6.5 The stack

We now look at one of the most important data structures in computer science, the *stack*, and describe the facilities provided by the 68K to support the stack (we provided a basic introduction in the previous chapter). A stack is a last-in-first-out *queue* with a *single end*, where items are added or removed. Unlike a conventional first-in-first-out queue (FIFO), the stack has only one end. The stack expands as items are added to it and contracts as they are removed. Items are removed from the stack in the reverse order to which they are entered. The point at which items are added to, or removed from, the stack is called the *top of stack* (TOS). The next position on the stack is referred to as *next on stack* (NOS). When an item is added to the stack it is said to be *pushed* on to the stack, and when an item is removed from the stack it is said to be *pulled* (or *popped*) off the stack.

Figure 6.24 presents a series of diagrams illustrating the operation of a stack as items A, B, C, D, and E, are added to it and removed from it.

Before we look at the stack's role in subroutines, we must mention the stack-based architecture that has been implemented by some special-purpose computers and by some experimental machines. Suppose a computer can transfer data between memory and the stack and perform monadic operations on the top item of the stack, or dyadic operations on the top two items of the stack. A dyadic operation (e.g. +, *, AND, OR) removes the top two items on the stack and pushes the result of the operation.

Figure 6.25 shows how an ADD instruction is executed by a stack-based computer. Figure 6.25(a) demonstrates a system

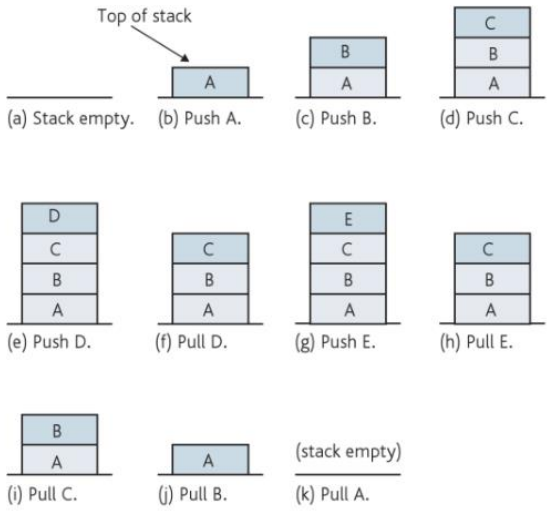


Figure 6.24 The stack.

with four data elements on the stack. When the ADD is executed, the element at the top of the stack is pulled (Fig. 6.25(b)) and sent to the adder. The *next* element (i.e. C, the old NOS) is now the new TOS. In Fig. 6.25(c) the element at the top of stack is pulled and sent to the adder. Finally, the output of the adder, D + C, is pushed onto the stack to create a new TOS.

Note how this ADD instruction doesn't have an operand unlike all the instructions we've described so far. A stack-based computer has so-called *addressless* instructions because they act on elements at the top of the stack.

The following example illustrates the evaluation of the expression (A + B)(C − D) on a hypothetical stack-based computer. We assume that the instruction PUSH pushes the contents of D0 onto the stack, ADD, SUB, and MULU all act on the top two items on the stack, and PULL places the top item on the stack in D0.

```
1.  MOVE A,D0    Get A in D0
2.  PUSH         Push it on the stack
3.  MOVE B,D0    Get B in D0
4.  PUSH         Push it on the stack
5.  ADD          Pull the top two items off the stack, add them, and push the result
6.  MOVE C,D0    Get C in D0
7.  PUSH         Push it on the stack
8.  MOVE D,D0    Get D
9.  PUSH         Push it on the stack
10. SUB          Pull the top two items off the stack, subtract them, and push the result
11. MULU         Pull the top two items off the stack, multiply them, and push the result
12. PULL         Pull the result off the stack and put it in D0
```

Figure 6.26 represents the state of the stack at various stages in the procedure. The number below each diagram corresponds to the line number in the program. Although the 68K and similar microprocessors do not permit operations

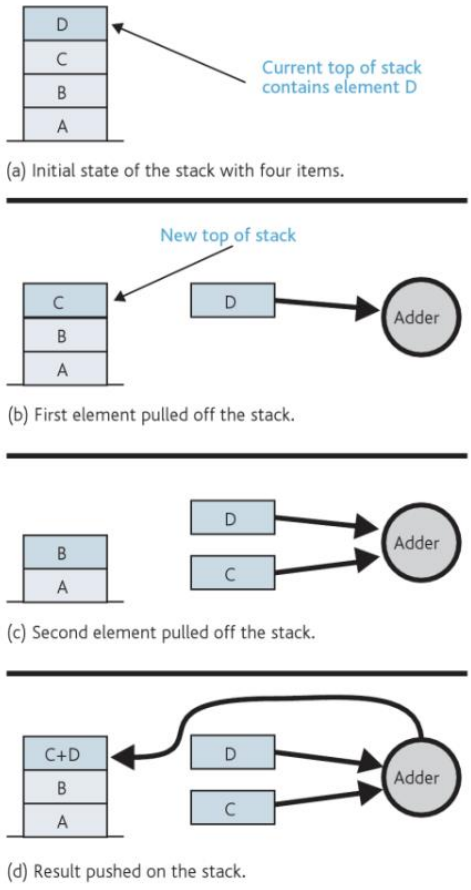


Figure 6.25 Executing an ADD operation on a stack machine.

on the stack in the way we've just described (e.g. ADD, SUB, MULU), special-purpose microprocessors have been designed to support stack-based languages. The 68K implements instructions enabling it to access a stack, although it's not a stack machine. Pure stack machines do exist, although they have never been developed to the same extent as the two-address machines like the 68K and Pentium.

6.5.1 The 68K stack

A hardware stack can be implemented as a modified shift register. When such a stack is implemented in hardware, the addition of a new item to the top of stack causes all other items on the stack to be pushed down. Similarly, when an item is removed from the stack, the NOS becomes TOS and all items move up.

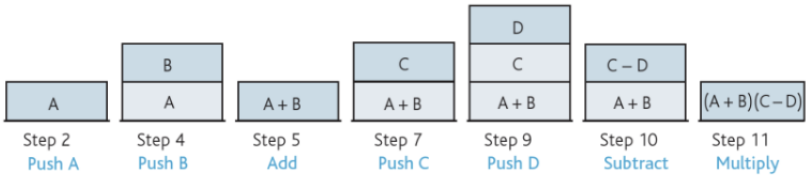


Figure 6.26 Executing a program on a stack machine.

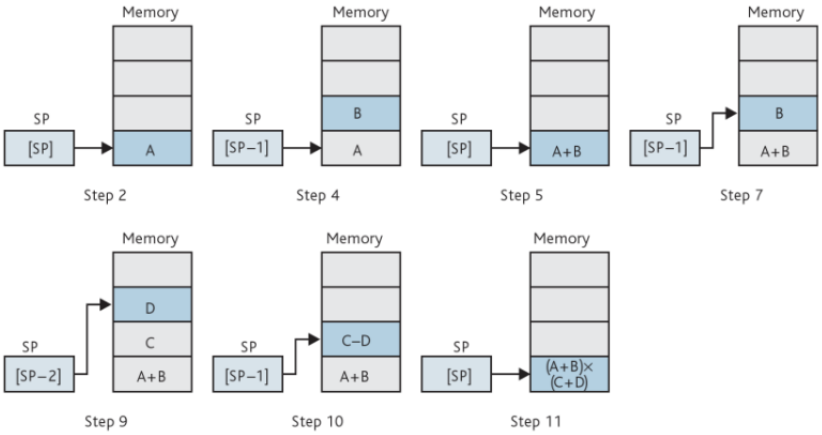


Figure 6.27 Executing the program of Fig. 6.26 on a machine with a stack pointer.

THE TWO 68K STACK POINTERS

The 68K has two A7 registers and, therefore, two *system* stack pointers. One A7 is the *supervisor* stack pointer and is associated with the operating system. The other is the *user* stack pointer and is associated with programs running under the operating system. The operating system controls the allocation of the computer's resources (memory and I/O), and is protected from errors caused by the less reliable user programs. A stack pointer dedicated solely to the operating system prevents user programs accessing and possibly corrupting the operating system's stack. Only one of these two A7s is accessible at a time, because the 68K is either running an operating system or it isn't.

Microprocessors don't implement a stack in this way and the items already on the stack don't move as new items are pushed and old ones pulled. The stack is located in a region of the main store and a *stack pointer* points to the top of the stack. This stack pointer points at the top of stack as the stack grows and contracts. In some microprocessors, the stack pointer points to the next free location on the stack, whereas in others, it points to the current top of stack.

Figure 6.27 demonstrates how the program illustrated in Fig. 6.26 is executed by a computer with a stack in memory and a stack pointer, SP.

The 68K doesn't have a special *system* stack pointer—it uses address register A7. We call A7 the *system* stack pointer because the stack pointed at by A7 stores return addresses during subroutine calls. Assemblers let you write either A7 or SP; for example, `MOVE.W D0, (A7)` and `MOVE.W D0, (SP)` are equivalent. The 68K can maintain up to eight stacks simultaneously, because all its address registers can be used as stack pointers.

In what follows, we use the 68K's stack pointer to illustrate the operation of a stack. You might expect the assembly language instruction that pushes the contents of D0 on the stack to be `PUSH D0`, and the corresponding instruction to pull an item from the stack and put it in D0 to be `PULL D0`. Explicit `PUSH` and `PULL` instructions are not provided by the 68K. You can use address register indirect with predecrementing addressing mode to push, and address register indirect with postincrementing addressing mode to pull.

Figure 6.28 illustrates the effect of a `PUSH D0` instruction, which is implemented by `MOVE.W D0, –(SP)`, and `PULL D0`, which is implemented by `MOVE.W (SP)+, D0`. The 68K's stack grows towards lower addresses as data is pushed on it; for example, if the stack pointer contains $80014C and a word is pushed onto the stack, the new value of the stack pointer will be $80014A.

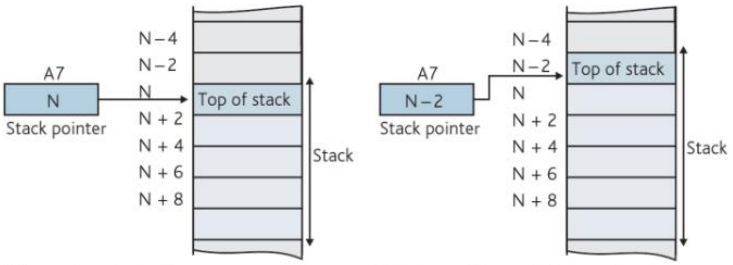
The 68K's push operation `MOVE.W D0,-(SP)` is defined in RTL as

```
[SP] ← [SP] - 2        Predecrement stack pointer to point to next free element
[[SP]] ← [D0]          Copy contents of D0 to the stack
```

and the 68K's pull operation `MOVE.W (SP)+,D0` is defined as

```
[D0] ←  [[SP]]         Copy the element on top of the stack to D0
[SP] ←  [SP] + 2       Postincrement the stack pointer to point to the new TOS
```

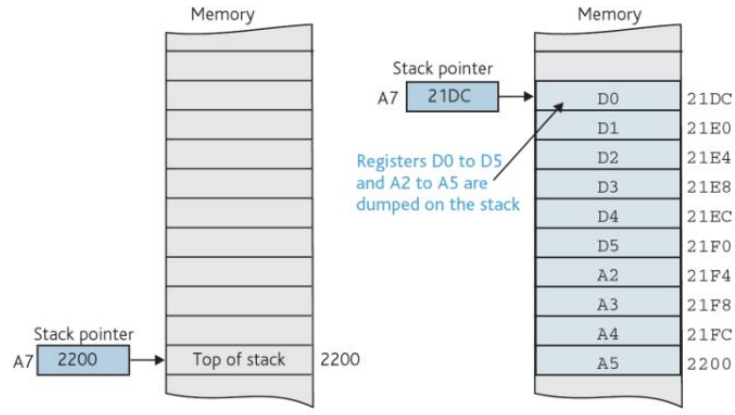
Push and pull operations use word or longword operands. A longword operand automatically causes the SP to be decremented or incremented by 4. Address registers A0 to A6 may be used to push or pull byte, `.B`, operands—but not the system stack pointer, A7. The reason for this restriction is that A7 must always point at a word boundary on an even address (this is an operational restriction imposed by the 68K's hardware).



(a) Snapshot of the 68K's stack.

(b) State of the stack after pushing a word by `MOVE.W D0,-(A7)`.

Figure 6.28 The 68K's stack.



(a) Initial state of stack.

(b) State of stack after `MOVEM.L D0-D5/A2-A5,-(A7)`.

Figure 6.29 The 68K's stack.

The 68K's stack pointer is decremented before a push and incremented after a pull. Consequently, the stack pointer always points at the item at the *top* of the stack; for example, `MOVE (SP)+,D3` pulls the top item off the stack and deposits it in D3. Note that `MOVE (SP),D3` copies the TOS into D3 without modifying the stack pointer.

When the stack shrinks after a `MOVE.W (SP)+,D0` operation, items on the stack are not physically deleted; they are still there in the memory until overwritten by, for example, a `MOVE.W D0,-(SP)` operation.

The stack can be used as a temporary data store. Executing a `MOVE.W D0,-(SP)` saves the contents of D0 on the stack, and executing a `MOVE.W (SP)+,D0` returns the contents of D0. The application of the stack as a temporary storage location avoids storing data in explicitly named memory locations. More importantly, if further data is stored on the stack, it does not overwrite the old data.

The 68K has a special instruction called *move multiple registers* (`MOVEM`), which saves or retrieves an entire group of registers. For example `MOVEM.L D0-D7/A0-A7,-(A7)` pushes all registers on the stack pointed at by A7. The register list used by `MOVEM` is written in the form `Di-Dj/ Ap-Aq` and

specifies data registers D*i* to D*j* inclusive and address registers A*p* to A*q* inclusive. Groups of registers are pulled off the stack by, for example, `MOVEM.L (A7)+,D0-D2/D4/A4-A6`. The most important applications of the stack are in the implementation of subroutines (discussed in the following section) and in the handling of interrupts. When autodecrementing is used, registers are stored in the order A7 to A0 then D7 to D0 with the highest numbered address register being stored at the lowest address. Figure 6.29 illustrates the effect of `MOVEM.L D0-D5/A2-A5,-(A7)`.