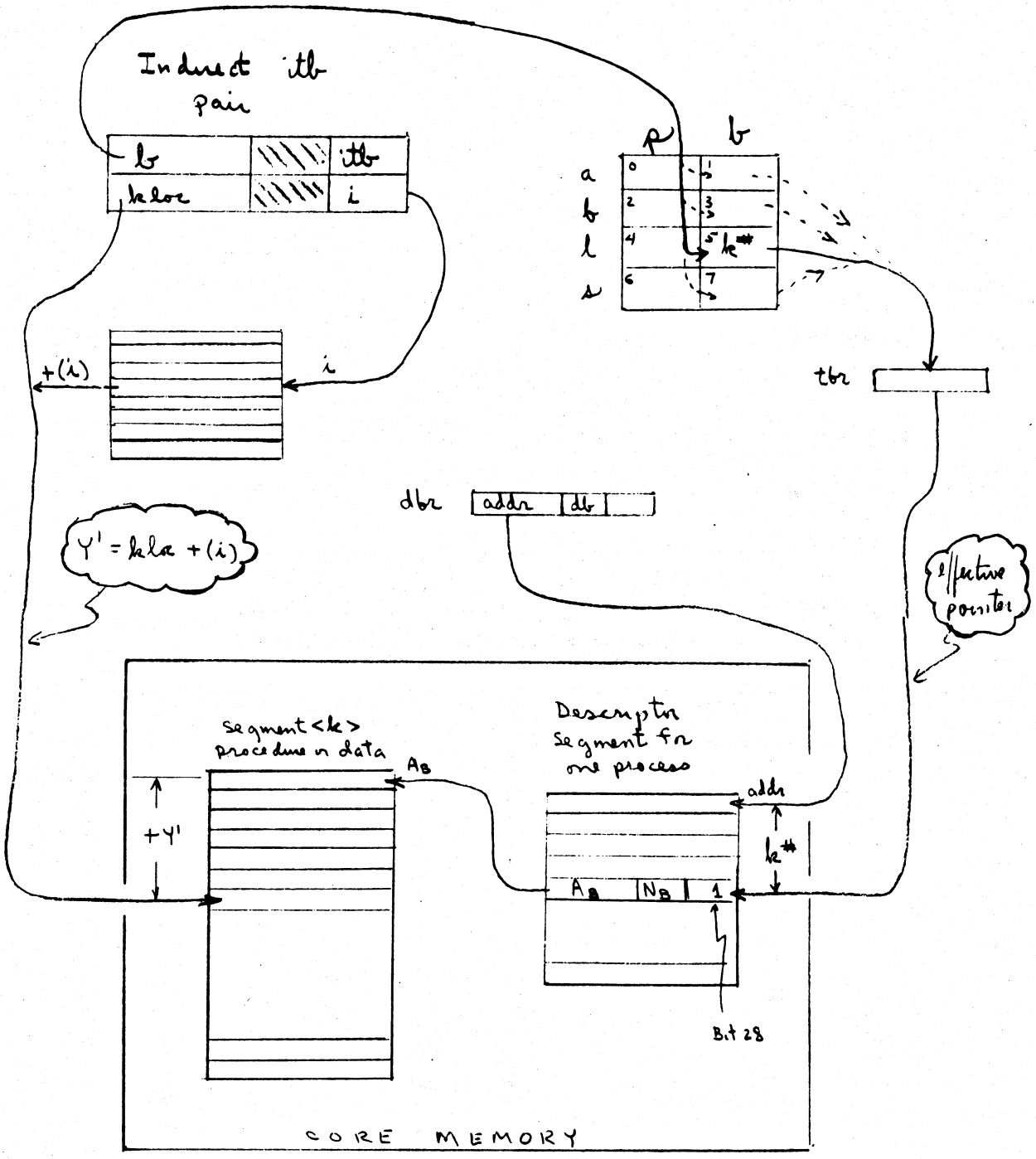
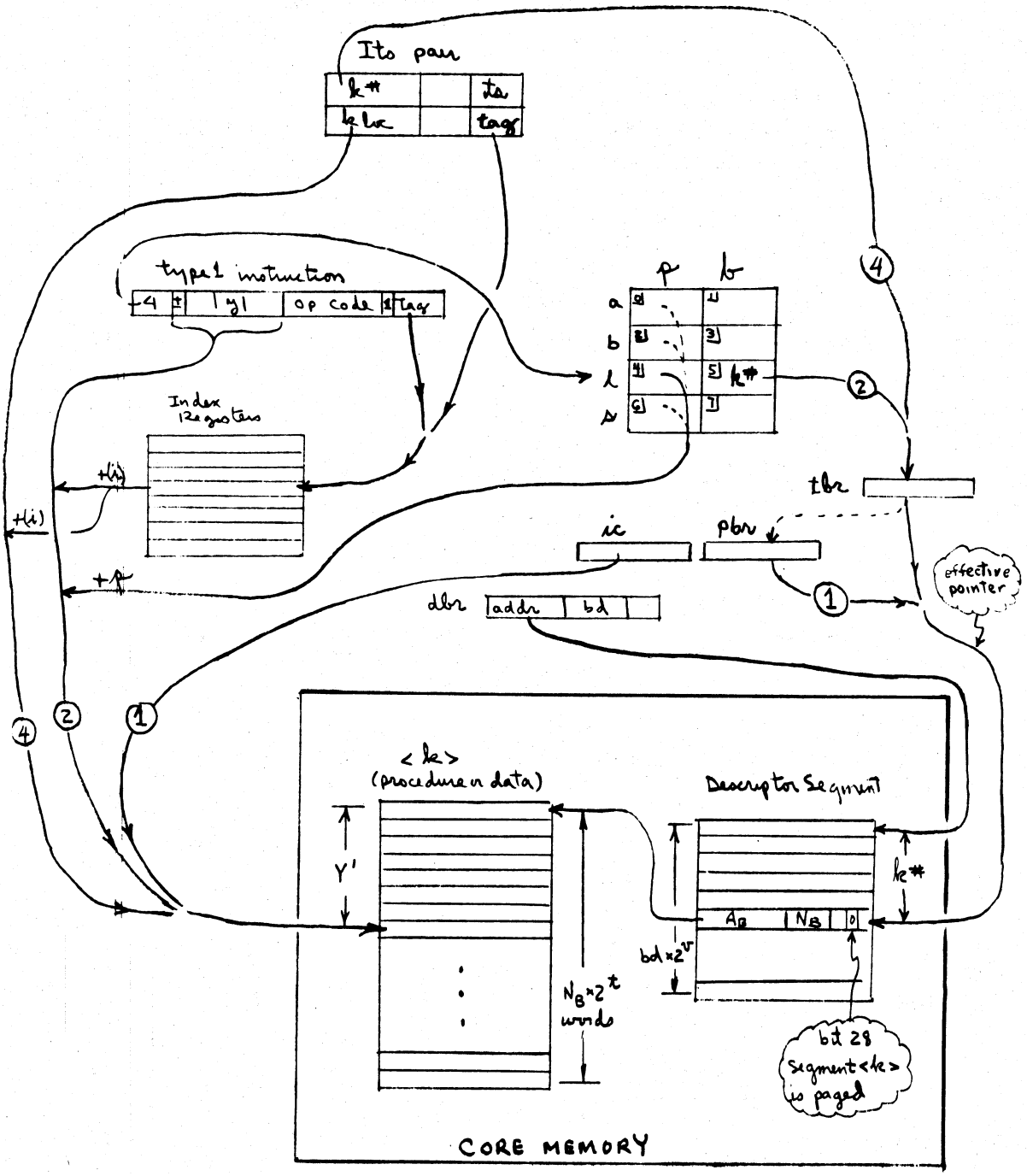


Figure 1-17. Address Formation for itb Pairs



(1) instruction cycle

(2) execute cycle - type 1 instruction

(4) execute cycle - its pair

Figure 1-18. Composite view of Three Types of Address Formation

1.4 SPECIAL INSTRUCTIONS TO MANIPULATE ADDRESS BASE REGISTERS

Easy manipulation of the contents of the eight address base registers in the GE 645 has thus far been implied. Here we enumerate some of the powerful instructions that are available to a subsystems writer to do the job. The most important ones are the eap and stp instructions.

	Symbol	Instruction
(1)	eapi	effective address to pair i i = 0, 2, 4, or 6

This very important instruction sets a pair of base registers specified by i. In Multics assembly language you actually use one of four symbolic op codes:

eapap instead of eap0,
eapbp instead of eap2,
eapℓp instead of eap4,
or eapsp instead of eap6,

The internal base is set from the effective internal address of the instruction (i.e., Y' = y + (i) + p). The external base is set from the effective pointer of the instruction.

The eapi instructions are especially useful in developing efficient code for the innermost loops of highly repetitive computations where one wishes to avoid indirect addressing. This concept will be further explored in the chapter on intersegment linking.

	Symbol	Instruction
(2)	stpi	store pair, meaning, store the contents of the i^{th} pair of registers (p and b) into a pair of memory words beginning at the specified core address which must be an even address. The format of the stored pair is:

Y (even)	b	0	its
Y + 1 (odd)	p	0	

In a Multics assembly language, you actually use one of four symbolic op codes:

stpap instead of stp0,
stpbp instead of stp2,
stplp instead of stp4,
or stpsp instead of stp6.

These two instructions are generally used in pairs. For example, to store and later restore the current values of the ab ← ap pair, you might write in assembly language:

```
stpap  hold + 6, 4
  .
  .
  .
eapap  hold + 6, 4*
```

Many examples of the use of these instructions will be found in the MSPM BD. 7. sections. See especially the save sequence in BD 7. 02. Also Chapter 3, p. 9.

	Symbol	Instruction action	
(3)	lbri	load base register i from specified core address	i = 0(1)7
(4)	sbri	store base register i into specified core address	
(5)	eabi	effective internal address to base register i the effective internal address of this instruction is assigned to address base register i	
(6)	adbi	add to base reg i from the specified address which is the effective internal address; i. e., from y + (i) + p. In slave mode i may be any register except sb	
(7)	ldb	in master mode load all 8 base registers from 8 successive memory words beginning at a specified core address which is (= 0 modulo 8). In slave mode, load 7 base registers, i. e., all but sb from 7 successive memory words beginning at the specified core address.	
(8)	stb	store all 8 base registers into 8 successive memory words beginning at a specified core address which is (= 0 modulo 8)	

The stb and ldb instructions are especially useful in call and return sequences because contents of the 8 bases along with the contents of the 8 index registers and other machine conditions (the (pbr) and the indicators) must be saved and restored in transferring to and returning from another procedure segment. These data are saved in a special segment given the standard name <stack>. Each process is automatically supplied by the supervisor with a unique <stack> segment.

A user is free to store the contents of all address base registers and is free to alter all but the sb base register, as previously suggested in Table 1-3. Only the supervisor needs to be able to set the sb register.

1.5 NOTES ON PAGING IN THE GE 645

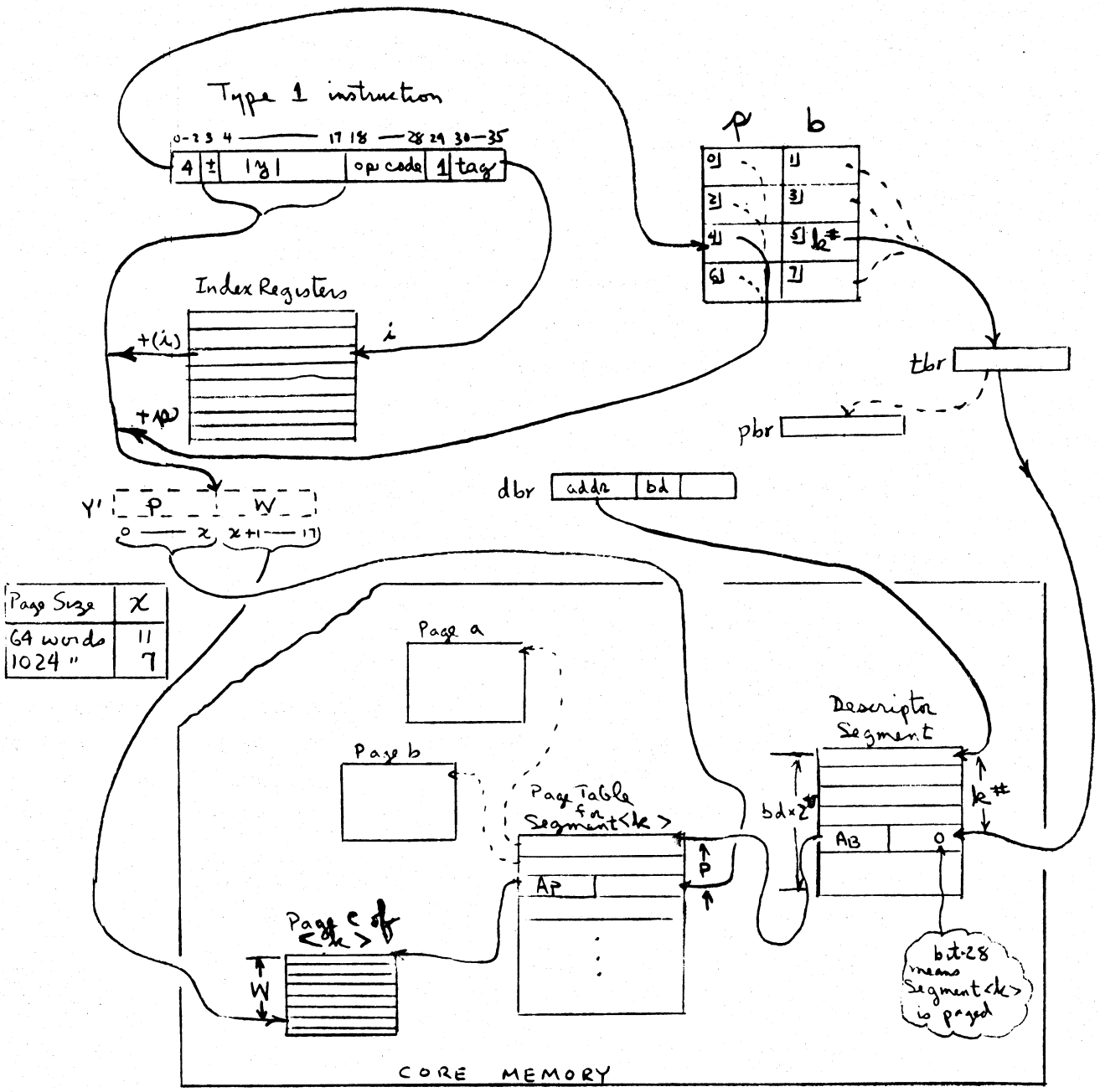
As mentioned earlier, segments are in fact always paged, that is, further subdivided into divisions called pages. The pages are stored in small blocks of memory, 64 words or 1024 words, wherever room can be found, by a supervisory routine called the Core Manager (see BG. 6). Neither the user nor the subsystems writer has control over the details of paging although he can give advice to the supervisor. In normal programming, even at the assembly language level he will not be concerned with paging. Moreover, there is no way that a user can detect paging.

In spite of this preamble on why paging need not be paid attention to, no red-blooded assembly language programmer will remain uncurious forever. Some details on the paging hardware may be found in G0029, especially pages 10–16. For the reader that must look, some figures are offered here which may help to picture address formation for a paged segment. These are Figures 1-19, 1-20, and 1-21, which are roughly the counterparts of Figures 1-11(a), 1-16, and 1-18, respectively.

Inspecting the latter figures, a segment descriptor word, or sdw, is now seen to point, not at word zero of the segment, but at word zero of the page table for that segment.

Each word of the page table serves as a page "descriptor word" or pdw, in the following sense:

1. If the associated page has been loaded in memory the pdw contains a pointer, A_p, to word zero of the page.
2. If the page is not present in memory, bits are set in the descriptor field of the pdw to indicate that the page is missing.
3. Access rights and use bits are also stored in the descriptor field.



Key to new notation:

P	means page number
W	means word number within the page
A_P	is address (times 2^{-6}) of the page

Figure 1-19. Address Formation for Type 1 Instructions (Paged mode)

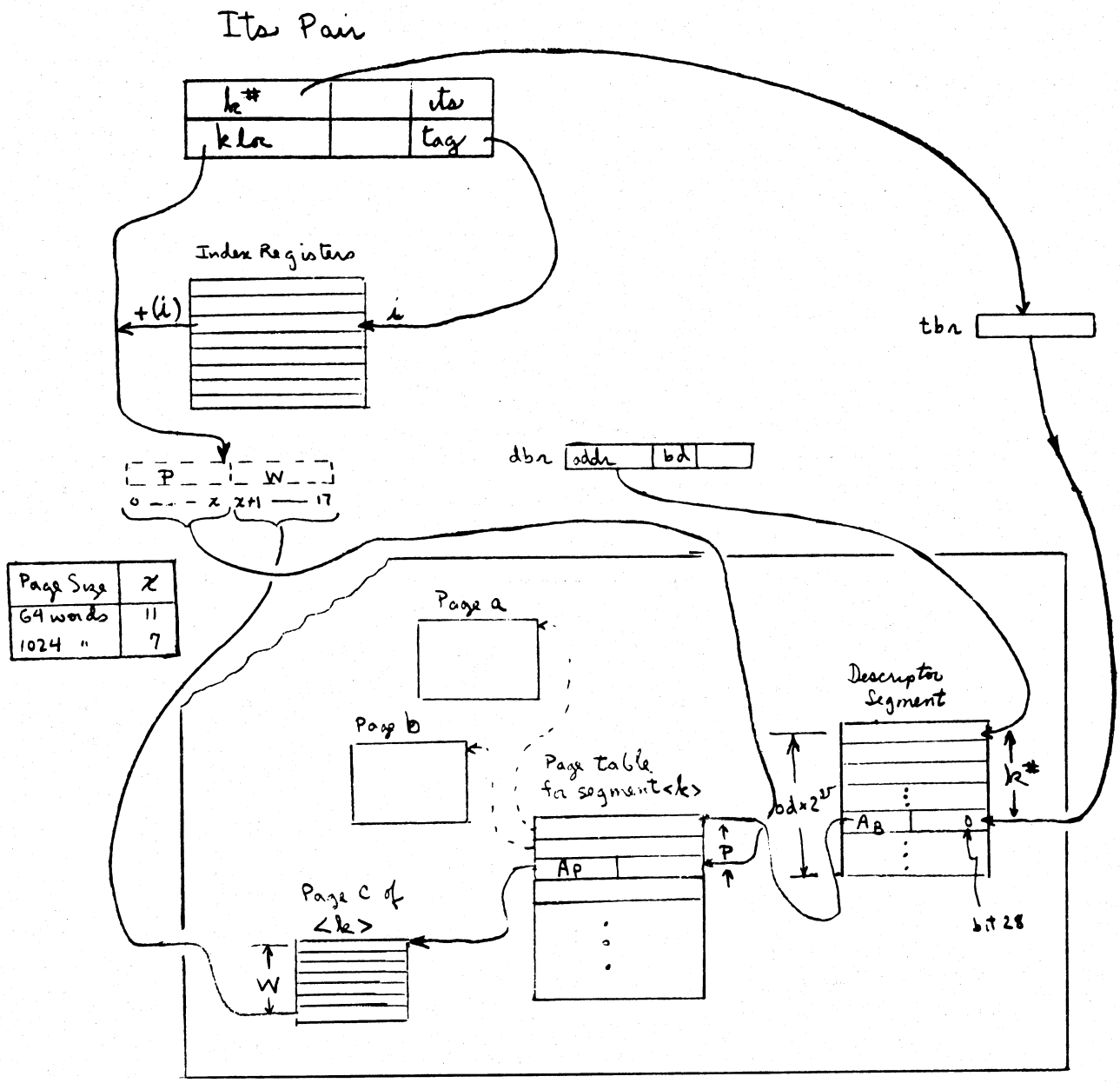
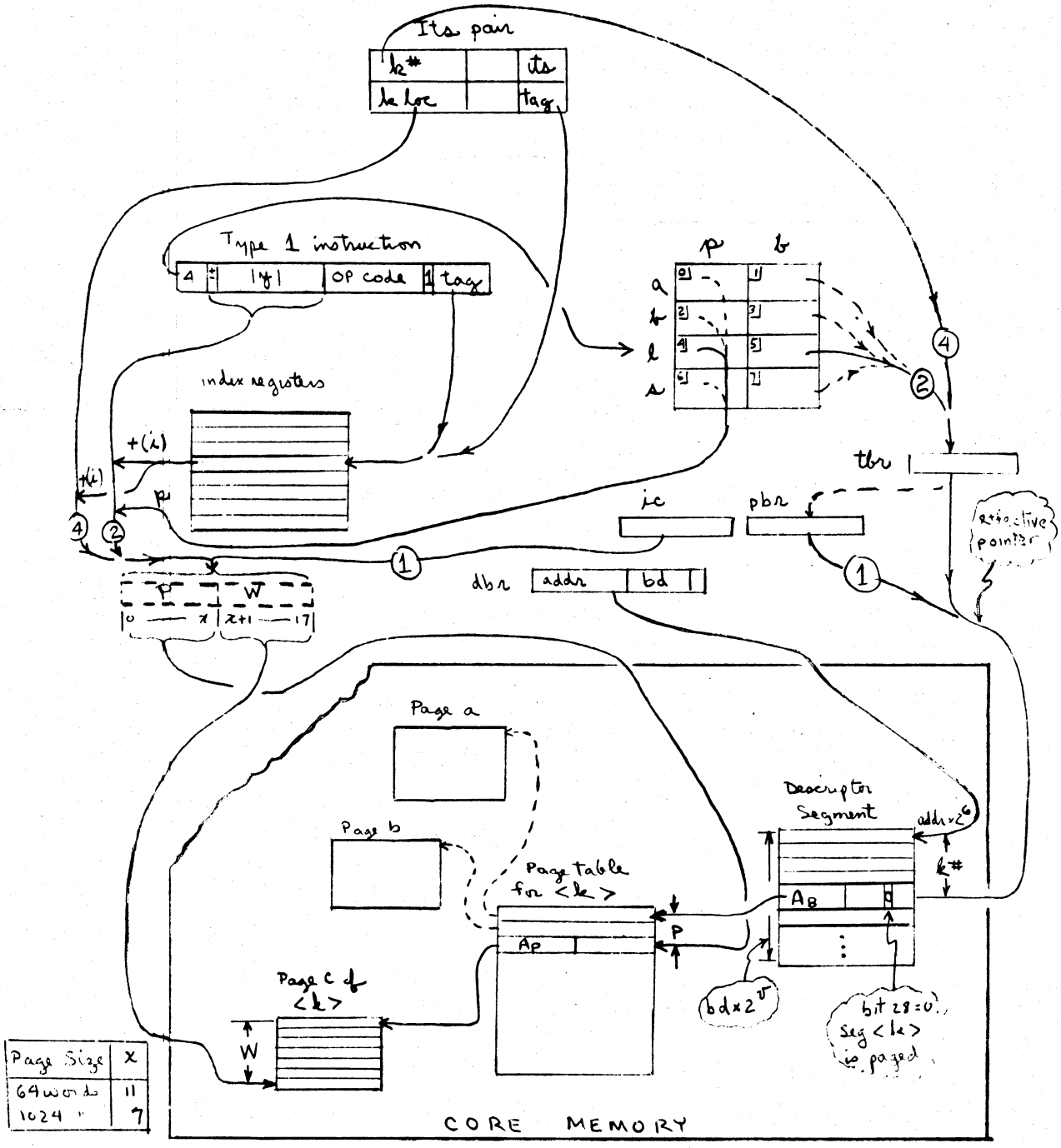


Figure 1-20. Address Formation for its Pairs (Paged mode)



(1) instruction cycle

(2) execute cycle - type 1 instruction

(4) execute cycle - its pair

Figure 1-21. Composite View of Address Formation (Paged mode)

The detailed use of the page table and its control information is a supervisory function and should not concern us. But the net effect is important. In particular, only those pages of a segment that are actually required during execution will be loaded. If a page is missing when referenced, a missing page fault will occur which will invoke a supervisory program that will create the missing page or load it from secondary storage.

In Figure 1-19 we see that the effective internal address, when formed, is split into two parts. The high order part serves as the page number, i.e., as an offset, P, within the page table. The low order part serves as the word number within the page, i.e., the offset W.

In Figure 1-20 we see how the internal component of the its pair is also split into the two parts P and W. Finally, in the composite, Figure 1-21, we note that the (ic) is also split the same way.

1.6 NOTES ON THE ASSOCIATIVE MEMORY ADDRESSING FACILITY

The elaborate address formation schemes thus far discussed or (in the case of paging) alluded to, are implemented efficiently in the hardware with the aid of a small Associative Memory (AM). This is a memory of z words, 59 bits each. The larger z, the more efficient the scheme. At present z = 16 in the Multics hardware. The operational details of the associative memory to speed up the address formation (and bounds checking) operations is, like paging, something that subsystems writers will never need to know about. Nevertheless, because it's new (and because it's complicated and a small challenge to understand) a red-blooded programmer will at one time or another take up the mountain climber's rallying cry — "Because it's there!" — and make the assault.

To help satisfy the too-early and too-curious, we shall first summarize the important concepts and then present some of the details using several figures and a very brief verbal description. Additional details may be found in G0029, pages 25–28.

Every reference to core memory ordinarily requires the use of a pointer A_B (in the segment descriptor word) or A_p (in the page descriptor

word) or both. Each time one of these descriptor words is needed, a copy is stored in the associative memory. Stored with each sdw or pdw will be the associated segment number. Naturally, since the AM is small, each new descriptor word that is entered calls for the discarding, by destructive read-in, of one already-resident descriptor word. A "fifo" (first-in-first-out) discipline is employed in deciding which word is discarded. (We shall describe how this is done momentarily.)

Suppose we now imagine the AM filled with an assortment of segment and page descriptor words and their associated segment numbers. Address formation in the GE 645 now takes on a "new look". Each time a value of A_B or A_p is needed, it is searched for in the associative memory using the effective pointer as the "input value". If found in the AM, then the respective memory cycles necessary to fetch these pointer values from the descriptor segment, or page table can be saved.

Basically, there are two possible "hits" that can occur in the search of the AM. Either the desired A_p is found or just the A_B is found. If neither value is found address formation as described in earlier sections of this chapter apply. That is, the necessary memory cycles must be taken to fetch first A_B and then A_p. If just A_B is found, then at least one cycle is saved because the page number portion, P, of the internal effective address will already be available. The pair (P, A_B) then determines the core address for the desired page table word holding A_p. If A_p is found in the AM, then two memory cycles can be saved. The A_p value can be paired with the word number portion of the effective internal address, W, which will have been simultaneously developed. The pair (W, A_p) then determines the desired core address which is really the principal target of the programmer's request.

Figure 1-22, suggests in a simplified way the functional value of the associative memory. The AM is searched once (one pass) when a segment in which a core address is being formed is not paged. The AM

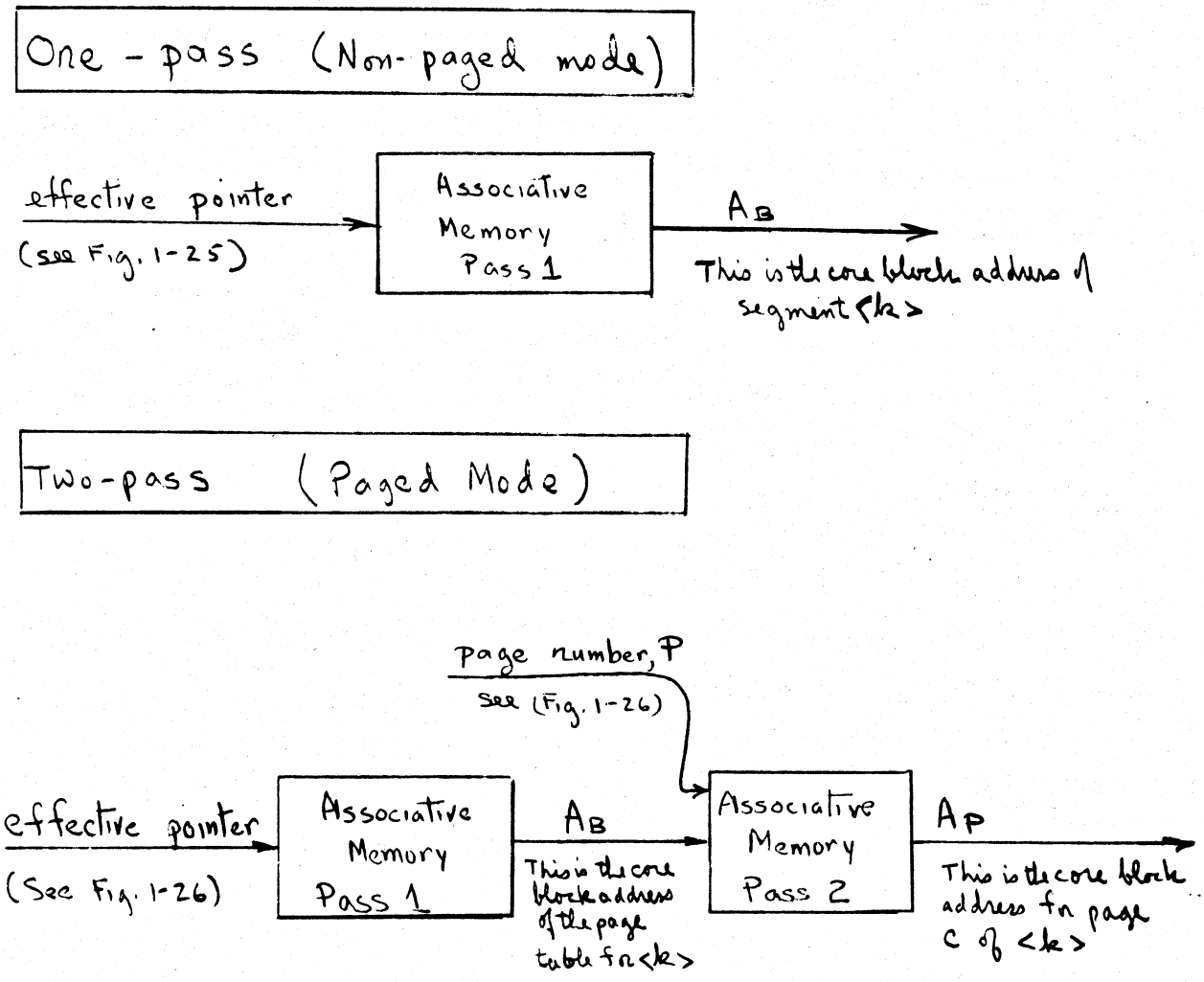


Figure 1-22. Functional Overview of the Associative Memory

is searched at least once when the segment is paged. We have called this more elaborate search the two-pass case.*

Figure 1-23 shows the format of the 59-bit words in the AM.

Figure 1-24 is a logic flow chart to show the how of the two-pass search of the AM. This figure and the discussion in the next section make reference to Figures 1-25 and 1-26. The latter two figures are the counterparts of Figures 1-18 and 1-21, respectively.

1.6.1 A "Walk" through the Associative Memory

In the following discussion, column numbers refer to those in Figure 1-23 and box numbers refer to the flow chart in Figure 1-24.

Pass 1: Locate an entry which satisfies these two conditions (box 1):

(1) Contents of Col 2 is a 1

(2) Contents of Col 3 matches the given effective pointer.

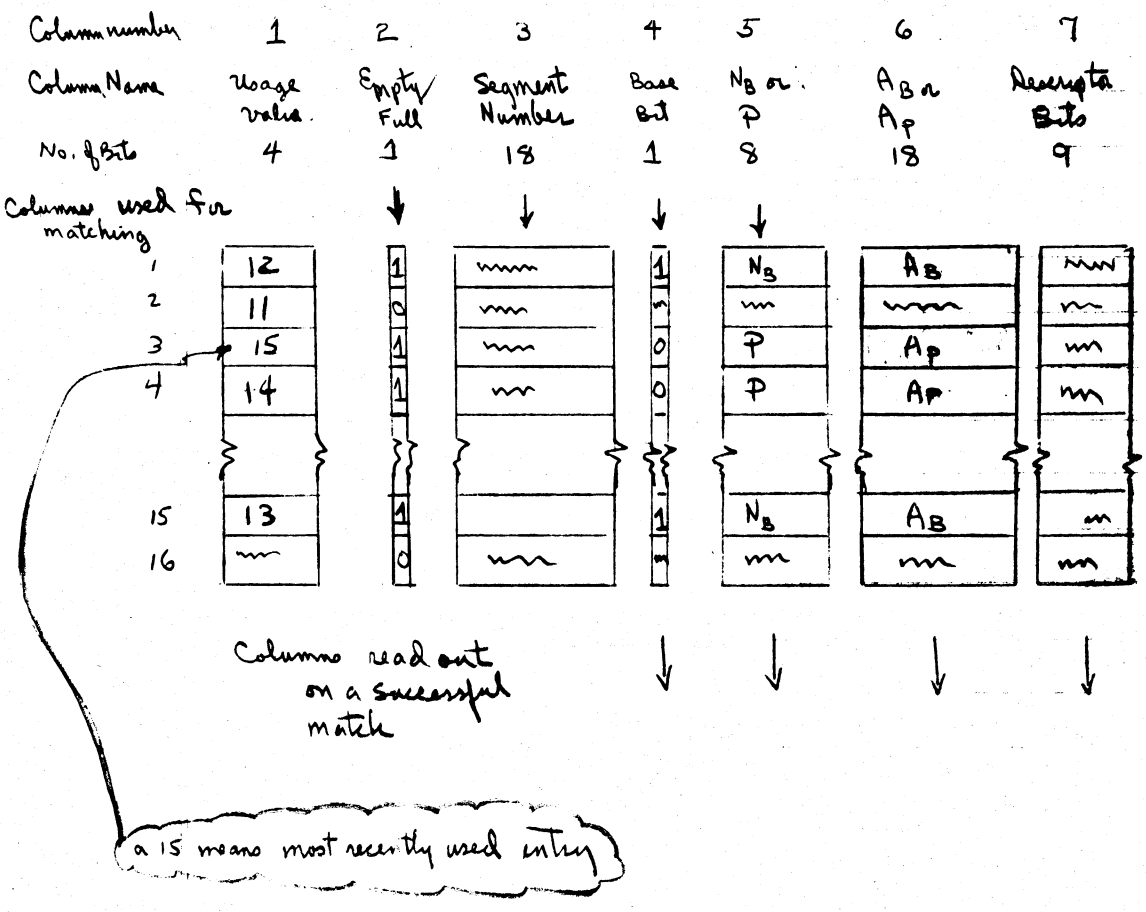
A failure signifies that no pertinent descriptor word resides in the AM. Route 3 (Figure 1-26) must therefore be taken. That is, the required descriptor words must be fetched either from the descriptor segment alone. (if the fetched descriptor word indicates no paging) or from both the descriptor segment and from the page table (if the fetched segment descriptor word shows the segment to be paged). Each fetched descriptor word is then inserted in the associative memory.

1.6.2 Inserting Descriptor Words into the Associative Memory

We digress momentarily to explain the rules for inserting descriptor words into the associative memory. Each inserted descriptor word replaces the one word whose usage value in Column 1 equals zero. Bits of the inserted descriptor word are placed into Columns (fields) 6, 5, and 7.

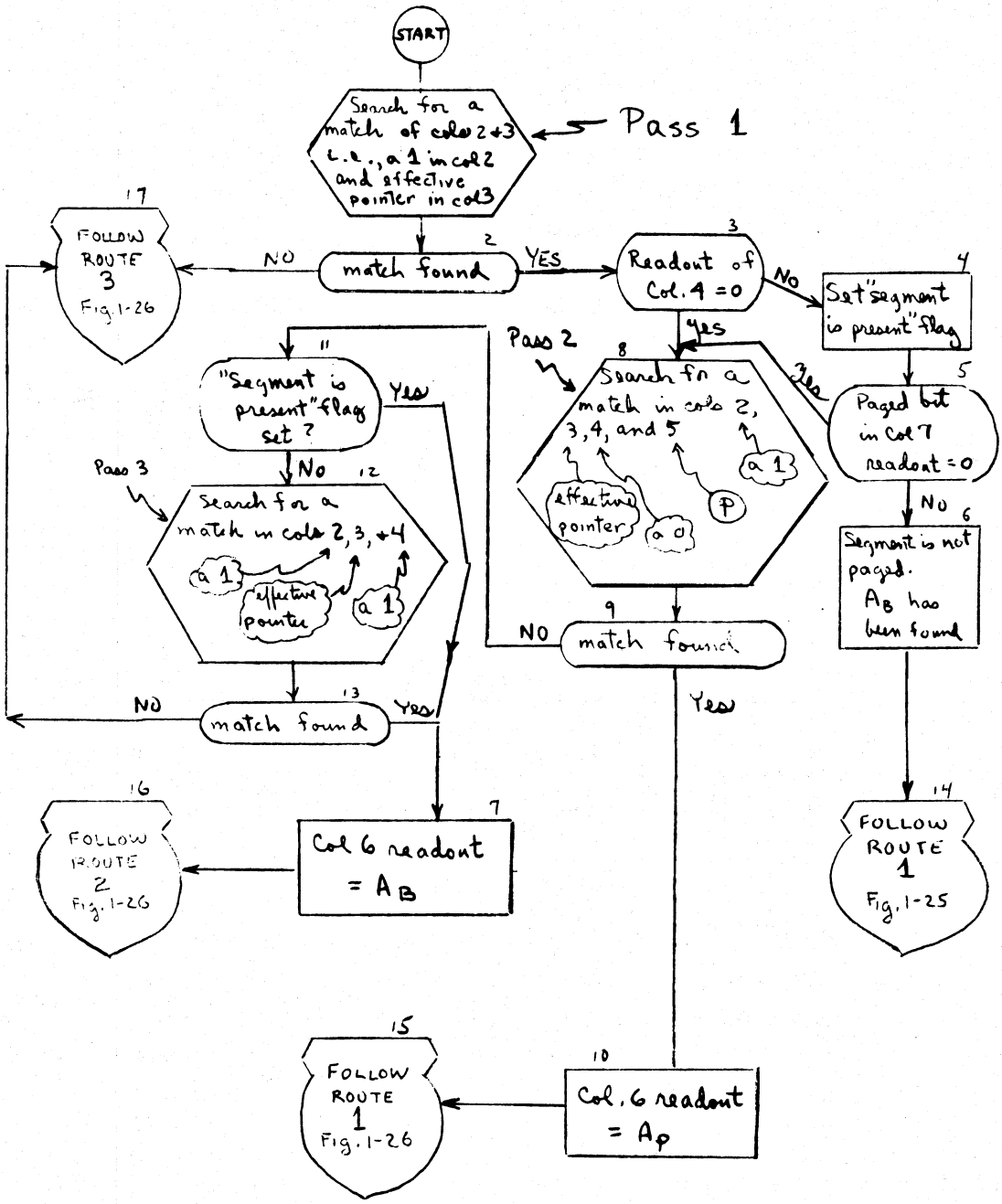
---

* Strictly speaking, in certain cases, up to three passes of the AM might have to be made before the GE 645 finds A_p or "abandons" the search. You will detect this third pass from a study Figure 1-24. Nevertheless, we prefer to call this a two-pass scheme in recognition of the fact that at least two passes are required before A_p can be determined.



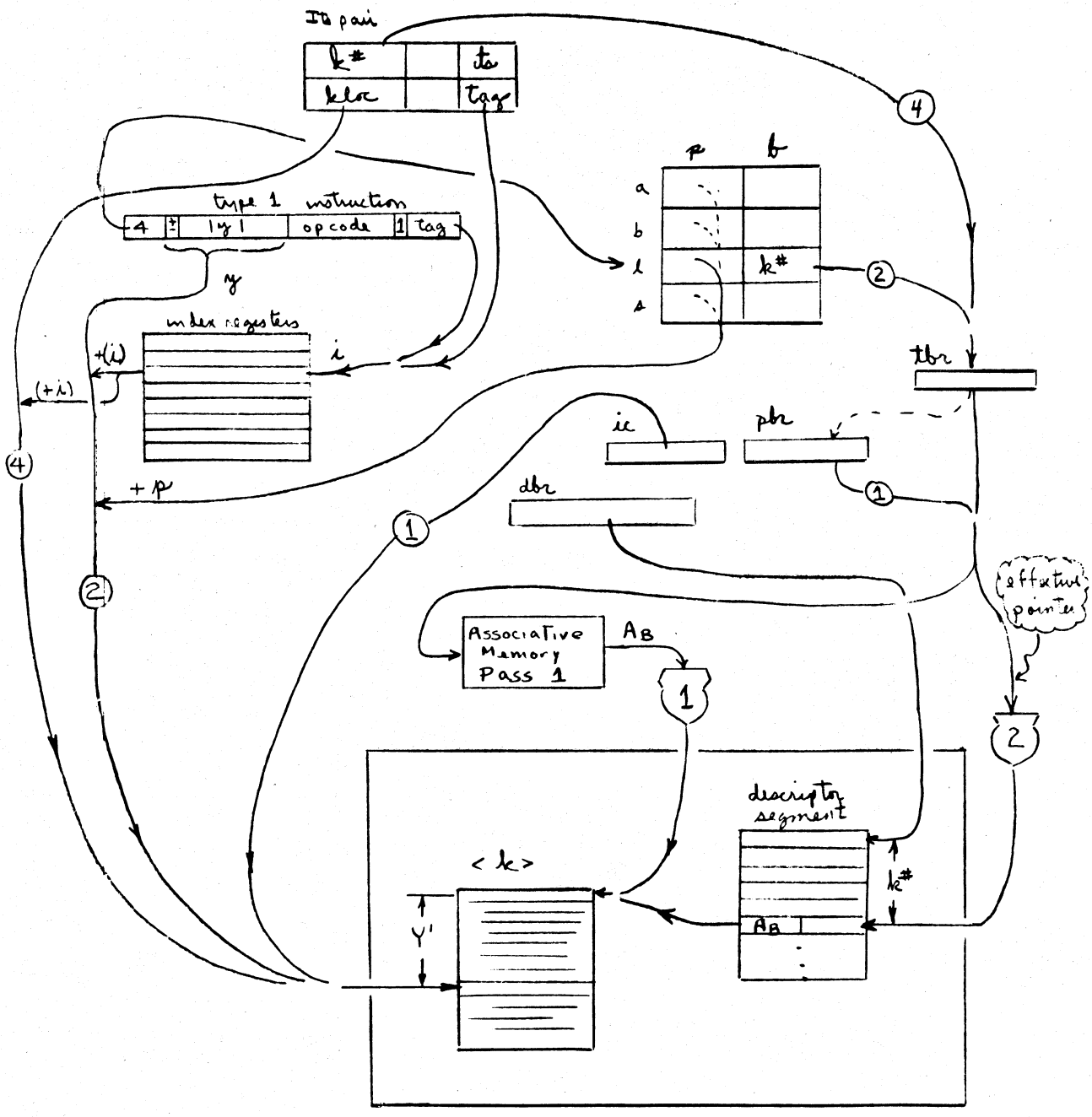
Note: There are 59 bits per entry.

Figure 1-23. Format of the 16 Associative Memory Entries



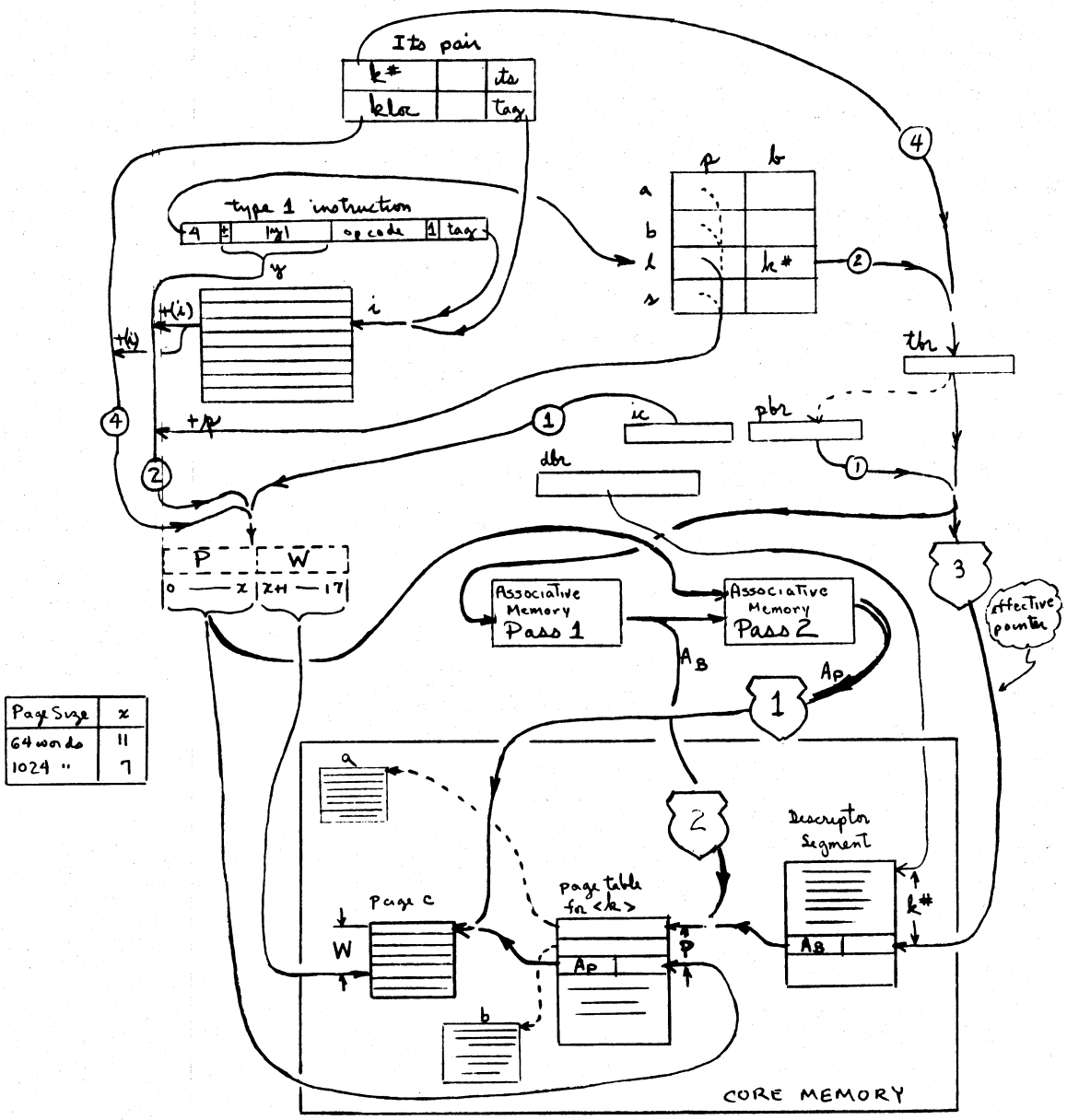
Note: When routes 2 or 3 are followed, there is a simultaneous (asynchronous task) adjustment of the associative memory to include an entry corresponding to the one not found in the pass that just failed.

Figure 1-24. Detailed Flow Chart Showing Associative Memory Action



This shows two routes to the block address of segment <k>. Route 1 through associative memory (Figure 1-22), and Route 2 through the descriptor segment mechanism.

Figure 1-25. Composite View of Three Types of Address Formation



We show two primary routes to the block address of page c of segment <k>. Route 1 is through the 2-stage associative memory mechanism. Route 3 is through the descriptor segment mechanism. Route 2 is an intermediate route; i.e., via the 1-stage associative memory to the page table for segment <k>.

Figure 1-26. Composite View of Three Types of Address Formation

The base bit (column 4) is set to 1 if an sdw (a segment descriptor word) is being inserted, and is set to 0 if a pdw (page descriptor word) is being added. The empty/full bit (col 2) is set to 1, and column 3 is set to the segment number associated with the inserted descriptor word. Lastly, the usage value for this word (column 2) is set to 15, and the usage value of every other word in the associative memory is decreased by 1. This last ritual guarantees the AM's fifo discipline. That is to say, each new insertion is made at the expense of the oldest resident.

A success (in pass 1) causes a "readout" of the information in columns 4, 5, 6 and 7 of the matching entry. The descriptor word just found may be an entry for another page of the same segment; it may be for the pdw we want, it may be the sdw for the referenced segment. The first step to resolve this three-way ambiguity is to decide whether the matched entry is an sdw or a pdw by checking the value of the column 4 readout (box 3). If a pdw has been found, we know the segment under consideration must be paged, so we now "gamble" and make a second search of the AM looking for the desired page table word (box 8). We'll pursue this thread at the paragraph marked Pass 2. If on the other hand an sdw was found, we've definitely made progress because we know that the column 6 readout is A_B for the referenced segment. So we set a flag to indicate this success (box 4).

Next we check to see if the descriptor bits of the discovered sdw indicate that the segment is paged (box 5). A no means success (box 6). We've found in one pass the desired value of A_B which locates the desired non-paged segment. We now follow Route 1 of Figure 1-25. A yes means the segment is paged, so another search of the associative memory should be made in hope of finding the desired pdw (box 8).

Pass 2: For this search we input both the effective pointer and the page number P. We try for a match on columns 2, 3, 4 and 5. That is we look for an entry that is "full" (column 2 = 1), that represents a pdw (column 4 = 0), and that has the desired segment number (column 3) and page number (column 5).

Success on this second search means the desired page address A_p is the column 6 readout (box 10). We now follow Route 1 of Figure 1-26.

Failure on this second search search prompts us to check if the "segment is present" flag was set earlier (box 11).

If yes, the column 6 readout in the first search was the desired A_B for the referenced (paged) segment. We follow Route 2 of Figure 1-26.

If no, it means that our first pass success resulted from the discovery of the wrong pdw for the right segment. Furthermore, our failure in the second pass means that the right pdw does not now reside in the associative memory. We now make a final search this time looking only for a match on a "full" entry that has its base bit (column 4) equal to 1, signifying that we will this time restrict the search to sdw's only. The input for this search (box 12) is the effective pointer to be matched by the contents of column 3.

If no match is found after these three passes we throw in the towel. We must follow Route 3, Figure 1-26. If a success, it means we've found A_B as the readout of column 6 and we can now follow Route 2, Figure 1-26.