moves through instructions sequentially. After executing each instruction, the PC is incremented, and a new instruction is fetched from the address indicated by the PC. The major exception to this linear behavior is when branch instructions are encountered. Branch instructions exist specifically to override the sequential execution of instructions. When the instruction decoder fetches a branch instruction, it must determine the condition for the branch. If the condition is met (e.g., the ALU zero flag is asserted), the *branch target address* is loaded into the PC. Now, when the instruction decoder goes to fetch the next instruction, the PC will point to a new part of the instruction sequence instead of simply the next program memory location.

3.3 SUBROUTINES AND THE STACK

Most programs are organized into multiple blocks of instructions called *subroutines* rather than a single large sequence of instructions. Subroutines are located apart from the main program segment and are invoked by a subroutine call. This call is a type of branch instruction that temporarily jumps the microprocessor's PC to the subroutine, allowing it to be executed. When the subroutine has competed, control is returned to the program segment that called it via a return from subroutine instruction. Subroutines provide several benefits to a program, including modularity and ease of reuse. A modular subroutine is one that can be relocated in different parts of the same program while still performing the same basic function. An example of a modular subroutine is one that sorts a list of numbers in ascending order. This sorting subroutine can be called by multiple sections of a program and will perform the same operation on multiple lists. Reuse is related to modularity and takes the concept a step farther by enabling the subroutine to be transplanted from one program to another without modification. This concept greatly speeds the software development process.

Almost all microprocessors provide inherent support for subroutines in their architectures and instruction sets. Recall that the program counter keeps track of the next instruction to be executed and that branch instructions provide a mechanism for loading a new value into the PC. Most branch instructions simply cause a new value to be loaded into the PC when their specific branch condition is satisfied. Some branch instructions, however, not only reload the PC but also instruct the microprocessor to save the current value of the PC off to the side for later recall. This stored PC value, or *subroutine return address*, is what enables the subroutine to eventually return control to the program that called it. Subroutine call instructions are sometimes called *branch-to-subroutine* or *jump-to-subroutine,* and they may be unconditional.

When a branch-to-subroutine is executed, the PC is saved into a data structure called a *stack*. The stack is a region of data memory that is set aside by the programmer specifically for the main purpose of storing the microprocessor's state information when it branches to a subroutine. Other uses for the stack will be mentioned shortly. A stack is a *last-in, first-out* memory structure. When data is stored on the stack, it is *pushed* on. When data is removed from the stack, it is *popped* off. Popping the stack recalls the most recently pushed data. The first datum to be pushed onto the stack will be the last to be popped. A *stack pointer* (SP) holds a memory address that identifies the *top* of the stack at any given time. The SP decrements as entries are pushed on and increments at they are popped off, thereby growing the stack downward in memory as data is pushed on as shown in Fig. 3.5.

By pushing the PC onto the stack during a branch-to-subroutine, the microprocessor now has a means to return to the calling routine at any time by restoring the PC to its previous value by simply popping the stack. This operation is performed by a return-from-subroutine instruction. Many microprocessors push not only the PC onto the stack when calling a subroutine, but the accumulator and ALU status flags as well. While this increases the complexity of a subroutine call and return somewhat, it is useful to preserve the state of the calling routine so that it may resume control smoothly when the subroutine ends.

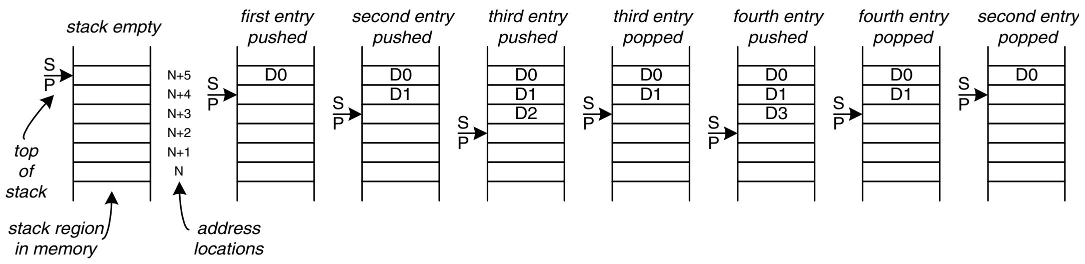

FIGURE 3.5 Generic stack operation.

The stack can store multiple entries, enabling multiple subroutines to be active at the same time. If one subroutine calls another, the microprocessor must keep track of both subroutines' return addresses in the order in which the subroutines have been called. This subroutine *nesting* process of one calling another subroutine, which calls another subroutine, naturally conforms to the last-in, first-out operation of a stack.

To implement a stack, a microprocessor contains a stack pointer register that is loaded by the programmer to establish the initial starting point, or top, of the stack. Figure 3.6 shows the hypothetical microprocessor in more complete form with a stack pointer register.

Like the PC, the SP is a counter that is automatically modified by certain instructions. Not only do subroutine branch and return instructions use the stack, there are also general-purpose push/pop instructions provided to enable the programmer to use the stack manually. The stack can make certain calculations easier by pushing the partial results of individual calculations and then popping them as they are combined into a final result.

The programmer must carefully manage the location and size of the stack. A microprocessor will freely execute subroutine call, subroutine return, push, and pop instructions whenever they are encountered in the software. If an empty stack is popped, the microprocessor will oblige by reading back whatever data value is present in memory at the time and then incrementing the SP. If a full stack is pushed, the microprocessor will write the specified data to the location pointed to by the SP and then decrement it. Depending on the exact circumstances, either of these operations can corrupt other parts of the program or data that happens to be in the memory location that gets overwritten. It is the programmer's responsibility to leave enough free memory for the desired stack depth and then to not nest too many subroutines simultaneously. The programmer must also ensure that there is symmetry between push/pop and subroutine call/return operations. Issuing a return-from-subroutine

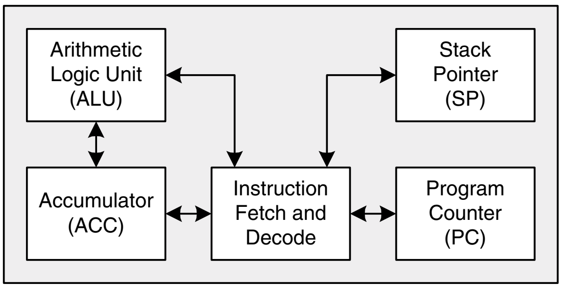

FIGURE 3.6 Microprocessor with stack pointer register.