

A register can be cleared to 0 with an exclusive-OR operation. This is because $x \oplus x = 0$.

It is apparent from these examples that many other microoperations can be generated in the CPU. The most efficient way to generate control words with a large number of bits is to store them in a memory unit. A memory unit that stores control words is referred to as a control memory. By reading consecutive control words from memory, it is possible to initiate the desired sequence of microoperations for the CPU. This type of control is referred to as microprogrammed control. A microprogrammed control unit is shown in Fig. 7-8. The binary control word for the CPU will come from the outputs of the control memory marked “micro-ops.”

8-3 Stack Organization

LIFO

A useful feature that is included in the CPU of most computers is a stack or last-in, first-out (LIFO) list. A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved. The operation of a stack can be compared to a stack of trays. The last tray placed on top of the stack is the first to be taken off.

stack pointer

The stack in digital computers is essentially a memory unit with an address register that can count only (after an initial value is loaded into it). The register that holds the address for the stack is called a stack pointer (*SP*) because its value always points at the top item in the stack. Contrary to a stack of trays where the tray itself may be taken out or inserted, the physical registers of a stack are always available for reading or writing. It is the content of the word that is inserted or deleted.

The two operations of a stack are the insertion and deletion of items. The operation of insertion is called *push* (or push-down) because it can be thought of as the result of pushing a new item on top. The operation of deletion is called *pop* (or pop-up) because it can be thought of as the result of removing one item so that the stack pops up. However, nothing is pushed or popped in a computer stack. These operations are simulated by incrementing or decrementing the stack pointer register.

Register Stack

A stack can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words or registers. Figure 8-3 shows the organization of a 64-word register stack. The stack pointer register *SP* contains a binary number whose value is equal to the address of the word that is currently on top of the stack. Three items are placed in the stack: *A*, *B*, and *C*, in that order. Item *C* is on top of the stack so that the content of *SP* is now 3. To remove the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of *SP*. Item *B* is now on top of the stack

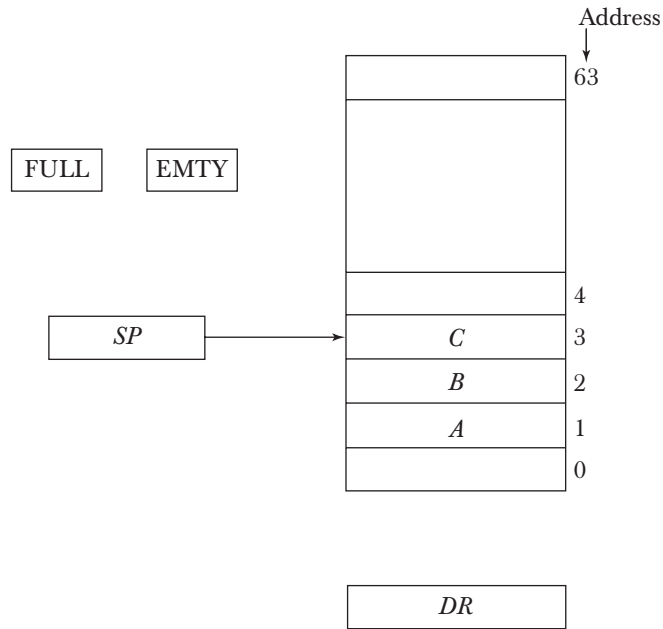


Figure 8-3 Block diagram of a 64-word stack.

since *SP* holds address 2. To insert a new item, the stack is pushed by incrementing *SP* and writing a word in the next-higher location in the stack. Note that item *C* has been read out but not physically removed. This does not matter because when the stack is pushed, a new item is written in its place.

In a 64-word stack, the stack pointer contains 6 bits because $2^6 = 64$. Since *SP* has only six bits, it cannot exceed a number greater than 63 (111111 in binary). When 63 is incremented by 1, the result is 0 since $111111 + 1 = 1000000$ in binary, but *SP* can accommodate only the six least significant bits. Similarly, when 000000 is decremented by 1, the result is 111111. The one-bit register *FULL* is set to 1 when the stack is full, and the one-bit register *EMPTY* is set to 1 when the stack is empty of items. *DR* is the data register that holds the binary data to be written into or read out of the stack.

Initially, *SP* is cleared to 0, *EMPTY* is set to 1, and *FULL* is cleared to 0, so that *SP* points to the word at address 0 and the stack is marked empty and not full. If the stack is not full (if *FULL* = 0), a new item is inserted with a push operation. The push operation is implemented with the following sequence of microoperations:

push

$SP \leftarrow SP + 1$

Increment stack pointer

$M[SP] \leftarrow DR$

Write item on top of the stack

If ($SP = 0$) then ($FULL \leftarrow 1$)	Check if stack is full
$EMPTY \leftarrow 0$	Mark the stack not empty

The stack pointer is incremented so that it points to the address of the next-higher word. A memory write operation inserts the word from DR into the top of the stack. Note that SP holds the address of the top of the stack and that $M[SP]$ denotes the memory word specified by the address presently available in SP . The first item stored in the stack is at address 1. The last item is stored at address 0. If SP reaches 0, the stack is full of items, so $FULL$ is set to 1. This condition is reached if the top item prior to the last push was in location 63 and, after incrementing SP , the last item is stored in location 0. Once an item is stored in location 0, there are no more empty registers in the stack. If an item is written in the stack, obviously the stack cannot be empty, so $EMPTY$ is cleared to 0.

pop A new item is deleted from the stack if the stack is not empty (if $EMPTY = 0$). The pop operation consists of the following sequence of micro-operations:

$DR \leftarrow M[SP]$	Read item from the top of stack
$SP \leftarrow SP - 1$	Decrement stack pointer
If ($SP = 0$) then ($EMPTY \leftarrow 1$)	Check if stack is empty
$FULL \leftarrow 0$	Mark the stack not full

The top item is read from the stack into DR . The stack pointer is then decremented. If its value reaches zero, the stack is empty, so $EMPTY$ is set to 1. This condition is reached if the item read was in location 1. Once this item is read out, SP is decremented and reaches the value 0, which is the initial value of SP . Note that if a pop operation reads the item from location 0 and then SP is decremented, SP changes to 111111, which is equivalent to decimal 63. In this configuration, the word in address 0 receives the last item in the stack. Note also that an erroneous operation will result if the stack is pushed when $FULL = 1$ or popped when $EMPTY = 1$.

Memory Stack

A stack can exist as a stand-alone unit as in Fig. 8-3 or can be implemented in a random-access memory attached to a CPU. The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer. Figure 8-4 shows a portion of computer memory partitioned into three segments: program, data, and stack. The program counter PC points at the address of the next instruction in the program. The address register AR points at an array of

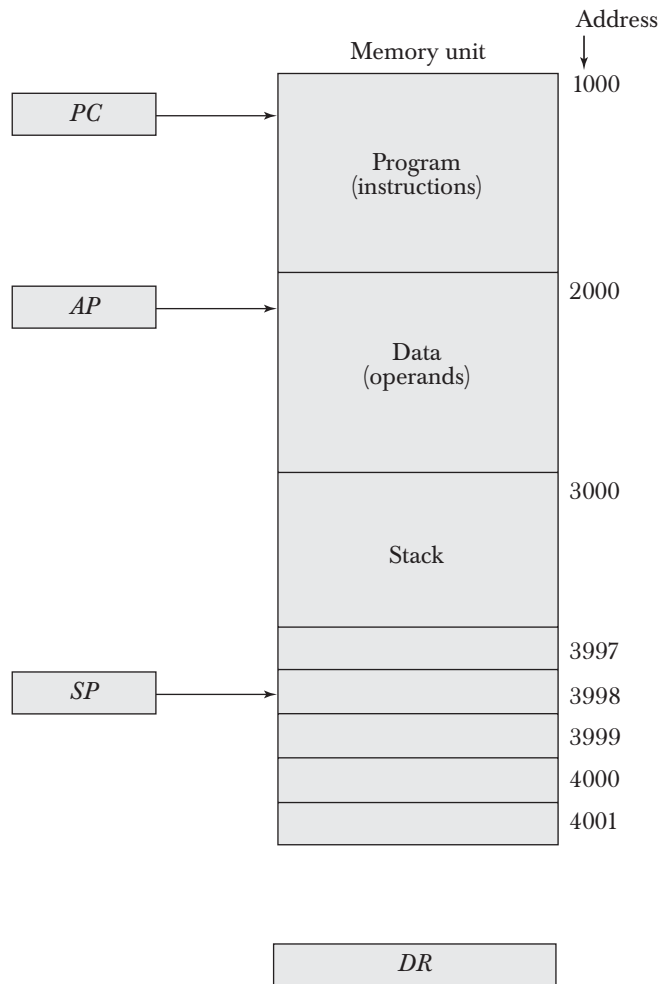


Figure 8-4 Computer memory with program, data, and stack segments.

data. The stack pointer *SP* points at the top of the stack. The three registers are connected to a common address bus, and either one can provide an address for memory. *PC* is used during the fetch phase to read an instruction. *AR* is used during the execute phase to read an operand. *SP* is used to push or pop items into or from the stack.

As shown in Fig. 8-4, the initial value of *SP* is 4001 and the stack grows with decreasing addresses. Thus the first item stored in the stack is at address 4000, the second item is stored at address 3999, and the last address that can be used for the stack is 3000. No provisions are available for stack limit checks.

We assume that the items in the stack communicate with a data register *DR*. A new item is inserted with the push operation as follows:

$$SP \leftarrow SP - 1$$

$$M[SP] \leftarrow DR$$

The stack pointer is decremented so that it points at the address of the next word. A memory write operation inserts the word from *DR* into the top of the stack. A new item is deleted with a pop operation as follows:

$$DR \leftarrow M[SP]$$

$$SP \leftarrow SP + 1$$

The top item is read from the stack into *DR*. The stack pointer is then incremented to point at the next item in the stack.

stack limits

Most computers do not provide hardware to check for stack overflow (full stack) or underflow (empty stack). The stack limits can be checked by using two processor registers: one to hold the upper limit (3000 in this case), and the other to hold the lower limit (4001 in this case). After a push operation, *SP* is compared with the upper-limit register and after a pop operation, *SP* is compared with the lower-limit register.

The two microoperations needed for either the push or pop are (1) an access to memory through *SP*, and (2) updating *SP*. Which of the two microoperations is done first and whether *SP* is updated by incrementing or decrementing depends on the organization of the stack. In Fig. 8-4 the stack grows by *decreasing* the memory address. The stack may be constructed to grow by *increasing* the memory address as in Fig. 8-3. In such a case, *SP* is incremented for the push operation and decremented for the pop operation. A stack may be constructed so that *SP* points at the next *empty* location above the top of the stack. In this case the sequence of microoperations must be interchanged.

A stack pointer is loaded with an initial value. This initial value must be the bottom address of an assigned stack in memory. Henceforth, *SP* is automatically decremented or incremented with every push or pop operation. The advantage of a memory stack is that the CPU can refer to it without having to specify an address, since the address is always available and automatically updated in the stack pointer.

Reverse Polish Notation

A stack organization is very effective for evaluating arithmetic expressions. The common mathematical method of writing arithmetic expressions imposes difficulties when evaluated by a computer. The common arithmetic expressions