Reprinted from
AFIPS
Conference Proceedings
Volume 32, 1968

Burroughs' B6500/B7500 stack mechanism

by E. A. HAUCK and B. A. DENT
Burroughs Corporation
Pasadena, California

INTRODUCTION

Burroughs' B6500/B7500 system structure and philosophy are an extension of the concepts employed in the development of the B5500 system. The unique features, common to both hardware systems, are that they have been designed to operate under the control of an executive program (MCP) and are to be programmed in only higher level languages (e.g., ALGOL, COBOL, and FORTRAN). Through a close integration of the software and hardware disciplines, a machine organization has been developed which permits the compilation of efficient machine code and which is addressed to the solution of problems associated with multiprogramming, multiprocessing and time sharing.

Some of the important features provided by the B6500/B7500 system are dynamic storage allocation, re-entrant programming, recursive procedure facilities, a tree structured stack organization, memory protection and an efficient interrupt system. A comprehensive stack mechanism is the basic ingredient of the B6500/B7500 system for providing these features.

B6500/B7500 processor

The command structure of the B6500/B7500 Processor is Polish string, which allows for the separation of program code and data addresses. The basic machine instruction is called an operator syllable. This operator syllable is variable in length, from a minimum of 8 bits to a maximum of 96 bits. In the interest of code compactness, more frequently used operator syllables are encoded in the 8 bit form.

The Processor is provided with a hardware implemented stack in which to manipulate data and store dynamic program history. Also, data may be located in arrays outside the stack and may be brought to the stack temporarily for processing. Program parameters, local variables, references to program procedures and data arrays are normally stored within the stack.

The data word of the B6500/B7500 Processor is 51 bits long. Data are transferred between memory and within the Processor in 51 bit words. The first 3 bits of the word are used as tag bits, which serve to identify the various word types as illustrated in Fig. 1. The remaining 48 bits are data. Tag bits, in addition to identifying word type, provide the B6500/B7500 Processor with two unique features: (1) data may be referenced as an operand, with the processor worrying about whether the operand consists of one or two words, and (2) system integrity and memory protection are extended to the level of the basic machine data words. If a job attempts to execute data as program code, or to modify program code, the system is interrupted.

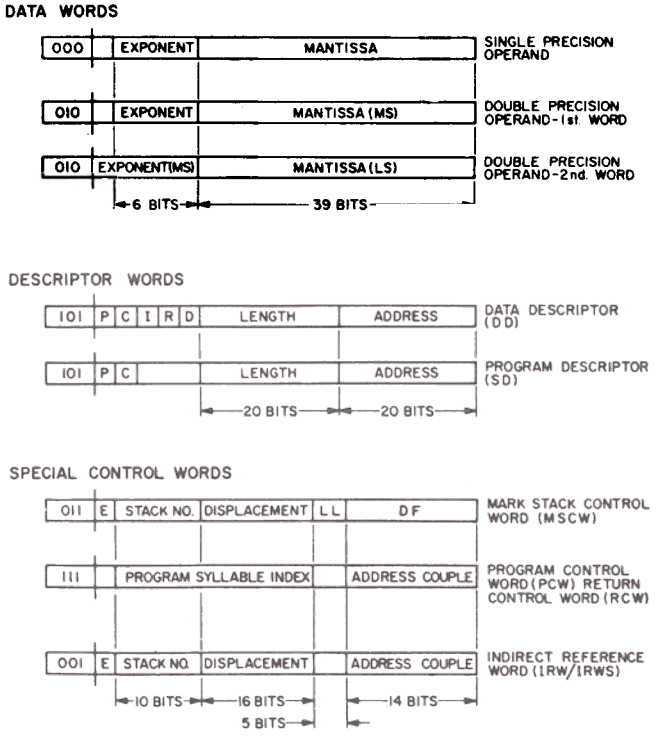

Figure 1 – B6500/B7500 word formats

The stack

The stack consists of an area of memory assigned to a job. This stack area serves to provide storage for basic program and data references associated with the job. In addition, it provides a facility for the temporary storage of data and job history. When the job is activated, four high speed registers (A, X, B and Y) are linked to the job's stack area (Fig. 2). This linkage is established by the stack pointer register (S), which contains the memory address of the last word placed in the stack memory area. The four top-of-stack registers (A, X, B and Y) function to extend the job's stack into a quick access environment for data manipulation.

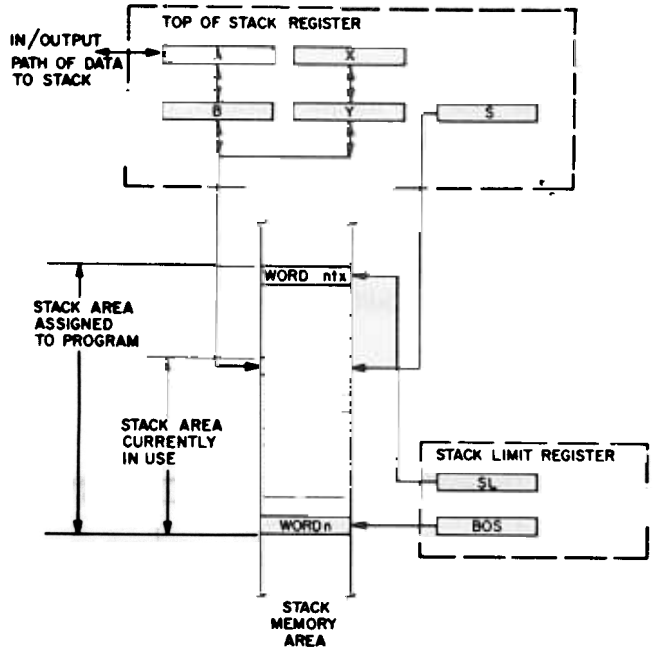

Figure 2—Top of stack and stack bounds registers

Data are brought into the stack through the top-of-stack registers. The stack's operating characteristic is such that the last operand placed into the stack is the first to be extracted. The top-of-stack registers become saturated after having been filled with two operands. Loading a third operand into the top-of-stack registers causes an operand to be pushed from the top-of-stack registers into the stack memory area. The stack pointer register (S) is incremented by one as each additional word is placed into the stack memory area; and is, of course, decremented by one as a word is withdrawn from the stack memory area and placed in the top-of-stack registers. As a result, the S register continually points to the last word placed into the job's stack memory area.

A job's stack memory area is bound, for memory protection, by two registers, the Base of Stack (BOS) register, and the Stack Limit (SL) register. The contents of the BOS register defines the base of the stack area, and the SL register defines the upper limit of the stack area. The job is interrupted if the S register is set to the value contained in either SL or BOS.

The contents of the top-of-stack registers are maintained automatically by the processor hardware in accordance with the environmental demands of the current operator syllable. If the current operator syllable demands that data be brought into the stack, then the top-of-stack registers are adjusted to accommodate the incoming data, and the surplus contents of the top-of-stack registers, if any, are pushed into the job's stack memory area. Words are brought out of the job's stack memory area and pushed into the top-of-stack register for operator syllables which require the presence of data in the top-of-stack registers, but do not explicitly move data into the stack.

Top-of-stack registers operate in an operand oriented fashion as opposed to being word oriented. Calling a double precision operand into the top-of-stack registers implies the loading of two memory words into the top-of-stack registers. The first word is always loaded into the A register where its tag bits are checked. If the word has a double precision tag, a second word is loaded into X. The A and X registers are then concatenated to form a double precision operand image. The B and Y registers concatenate when a double precision operand is moved to the B register. The double precision operand splits back to single words as it is pushed from the B and Y registers into the stack memory area. The reverse process is repeated when the double precision operand is eventually popped up from the stack memory area back into the top-of-stack registers.

Data addressing

Three mechanisms exist within the B6500/B7500 Processor for addressing data or program code: (1) Data Descriptor (DD)/Segment Descriptor (SD), (2) Indirect Reference Word (IRW), and (3) Stuffed Indirect Reference Word (IRWS). The Data Descriptor (DD) and Segment Descriptor (SD) are B5500 carryovers and provide the basic mechanism for addressing data or program segments which are located outside of the job's stack area. The basic addressing component of the descriptor is an absolute machine address. The Indirect Reference Word (IRW) and the Stuffed Indirect Reference Word (IRWS) are B6500/B7500 mechanisms for addressing data located within the job's stack memory area. The addressing component of both the IRW and IRWS is a relative address. The IRW is used to address within the im-

mediate environment of the job's stack, and addresses relative to a display register (described later in Non-local Addressing). The IRWS is used to address beyond the immediate environment of the current procedure, and addresses relative to the base of the job's stack. Addressing across stacks is accomplished with an IRWS.

The descriptor

In general, the descriptor functions to describe and locate data or program code associated with a given job. The Data Descriptor (DD) is used to fetch data to the stack or store data from the stack into an array which resides outside the job's stack area. The format of Data and Segment Descriptors are illustrated in Fig. 1. The ADDRESS field of both descriptors is 20 bits in length and contains the absolute address of an array in either main system memory or in the back-up disk store. The Presence bit (P) indicates whether the referenced data are present in main system memory or in the back-up disk store, and is set equal to ONE when the referenced data are present in main system memory.

A Presence Bit Interrupt is incurred when the job makes reference to data via a descriptor which has a P bit equal to ZERO. The Presence Bit Interrupt stimulates the operating system (called the Master Control Program, or MCP) to move the data from disk to main memory. The data location on disk is contained in the ADDRESS field of the DD when the P bit is equal to ZERO. After transferring the data array into the main memory, the operating system (MCP) marks the descriptor present by setting the P bit equal to ONE, and places the current memory address into the ADDRESS field of the descriptor. The interrupted job is then reactivated.

A Data Descriptor may describe either an entire array of data words, or a particular element within an array of data words. If the descriptor describes an entire array, the Indexed bit (I-bit) in the descriptor is ZERO, indicating that the descriptor has not yet been indexed. The LENGTH field of the descriptor defines the length of the data array.

A particular element of an array may be described by indexing an array descriptor. Memory protection is insured during indexing operations by performing a comparison between the LENGTH field of the descriptor and the index being applied to it. An Invalid Index Interrupt is incurred if the index value exceeds the length of the memory area defined by the descriptor.

If the value being used to index the descriptor is valid, the LENGTH field of the descriptor is replaced by the index value. At this time the I-bit in the descriptor is set to ONE to indicate that indexing has taken place. The ADDRESS and LENGTH fields are added together to generate an absolute machine address whenever a present, indexed Data Descriptor is used to fetch or store data.

The Double Precision bit (D) is used to identify the referenced data as being either single or double precision and, as a result, is also associated with the indexing operation. The D bit being equal to ONE signifies double precision and implies that the index value be multiplied by two before indexing.

The Read-Only bit (R) specifies that the memory area described by the Data Descriptor is a read-only area. An interrupt is incurred upon referencing an area through a descriptor with the intention to write if the R bit is equal to ONE.

The Copy bit (C) identifies a descriptor as being a copy of a master descriptor and is related to the present bit action. The intent of the copy action is to keep multiple copies of an absent descriptor linked back to one master descriptor. Copy action is incurred when a job attempts to pass by name an absent Data Descriptor. When this occurs, the hardware manufactures a copy of the master descriptor, forces the C bit equal to ONE and inserts into the ADDRESS field the address of the master descriptor. Thus, multiple copies of absent descriptors are all linked back to the master descriptor.

Non-local addressing

The most important single aspect of the B6500/B7500 stack is its facility for storing the dynamic history of a program under execution. Two lists of program information are saved in the B6500/B7500 stack, the stack history list and the addressing environment list. The stack history list is dynamic in nature, varying as the job is driven through different program paths with changing sets of data. Both lists are generated and maintained by the B6500/B7500 hardware system.

The stack history list is formed from a list of Mark Stack Control Words (MSCW) which are linked together by their DF fields (Fig. 3). A MSCW is inserted into the stack as a procedure is entered, and is extracted as that procedure is exited. Therefore, the stack history list grows and contracts in accordance with the procedural depth of the program. Mark Stack Control Words serve to identify the portion of the stack related to each procedure. When the procedure is entered, its parameters and local variables are entered in the stack following the MSCW. When executing the procedure, its parameters and local variables are referenced by addressing relative to the location of the related MSCW.

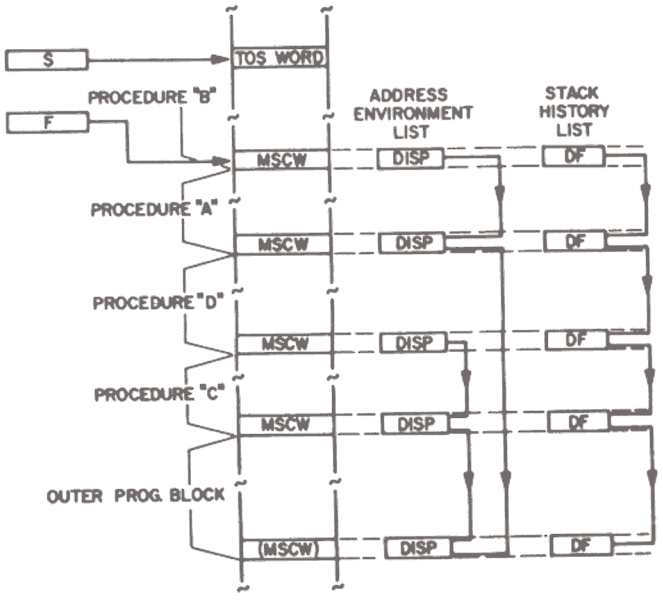

Figure 3—Stack history and addressing environment list

Each MSCW is linked back to the prior MSCW through the contents of its DF field to identify the point in the stack where the prior procedure began. When a procedure is exited, its related portion of the stack is discarded. This action is achieved by setting the stack pointer register (S) to point to the memory cell preceding the most recent MSCW (Fig. 4). This top-most MSCW, pointed to by another register (F), is in effect deleted from the stack history list by causing F to point back at the prior MSCW, thereby placing it at the head of the stack history list.

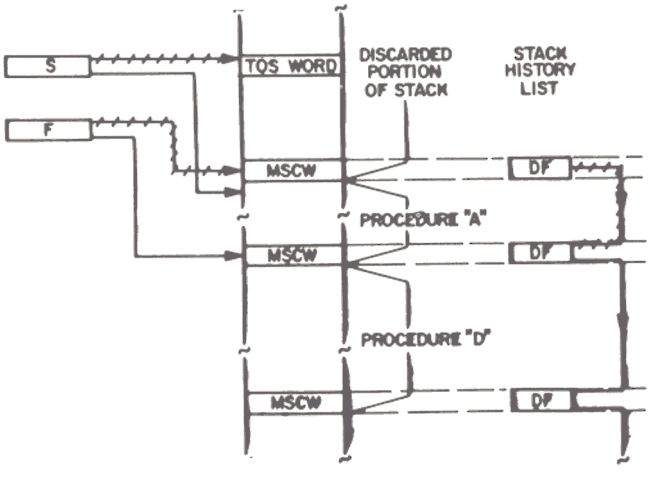

Figure 4—Stack cut-back operation on procedure exit

This concept is implemented in the Burroughs' B5500 system, and it provides a convenient means to handle subroutine entry and exit. But this mechanism alone also gives rise to one of the most serious limitations of the ALGOL implementation on the B5500. In the B5500 stack, local variables are addressed relative to the first Mark Stack Control Word (which corresponds to the outer-most block), or relative to the most recent Mark Stack Control Word (which corresponds to the current procedure). All intervening Mark Stack Control Words, however, are invisible to the current procedure. This means that the variables declared global to the current procedure, but local to some other procedure, cannot be addressed at all! This inability to reference variables declared non-local to the current procedure but local to some other procedure is termed the non-local addressing problem.

The manner in which these variables are addressed in the B6500/B7500 stack can best be understood by analyzing the structure of an ALGOL program. The addressing environment of an ALGOL procedure is established when the program is structured by the programmer, and is referred to as the lexicographical ordering of the procedural blocks (Fig. 6A). At compile time, this lexicographical ordering is used to form address couples. An address couple consists of two items: 1) the lexicographical level (ℓ) of the variable and 2) an index value (δ) used to locate the specific variable within a given lexicographical level. The lexicographical ordering of the program remains static as the program is executed, thereby allowing variables to be referenced via address couples as the program is executed.

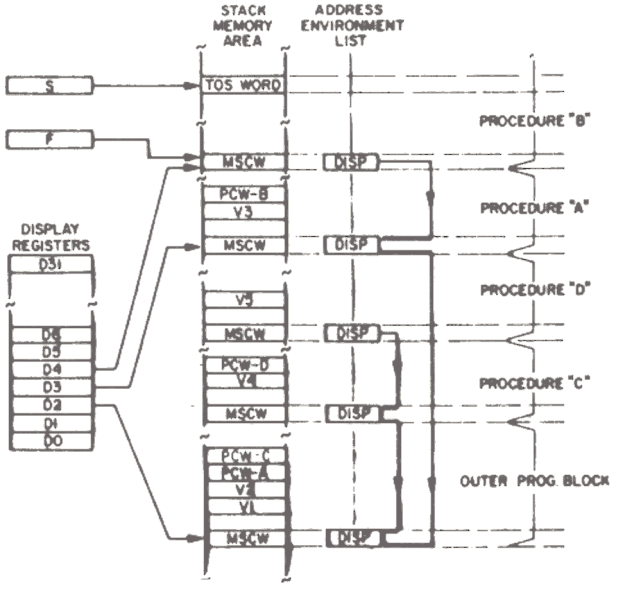

Figure 5—Display registers indicating current addressing environment

```
BEGIN                                          Lexicographical Level "2"
   REAL V1;                                    ll=2, δ=2
   REAL V2;                                    ll=2, δ=3
   PROCEDURE A;                                ll=2, δ=4
      BEGIN                                    Lexicographical Level "3"
         REAL V3;                              ll=3, δ=2
         PROCEDURE B;                          ll=3, δ=3
            BEGIN                              Lexicographical Level "4"
               V3←3;
               V1←V3;
            END;
         B;
      END;
   PROCEDURE C;                                ll=2, δ=5
      BEGIN                                    Lexicographical Level "3"
         REAL V4;                              ll=3, δ=2
         PROCEDURE D;                          ll=3, δ=3
            BEGIN                              Lexicographical Level "4"
               REAL V5;                        ll=4, δ=2
               V4←4;
               V5←5;
               A;
               V2←V4;
            END;
         D;
      END;
   C;
END;
```

Figure 6a—ALGOL program with lexicographical structure indicated

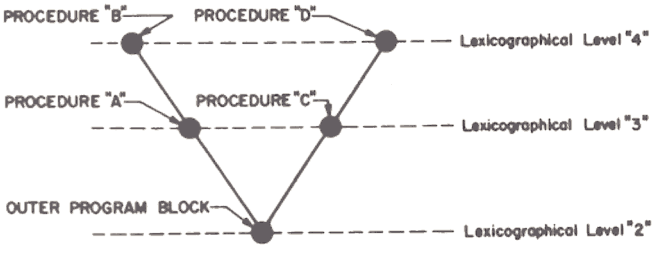

Figure 6b—Addressing environment tree of ALGOL program in Figure 6a

The B6500/B7500 contains a network of Display Registers (D0 through D31) which are caused to point at the appropriate MSCW (Fig. 5). The local variables of all procedures global to the current procedure are addressed in the B6500/B7500 relative to the Display Registers.

The address couple is converted into an absolute memory address when the variable is referenced. The lexicographical level portion of the address couple functions to select the Display Register which contains an absolute memory address pointing at the MSCW related to the procedural block (environment) where the referenced variable is located. The index value of the address couple is then added to the contents of the Display Register to generate an absolute memory address to locate the variable.

It should be recognized that the address couples assigned to the variables in a program are not unique. This is true because of the ALGOL scope of definition rules, which imply that two variables may have identical address couples only if there is no procedure within which both of the variables can be addressed. So this addressing scheme works because, whereas two variables may have the same address couples, there is never any doubt as to which variable is being referenced within any particular procedure.

What this does imply, however, is that there is a unique place (a MSCW) to which each Display Register must point during the execution of any particular procedure, and that the settings of the Display Registers might have to be changed, upon procedure entry or exit, to point to the correct MSCW. This list of MSCWs to which the Display Registers must point is called the addressing environment of the procedure.

The addressing environment of the program is maintained by the hardware. It is formed by linking the MSCW's together in accordance with the lexicographical structure of the program. This linkage information is contained with the Stack Number (Stack No.) and Displacement (DISP) fields of the MSCW, and is inserted into the MSCW whenever a procedure is entered. The contents of the DISP field indicate the environment in which the entered procedure was declared. Thus the addressing environment list is formed by linking each procedure entry Mark Stack Control Word back to the MSCW appearing immediately below the declaration for that procedure. This forms a tree structured list which indicates the legitimate addressing environment of each procedure under dynamic conditions (Figs. 5 and 6B). This list is searched by the hardware to update the Display Registers' contents whenever a procedure entry or exit occurs.

The entry and exit mechanism of the Processor hardware automatically maintains both stack lists to reflect the current status of the program. Therefore, the system is able to respond to, and return from, interrupts conveniently. Interrupt response is handled as a procedure entry. Upon recognition of an interrupt condition, the hardware causes the stack to be marked, inserts into the stack an indirect reference word (address couple) pointing to the interrupt handling procedure, inserts a literal constant to identify the interrupt condition, and then causes an entry into the operating system interrupt-handling procedure. The Display Registers will track with the entry into the interrupt-handling procedure to make all legitimate

variables visible. Also upon return, the Display Registers track back to the environment of the former procedure, making all of its variables visible again.

Multiple stacks and re-entrant code

The B6500/B7500 stack mechanism provides a facility to handle several active stacks. These stacks are organized into a single tree structure. The trunk of this tree structure is a stack which contains certain operating system global variables, and contains all of the Segment Descriptors describing the various procedures within the operating system.

Let us make a distinction between a program, which is a set of executable instructions, and a job, which is single execution of a program for a particular set of data. As the operating system is requested to run a job, a level-1 branch of the basic stack is created. This level-1 branch is a stack which contains only the Segment Descriptors describing the executable code for the named program. Emerging from this level-1 branch is a level-2 branch, a stack to contain the variables and data for this job. Thus, starting from the job's stack and tracing downward through the tree-structure, one would find first the stack containing the variables and data for the job (at level 2), the program code to be executed (at level 1), and finally the operating system's stack at the trunk (level 0).

A subsequent request to run another execution of an already-running program would require that only a level-2 branch be established. This level-2 stack branch would sprout from the level-1 stack that describes the already running program. Thus two jobs which are different executions of the same program will have a common node, at level 1, which describes the executable code. It is in this way that program code, which is not modifiable, is re-entrant and shared. It comes about simply from the proper tree-structured organization of the various stacks within the machine. Thus all programs within the system are re-entrant, including all user programs as well as the compilers and the operating system itself.

The B6500/B7500 stack mechanism also provides the facility for a single job to split itself into two independent jobs. It is anticipated that the most common use of this facility will occur when there is a point in a job where two relatively large independent processes must be performed. This kind of splitting could be used to make full use of a multiprocessor configuration, or simply to reduce elapsed time by multiprogramming the independent processes.

This kind of program splitting becomes almost literally "reproduction by budding" in the B6500/B7500 system. A split of this type is handled by establisting a new limb of the tree structured stack, with the two independent jobs sharing that part of the stack which was created before the budding was requested. The process is recursively defined, and can happen repeatedly at any level. An implementation restriction limits the total number of separate stacks to 1024.

This tree-structured organization for handling multiple stacks is referred to as the Saguaro Stack System.

Linkage of stack branches is achieved through a single array of data descriptors, the stack vector array (Fig. 7). A data descriptor is entered into the array for every stack branch as it is set up by the operating system. This data descriptor, the stack descriptor, serves to describe the length of the memory area assigned to a stack branch, and its location in either main memory or on disk.

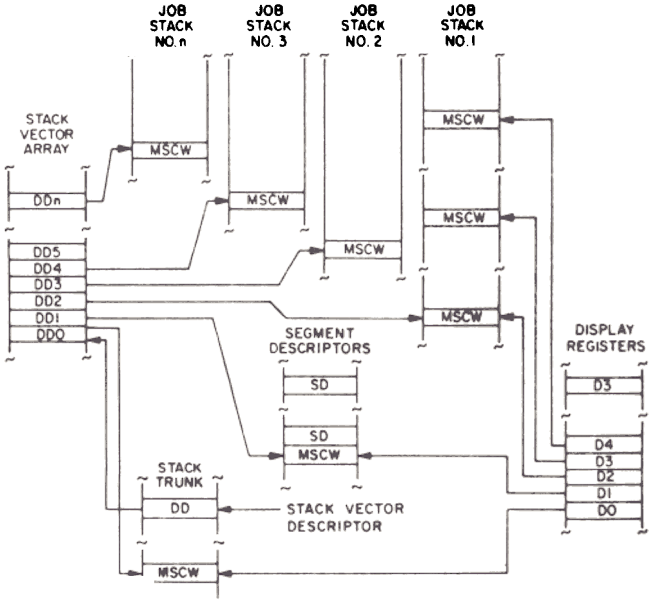

Figure 7 – Multiple linked stacks

A stack number is assigned to each stack branch to indicate the position of its stack descriptor within the stack vector array. The stack number is used as an index value to locate the related stack descriptor from the stack vector array for subsequent reference.

The stack vector array's size and location in memory is described by the stack vector descriptor. This descriptor is located in a reserved position of the stack's trunk (Fig. 7). All references to stack branches are made through the stack vector descriptor which is indexed by the value of the stack number

to select the stack descriptor for the referenced stack.

A Presence Bit Interrupt is incurred upon making reference to a stack which is not present in memory. This Presence Bit Interrupt facility provides the means to permit stack overlays and recalls under dynamic conditions. Idle or inactive stacks may be moved from main memory to disk as the need arises, and when subsequently referenced will cause a Presence Bit Interrupt which triggers the operating system to recall the non-present stack from disk.

Referencing a variable within the current addressing environment of an active procedure is accomplished through the use of the address couples contained in the IRW and the address couple field of the Program Control Word (PCW) as shown in Fig. 1. Both references are made relative to the Display Registers specified by the address couple. The address couple and Display Registers are usable only for addressing variables within the scope of the current addressing environment. Reference to variables beyond the scope of the current environment is accomplished by a stuffed IRWS. This causes the addressing to be accomplished by addressing relative to the base of the stack (BOS) in which the variable is located.

The IRWS contains information specifying the stack number (Stack No.), the location (DISP) of the related MSCW, and the displacement (δ) of the parameter relative to the MSCW. The absolute memory location of the sought parameter is formed by adding the contents of DISP and δ to the base address of the referenced stack. The base address of the stack is determined by accessing the stack descriptor as described previously. The information contents of the stuffed IRWS with the exception of δ, is dynamic in nature and must therefore be accumulated as the program is executed. The contents of the stack number (Stack No.) and DISP fields are entered into the IRWS by a special hardware operator which is invoked by the software whenever the program attempts to pass a parameter by name.

ACKNOWLEDGMENTS

Recognition for the stack concepts and operating philosophy of Burroughs' B6500/B7500 system must be extended to many system designers engaged in both the B5500 and B6500/B7500 programs. Among the contributors, special mention should be made for B. A. Creech, Burroughs Corporation, and R. S. Barton, W. M. McKeeman, consultants.

REFERENCES

1 *Burroughs' B5500 information processing system reference manual*
Burroughs Corporation 1964

2 *A narrative description of the Burroughs' B5500 disk file master control program*
Burroughs Corporation 1965

3 B RANDALL L J RUSSELL
ALGOL 60 implementation
Academic Press 111 5th Avenue New York 1964