THE MECHANIZATION OF A PUSH-DOWN STACK

C. B. Carlson
Burroughs Corporation
ElectroData Division
Pasadena, California

INTRODUCTION

The present trend in programming General Purpose Computers is toward the use of Problem Oriented Languages such as ALGOL and COBOL. These languages provide convenient coupling between Man and Machine but impose additional requirements such as Automatic Compilers.

In the past, application of Problem Oriented Languages has been hampered because the Compiler has been obliged to work in machines that were designed for less sophisticated languages. The efficiency of both the Compiler and the Object Program can be greatly increased if the hardware provides nearly a "one for one" correspondence between the Source Language Operators and the Machine Instructions.

Early in the design of the Burroughs B5000 Processor, the decision was made to create an Operator Set and Logical Functions that provide, as nearly as possible, this "one for one" relationship.

The resulting machine has some interesting features and among them is an Automatic Push-Down Stack. This Stack has been built into the hardware and is now in operation.

It is the purpose of this paper to describe the organization, automatic control and application of this Stack.

GENERAL SYSTEM ORGANIZATION

The B5000 is a medium size general purpose data processing system designed for both scientific and commercial data processing. It is modular in design, permitting changes in the system configuration to fit the requirements of the user. Systems are made up from combinations of the following modules:

1 Central Control Unit
1 or 2 Processors
Up to 8 Memory Units or 4096 words each
1 or 2 Magnetic Drums of 32786 word capacity
1 to 4 Input/Output Control Units
1 to 16 Magnetic Tape Transports
1 or 2 High Speed Printers
1 or 2 Card Reader Units, either 200 or 800 CPM
1 Card Punch Units, either 100 or 300 CPM
1 or 2 Paper Tape Readers { A total of 3
1 or 2 Paper Tape Punches } units
1 Control Console

The system operates at a one megacycle clock frequency, performing a 39-bit addition in one clock pulse. The processing speed is so much greater than the speed of the peripheral machines that full utilization of the processor becomes a problem. Much of the imbalance in speed between the processor and the I/O operations is absorbed by parallel operation of the Processor and I/O Units.

Parallel operation is facilitated by a Memory Exchange switching interlock whereby memory is time shared by, and accessible to, both the Processor and the I/O Control Units. There is no direct transfer of data between these units but data flow is effected by means of individual
memory read or write operations originating with the Processor or the I/O Control Units. Thus a Processor may initiate an I/O operation that will be carried to completion by the I/O Control Unit. Once the I/O operation is in progress the Processor is free to execute another program. This ability to transfer the Processor from one program to another is called multiprocessing and will be discussed in further detail.

An executive routine called the Master Control Program (MCP) is required to maintain efficient control of these operations. The MCP schedules all jobs assigned to the system and controls such operating functions as assigning memory space, compiling object programs and directly I/O operations. An extensive automatic interrupt system is built into the hardware to provide the necessary controls for the MCP.

THE HARDWARE STACK

Into this system organization is incorporated a push-down stack that provides an automatic temporary store for parameters and control information relating to the program currently being executed. We will now consider the hardware that constitutes this stack.

Actually the amount of this hardware is small.

1. An assigned memory area of not more than 1024 words.
2. A 15-bit address register s.*
3. An identical register f.
4. Two 48-bit registers a and b in the Processor and their satellite flip-flops afof and brof.
5. The associated transfer paths consisting of cables and logical gates.

Memory Stack

The memory space assigned to the stack is specified by the memory allocation routine of the MCP. The size and location of this space may be changed from time to time by the MCP. However, to an object program, the top of the stack is specified as the absolute address contained in the register s. As words are stored in the stack, the s register is incremented and as words are read from the stack the s register is decremented such that s is always pointing to the top word in the memory stack.

Stack underflow, i.e., the possibility of the s register counting down below the assigned Stack area, is checked programmatically. Essentially, the object program never attempts to access data that was never placed in the Stack.

Detection of Stack overflow is checked by the hardware. When an object program is brought into the core memory, the Master Control Program assigns contiguous memory areas to the Stack and the Program Reference Table (PRT). The PRT is positioned above the Stack such that the base of the PRT is the upper limit of the Stack.† When a Processor is executing an object program the r register is set to the address of the base of the PRT and it remains set to this value so long as the Stack is active.

* Figure 6 lists the registers and symbols referred to in this paper.

† See Figure 1 for a diagram of these memory areas.
The test for overflow is simply a test for \( s = r \) and is performed each time \( s \) is counted plus one. When \( s = r \), an interrupt condition exists which requires that the state of the program be stored in the Stack and that control be turned over to the MCP. To provide a buffer area in which to store this overflow information the \( s \) register is allowed to count several locations into the PRT. Since the assignment of space in the PRT is controlled by program it is a simple matter to reserve this space.

**The Active Top of the Stack**

When a Processor is executing an object program, any word that is placed in the \( a \) register or in the \( b \) register is considered to be in the Stack. The \( a \) register is the top position of the Stack and the \( b \) register is the second position. Either, or both of these positions may be full or empty, thus if the \( a \) register is full and the \( b \) register empty there is a void in the Stack. Full control of this condition is provided by the two satellite flip-flops \( a_{\text{rof}} \) and \( b_{\text{rof}} \) which indicate the occupancy of the \( a \) register and the \( b \) register. When \( a_{\text{rof}} \) is true the word in \( a \) is a valid member of the Stack. If \( a_{\text{rof}} \) is false the \( a \) register is empty. Likewise, \( b_{\text{rof}} \) indicates the status of the \( b \) register.

**CONTROL OF THE STACK**

The hardware just described provides the mechanism to store data into and extract data from the Stack. It also provides automatic control of addressing at the micro level.

A higher level of mechanization exists in the logic for each operator syllable. Some of this logic acts directly to provide automatic functions such as stack adjustment, the generation and use of Control Words\(^1\) and subroutine linkage control. However, the full effect of the Stack under control of the operators is not apparent until the operators are used in program sequence. The balance of this paper is devoted to a description of some features of the machine language operators and the application of the Stack to the execution of a simple ALGOL statement.

**Automatic Stack Adjustment**

Essentially all the operator syllables or instructions, that comprise the B5000 machine language require that operations be performed in the \( a \) or \( b \) registers. In order to accomplish this it is first necessary to see that space is available in these registers or that the requisite data is pushed up from the Stack. This operation is called stack adjustment and automatic control of this function is mechanized as follows.

There are four possible states that the two flip-flops \( a_{\text{rof}} \) and \( b_{\text{rof}} \) can assume, at least one of which reflects the initial data requirements of any operator syllable. Some operators are binary and require two operands. Others are unary and require only one word in the \( a \) or the \( b \) register before they can perform their function. An automatic stack adjustment operation which loads \( a \) and/or \( b \) as needed, is an integral part of each operator and logical gates are built into the hardware which activate the necessary push-up and push-down operations.

The function of these gates can be described as a series of tests and actions. Figure 2 is a set of four Microprograms which define the Stack adjustment operation required to establish each of the four states for \( a_{\text{rof}} \) and \( b_{\text{rof}} \).

In the first Microprogram it is required that any valid words remaining in the \( a \) or the \( b \) registers be placed in the Memory Stack (pushed down). This is the state where \( a_{\text{rof}} = b_{\text{rof}} = 0 \). Actually the logical steps defined by the Microprogram are quite simple. First the vector \( y \) is tested for the value zero. If this is true then the operation is already complete, otherwise the next test is made—(Is the second term in \( y \) (\( b_{\text{rof}} \) equal to 1)?). If true we count the \( s \) register +1 to address an empty cell in the Stack. Having counted \( s + 1 \), a test is made for stack overflow, \( (s : r) \). If there is no stack overflow \( (s \neq r) \) then the content of the \( b \) register is stored in the memory location addressed by the content of \( s \). The \( b \) register is now empty and \( b_{\text{rof}} \) is set to zero to indicate this fact.

---

\(^1\) Control Words are 48-bit words which contain register settings, flip-flop states, address fields, etc. The format of some control words and data words are shown in Figure 5.
The state of arof is next tested—(Is the first term in y (aroF) equal to 1?). If it is not one, then both a and b registers are empty and the operation is finished. However, if arof is found equal to one, the contents of the a register are stored in memory in the same manner that b was stored and the operation is complete.

A study of these Microprograms makes it apparent that the hardware is automatically performing functions that would otherwise be done by program. The fact that these functions are common to almost all machine language operators justifies their hardware implementation.

**Multiprocessing**

During the execution of a program it is frequently necessary to stop the processing operations while a data area is filled or a new segment of program is brought into Core Memory. To allow the processor to stand idle during this I/O operation is wasteful and cannot be tolerated.

The programming concept of the B5000 provides that several programs may be in core at any time and that the Processor may switch from one to another under control of the MCP. Mechanization of this operation is accomplished in the following manner.

All the control information for a program is placed in three areas as it is brought into Core Memory. These areas are the Program Reference Table, the Stack and the Program Area (PA).

Figure 1 is a block diagram showing these areas. If the Processor is executing Program A and, before it has completed it, it is called upon to process Program B, the Processor will first store all the information it contains relative to Program A in the Stack. This includes the push-down of any data in the a register or the b register and the storage of control words that contain the essential state of the Processor. In this manner the integrity of Program A is preserved and the Processor is free to set its registers r, s and c to the addresses belonging to Program B. Subsequently, this Processor (or the second Processor in a two Processor System) may return to Program A, reset its r, s and c registers, restore itself from the information stored in the Stack and continue with the execution of Program A.

This complete independence and integrity of each program is the basis for efficient Multiprocessing. Programs can be written without regard for other programs that may be in the system at the same time.

**Application of the Stack to an Object Program**

The use of the Stack in conjunction with the PRT and the PA may be illustrated with a simple ALGOL Program:

```
BEGIN
REAL A,B,C,D;
A ← B + C × D;
END.
```

During compilation the Declaration "REAL" will reserve a location in the PRT for each of the four variables A, B, C, and D. Compilation also generates the following program string and stores it in the PA:

```
"OPDC B, OPDC c, OPDC J, MUL, ADD, LITC 0025, STD"
```

1 These are machine operator codes. More detail on their function may be found in 5.
Processing this object program proceeds as follows:

1. The first syllable OPDC_B (an operand call on the word B) brings the Operand Word B into the Stack.
2. The second syllable places the Operand Word C in the Stack, positioned above B.
3. In like manner, D is placed in the Stack.
4. The next syllable encountered by the Processor is MUL (multiply) which operates on the top two words in the Stack, destroying C and D and leaving their product at the top of the Stack.
5. The fifth syllable, ADD (add), is also binary and at this point, a push-up is automatically initiated to obtain B. The addition is then performed in the a and b registers leaving at the top of the Stack the computed value for A.
6. The next operator is a Literal Call Syllable which brings the r relative address which is reserved for A, into the top of the Stack.
7. The final operator stores the value of A in the PRT location reserved for A, (r + 25).

This program is comprised of Word Mode Operators, as are practically all operators that manipulate the Stack. In Character Mode the Stack remains intact and is used to pass parameters from and to Word Mode. However, operation of the Stack is basically a Word Mode Function.

In passing, it is interesting to note that there are seven terms in this original ALGOL statement and there are seven Program syllables in the compiled program. Although the correspondence is not always 100% as in this case, this illustrates the “one for one” relationship of operands, literals and operators that contributes to program and compiler efficiency.

---

**Application of the Stack to Subroutine Operation**

Entering and leaving Procedures or Subroutines is similar to initiating and interrupting programs during Multiprocessing to the extent that both require the Processor to staticize one program while it addresses itself to another. In the case of Multiprocessing, this transfer of control is effected via the MCP whereas subroutine entry is performed by the object program. To provide the compiler and object programs with means to enter subroutines using a minimum amount of coding, a large part of the address logic has been built into hardware. Several powerful operators are provided which automatically generate or use Control Words which are stored in the Stack. The control of the Stack during a subroutine entry and return can best be illustrated by another ALGOL Program.

Assume a program calls upon a variable “X,” as in the case shown in Figure 3. The object program will compile substantially as shown under “Main Program” and “Sub-}

---

**Figure 3. Call on Simple Procedure “X.”**
At the bottom of the figure are shown five stages in the life of the Stack during the processing of this operator string.

Executing the Main Program syllables from left to right, the Processor encounters a Mark Stack Operator. This operator first pushes down into Memory Stack, any valid words in the a or b registers. It then builds a MSCW (Mark Stack Control Word) containing the \( r \) and \( f \) registers and msff and salff, and this MSCW word is pushed down into Memory Stack. The address of this MSCW is temporarily placed in the \( f \) register. Stage 1 shows the Stack with \( b, a \) and the MSCW in the Memory Stack.

The Parameters \( Y \) and \( Z \) are next placed in the Stack by two Operand Call syllables. Stage 2 depicts the addition of these parameters.

The next operator encountered in the Main Program is an Operand Call on a Program Descriptor. The ensuing operation builds a Return Control Word (RCW) from the pertinent registers in the Processor, including the \( f \) register, and pushes it into the Stack. The contents of the \( f \) register are now replaced with the address of this RCW. This completes the linkage back to the Main Program level. This Operand Call also addresses the \( c \) and \( l \) registers to the subroutine and places the Processor in Sublevel. The Stack at stage 3 shows the addition of the RCW.

The Processor now encounters the syllables in the subroutine, the first three of which comprise a program to compute the value for \( X \). Stage 4 shows the use of additional area in the Stack for this computation.

The subroutine is terminated with a Return Operator that accesses the RCW and MSCW and readresses the Processor to the Main Program. Stage 5 shows the \( s \) register pointing to the original content of the top of the Stack. The result of the subroutine has been passed back to the Main Program and is now in the \( a \) register. The Processor registers (\( H, V, L, G, K, F, C, R \)) and flip-flops (msff, salff) have been restored from the content of the control words.

**Application of the Stack to Recursive Procedures**

The ability to enter a subroutine is recursive and it is essential that the control linkage from one level to the next be preserved.

The following example demonstrates how the stack is used to retain this linkage for subroutines of unlimited depth.

Consider an ALGOL Program that calls on a Procedure “FACT”:

\[
X \leftarrow \text{FACT}(3); \\
\text{PROCEDURE FACT:} \\
\text{REAL PROCEDURE FACT(N) ;} \\
\text{IF N = 1 THEN FACT \leftarrow 1} \\
\text{ELSE FACT \leftarrow N \times \text{FACT}(N - 1); }
\]

Figure 4 shows the Stack at the limit of recursion with the control words and parameters.

---

\# Sublevel and Program Level are two conditions under which the Processor operates. They are described in 5.
ters that it contains. The \( f \) register is set to the address of each control word as it is placed in the Stack and the sequential values are indicated with a superscripted \( F^x \). Stored in each control word is the content of the \( f \) register at the time the control word was built. This value of \( F^x \) is the address of the previous control word. Thus there is stored in the Stack a full linkage from the latest address in the \( f \) register back through the control words to the original program level.

Following each MSCW in this Stack, is a Formal Parameter that is to be passed to the subroutine. The Procedure requires this parameter twice during each recursion. First, it is required to derive the value \( N \), at which time it is accessed with an \( f \) relative address \((f - 1)\). The second time it is required to compute \((N - 1)\). By this time the \( f \) register has been set to a new value and access to \( N \) via the Control Word Linkage could become complex. This search can be avoided if the settings of \( F^2 \) (and later \( f^3 \)) are preserved in the PRT at \((r + 7)\). Operator logic is provided to automatically access the “old” value of \( F \) in \((r + 7)\) when these Global Parameters are required. This use of the content of \((r + 7)\) as a base for relative addressing is simply an extension of \( f \) relative addressing.

The return to the original program is performed sequentially with a return to the previous level, multiply \((N - 1) \times N\) and return to the next higher level, etc., until the original level is reached.

This elementary program illustrates how the Control Words, which are built and used by the hardware, automatically provide address links for any depth of subroutine. Also, the ability to address words \((f - )\) relative and \((f + )\) relative provides access to data that is below the current top of the Stack, allowing parameters to be passed forward from one level to the next.

**CONCLUSION**

It has not been possible to describe all the ramifications of the Stack and its use by the software. The applications given serve to illustrate how the mechanization of a push-down stack was organized in conjunction with the software design to automate many recurring

---

**Figure 5. Word Formats.**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>( w )</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>48</td>
<td>( A ) register Top position of Stack</td>
</tr>
<tr>
<td>B</td>
<td>48</td>
<td>( B ) register Second position of Stack</td>
</tr>
<tr>
<td>F</td>
<td>48</td>
<td>( F ) register Holds Program word</td>
</tr>
<tr>
<td>I</td>
<td>12</td>
<td>( I ) register Holds Program syllable</td>
</tr>
<tr>
<td>S</td>
<td>15</td>
<td>Program word address register</td>
</tr>
<tr>
<td>L</td>
<td>2</td>
<td>( L ) register Syllable pointer</td>
</tr>
<tr>
<td>J</td>
<td>4</td>
<td>( J ) register Sequence count</td>
</tr>
<tr>
<td>E</td>
<td>15</td>
<td>( E ) register Base of PRT</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>( F ) register Subroutine Stack pointer</td>
</tr>
<tr>
<td>OF</td>
<td>1</td>
<td>( OF ) register Occupancy flip-flop</td>
</tr>
<tr>
<td>ERF</td>
<td>1</td>
<td>( ERF ) register Occupancy flip-flop</td>
</tr>
<tr>
<td>STF</td>
<td>1</td>
<td>Mark Stack flip-flop</td>
</tr>
<tr>
<td>STF</td>
<td>1</td>
<td>Sub-level flip-flop</td>
</tr>
<tr>
<td>N</td>
<td>48</td>
<td>Core Memory Matrix (Memory Unit)</td>
</tr>
</tbody>
</table>

**Figure 6. Register and Symbols.**

NOTE: This list does not comprise all the registers in the Processor.
operations. The resulting powerful machine language operators have reduced the coding required to produce efficient compilers and object programs.

There is an opportunity for progress in the design of general purpose computers if consideration is given to the balance between the hardware logic and the logic performed by program. In the machine described, the automatic hardware stack has relieved the software of several arduous tasks and the cost in added components has been relatively small.

BIBLIOGRAPHY