Appendix O
Stacks

William Stallings

Supplement to
Computer Organization and Architecture, Ninth Edition
Prentice Hall 2012
ISBN: 013293633X
http://williamstallings.com/ComputerOrganization

This appendix provides an overview of the two most widely used protocol architectures.

L.1 STACKS

A *stack* is an ordered set of elements, only one of which can be accessed at a time. The point of access is called the *top* of the stack. The number of elements in the stack, or *length* of the stack, is variable. The last element in the stack is the *base* of the stack. Items may only be added to or deleted from the top of the stack. For this reason, a stack is also known as a *pushdown list*[1] or a *last-in-first-out (LIFO) list.*

Figure O.1 shows the basic stack operations. We begin at some point in time when the stack contains some number of elements. A PUSH operation appends one new item to the top of the stack. A POP operation removes the top item from the stack. In both cases, the top of the stack moves accordingly. Binary operators, which require two operands (e.g., multiply, divide, add, subtract), use the top two stack items as operands, pop both items, and push the result back onto the stack. Unary operations, which require only one operand (e.g., logical NOT), use the item on the top of the stack. All of these operations are summarized in Table O.1.

[1] A better term would be *place-on-top-of list* because the existing elements of the list are not moved in memory, but a new element is added at the next available memory address.

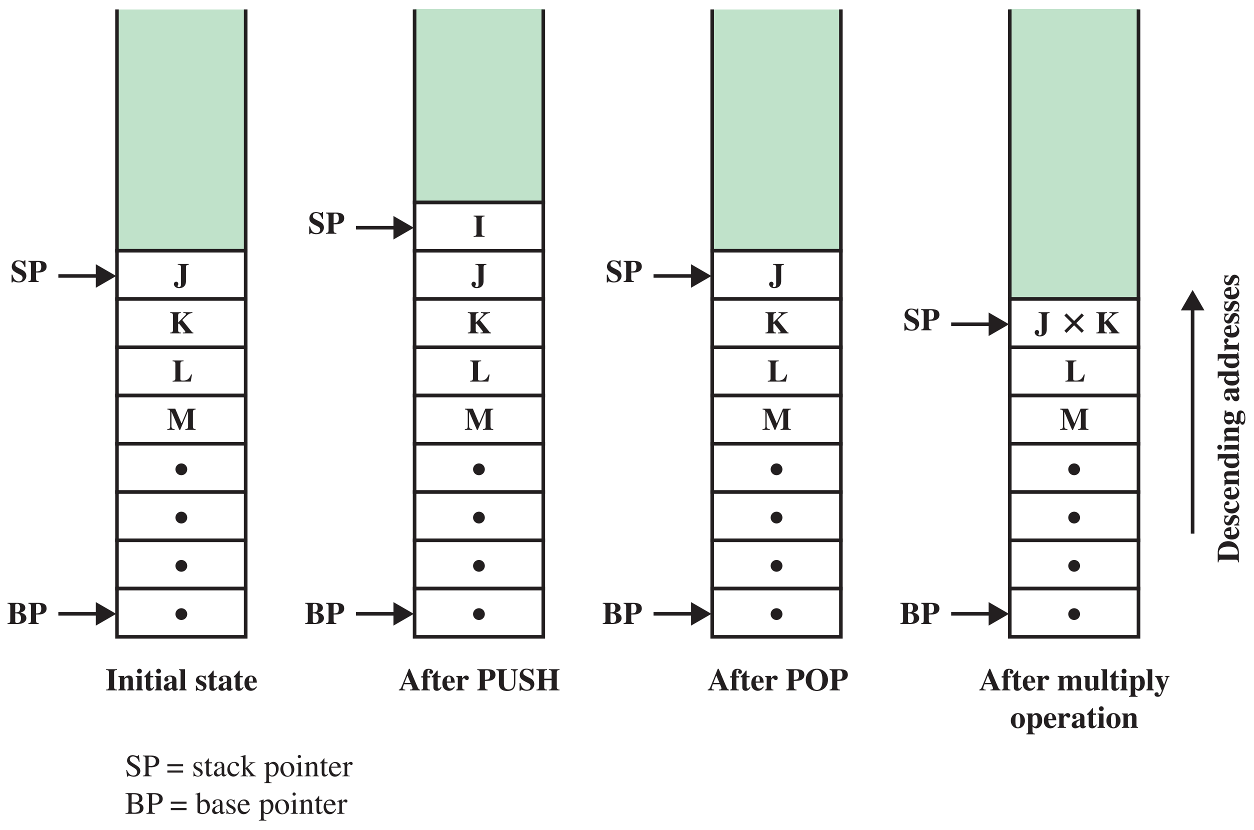

Figure O.1 Basic Stack Operation (full/descending)

Table O.1 Stack-Oriented Operations

PUSH	Append a new element on the top of the stack.
POP	Delete the top element of the stack.
Unary operation	Perform operation on top element of stack. Replace top element with result.
Binary operation	Perform operation on top two elements of stack. Delete top two elements of stack. Place result of operation on top of stack.

L.2 STACK IMPLEMENTATION

The stack is a useful structure to provide as part of a processor implementation. One use, discussed in Section 12.4, is to manage procedure calls and returns. Stacks may also be useful to the programmer. An example of this is expression evaluation, discussed later in this section.

The implementation of a stack depends in part on its potential uses. If it is desired to make stack operations available to the programmer, then the instruction set will include stack-oriented operations, including PUSH, POP, and operations that use the top one or two stack elements as operands. Because all of these operations refer to a unique location, namely the top of the stack, the address of the operand or operands is implicit and need not be included in the instruction. These are the zero-address instructions referred to in Section 12.1.

If the stack mechanism is to be used only by the processor, for such purposes as procedure handling, then there will not be explicit stack-oriented instructions in the instruction set. In either case, the implementation of a stack requires that there be some set of locations used to store the stack elements. A typical approach is illustrated in Figure O.2. A contiguous block of locations is reserved in main memory (or virtual memory) for the stack. Most of the time, the block is partially filled with stack elements and the remainder is available for stack growth.

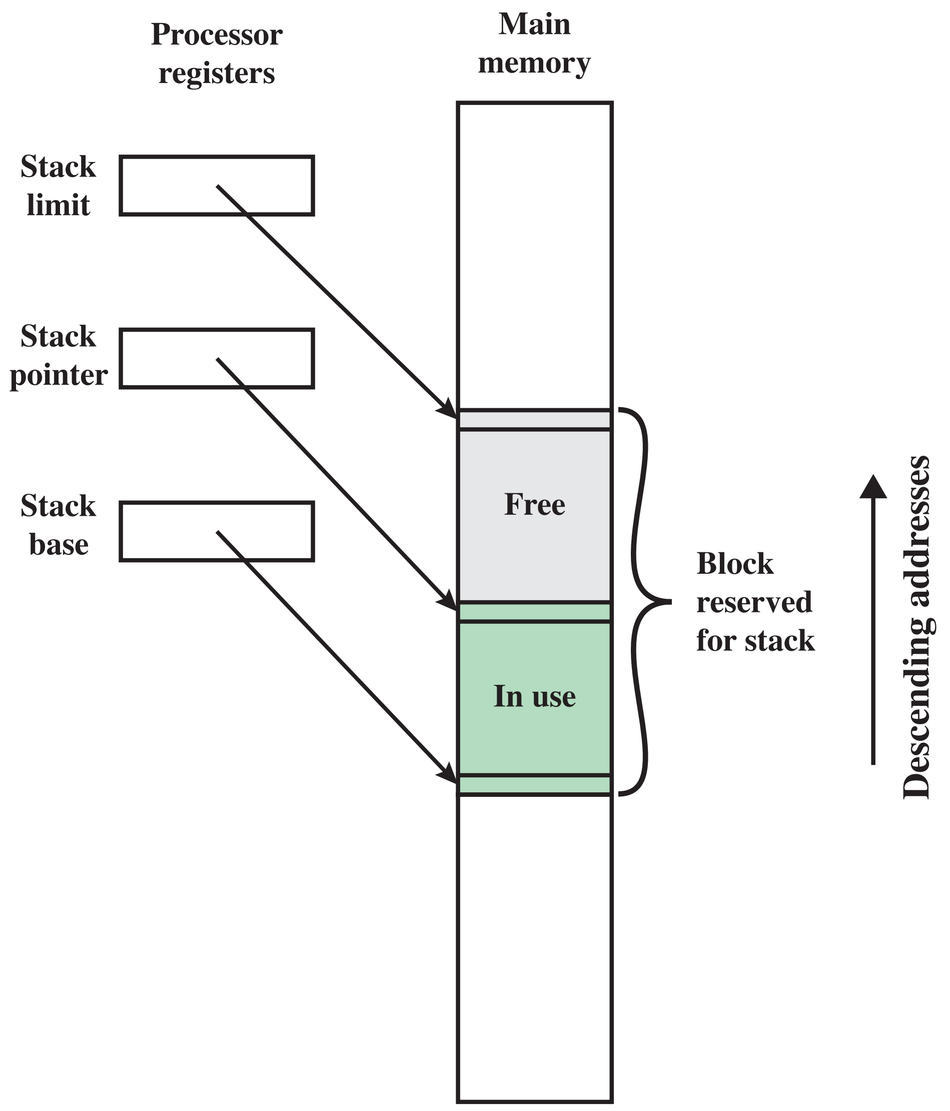

Figure O.2 Typical Stack Organization (full/descending)

Three addresses are needed for proper operation, and these are often stored in processor registers:

- **Stack pointer (SP):** Contains the address of the top of the stack. If an item is appended to or deleted from the stack, the pointer is incremented or decremented to contain the address of the new top of the stack.

- **Stack base:** Contains the address of the bottom location in the reserved block. If an attempt is made to POP when the stack is empty, an error is reported.
- **Stack limit:** Contains the address of the other end of the reserved block. If an attempt is made to PUSH when the block is fully utilized for the stack, an error is reported.

Stack implementations have two key attributes:

- **Ascending/descending**: An ascending stack grows in the direction of ascending addresses, starting from a low address and progressing to a higher address. That is, an ascending stack is one in which the SP is incremented when items are pushed and decremented when items are pulled. A descending stack grows in the direction of descending addresses, starting from a high address and progressing to a lower one. Most machines implement descending stacks as a default.
- **Full/empty**: This is a misleading terminology, because is does not refer to whether the stack is completely full or completely empty. Rather, the SP can either point to the top item in the stack (full method), or the next free space on the stack (an empty method). For the full method, when the stack is completely full, the SP points to the upper limit of the stack. For the empty method, when the stack is completely empty, the SP points to the base of the stack.

Figure O.1 is an example of a descending/full implementation (assuming that numerically lower addresses are depicted higher on the page). The ARM architecture allows the system programmer to specify the use of ascending or descending, empty or full stack operations. The x86 architecture uses a descending/empty convention.

L.3 EXPRESSION EVALUATION

Mathematical formulas are usually expressed in what is known as **infix** notation. In this form, a binary operator appears between the operands (e.g., a + b). For complex expressions, parentheses are used to determine the order of evaluation of expressions. For example, a + (b × c) will yield a different result than (a + b) × c. To minimize the use of parentheses, operations have an implied precedence. Generally, multiplication takes precedence over addition, so that a + b × c is equivalent to a + (b × c).

An alternative technique is known as **reverse Polish***,* or **postfix**, notation. In this notation, the operator follows its two operands. For example,

a + b	becomes a b +
a + (b × c)	becomes a b c × +
(a + b) × c	becomes a b + c ×

Note that, regardless of the complexity of an expression, no parentheses are required when using reverse Polish.

The advantage of postfix notation is that an expression in this form is easily evaluated using a stack. An expression in postfix notation is scanned from left to right. For each element of the expression, the following rules are applied:

1. If the element is a variable or constant, push it onto the stack.

2. If the element is an operator, pop the top two items of the stack, perform the operation, and push the result.

	Stack	General Registers	Single Register
	Push a Push b Subtract Push c Push d Push e Multiply Add Divide Pop f	Load R1, a Subtract R1, b Load R2, d Multiply R2, e Add R2, c Divide R1, R2 Store R1, f	Load d Multiply e Add c Store f Load a Subtract b Divide f Store f
Number of instructions	10	7	8
Memory access	10 op + 6 d	7 op + 6 d	8 op + 8 d

Figure O.3 Comparison of Three Programs to Calculate

After the entire expression has been scanned, the result is on the top of the stack.

The simplicity of this algorithm makes it a convenient one for evaluating expressions. Accordingly, many compilers will take an expression in a high-level language, convert it to postfix notation, and then generate the machine instructions from that notation. Figure O.3 shows the sequence of machine instructions for evaluating $f = (a - b)/(c + d \times e)$ using stack-oriented instructions. The figure also shows the use of one-address and two-address instructions. Note that, even though the stack-oriented rules were not used in the last two cases, the postfix notation served as a guide for generating the machine instructions. The sequence of events for the stack program is shown in Figure O.4.

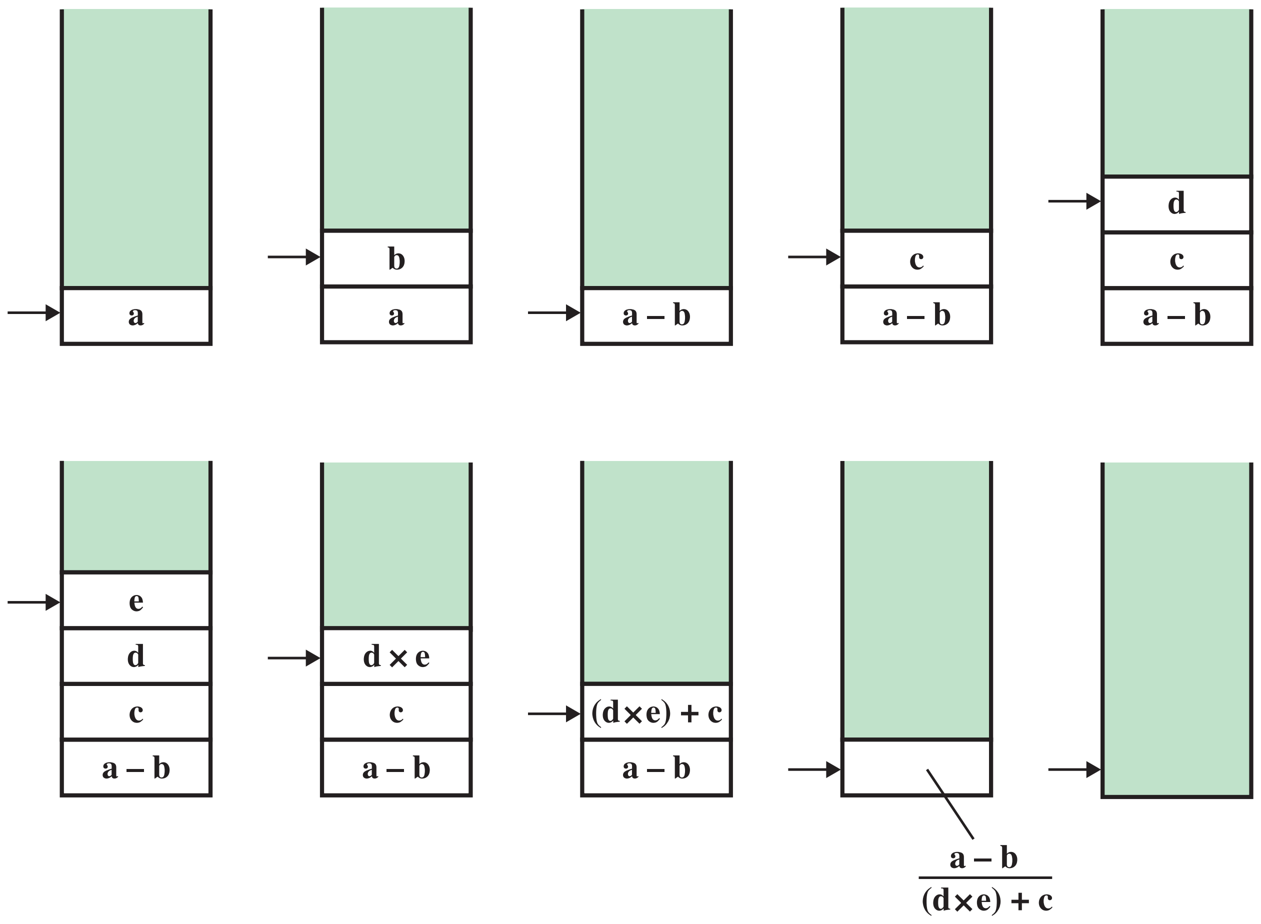

Figure O.4 Use of Stack to Compute f = (a – b)/[(d×e) + c]

The process of converting an infix expression to a postfix expression is itself most easily accomplished using a stack. The following algorithm is due to Dijkstra [DIJK63]. The infix expression is scanned from left to right, and the postfix expression is developed and output during the scan. The steps are as follows:

1. Examine the next element in the input.
2. If it is an operand, output it.
3. If it is an opening parenthesis, push it onto the stack.
4. If it is an operator, then

   - If the top of the stack is an opening parenthesis, then push the operator.
   - If it has higher priority than the top of the stack (multiply and divide have higher priority than add and subtract), then push the operator.
   - Else, pop operation from stack to output, and repeat step 4.
5. If it is a closing parenthesis, pop operators to the output until an opening parenthesis is encountered. Pop and discard the opening parenthesis.
6. If there is more input, go to step 1.
7. If there is no more input, unstack the remaining operands.

Figure O.5 illustrates the use of this algorithm. This example should give the reader some feel for the power of stack-based algorithms.

References

DIJK63 Dijkstra, E. "Making an ALGOL Translator for the X1." In *Annual Review of Automatic Programming, Volume 4*. Pergamon, 1963.

Input	Output	Stack (top on right)
A + B × C + (D + E) × F	empty	empty
+ B × C + (D + E) × F	A	empty
B × C + (D + E) × F	A	+
× C + (D + E) × F	A B	+
C + (D + E) × F	A B	+ ×
+ (D + E) × F	A B C	+ ×
(D + E) × F	A B C × +	+
D + E) × F	A B C × +	+ (
+ E) × F	A B C × + D	+ (
E) × F	A B C × + D	+ (+
) × F	A B C × + D E	+ (+
× F	A B C × + D E +	+
F	A B C × + D E +	+ ×
empty	A B C × + D E + F	+ ×
empty	A B C × + D E + F × +	empty

Figure O.5 Conversion of an Expression from Infix to Postfix Notation