

SUBROUTINES AND STACKS

(R. Horvath, Introduction to Microprocessors, Chapter 8)

Microprocessors support one or more special structures in read-write memory known as stacks. These structures are used

- to retain register information whenever the processor interrupts its regular flow of fetching and executing instructions in their normal sequence.
- to call subroutine or
- whenever some external device signals that it needs special attention.
- to retain information for future reference.

8.1 SUBROUTINES

A subroutine is a specialized program module that may be called upon from elsewhere within the program.

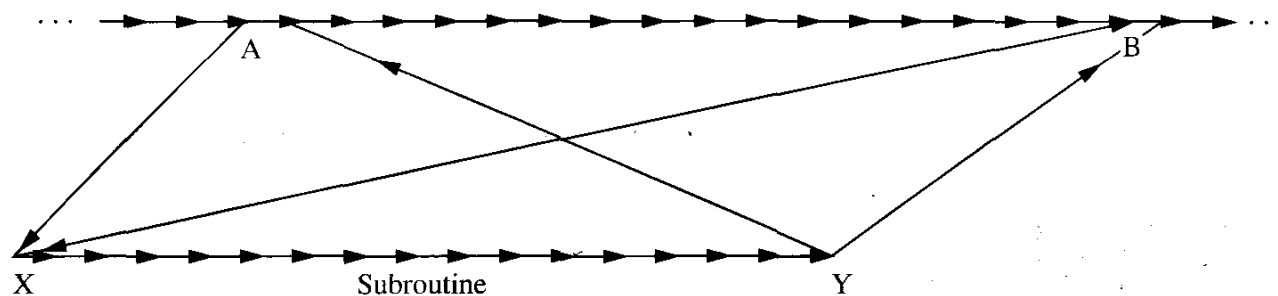
The original intent was to allow the subroutine module to be coded only once but called upon many times during the course of execution of the program.

The resulting reduction in program size is one of the reasons.

Modern programming techniques, so-called modular programming, however, encourage the use of subroutines throughout a program, even in instances where they are called upon only once.

At point A an instruction in the program calls upon the subroutine and so the processor must branch to it at point X.

Main program sequence:



Upon the completion of the subroutine at point Y, the processor must branch back to just after point A in order to continue along the main sequence. Later, at point B, the subroutine is called upon again.

Thus, upon its completion, the subroutine in this example must branch to one of two different locations, to just after *A* or to just after *B*. The branch instructions described earlier is capable of that behavior.

An entirely different type of branch instruction is necessary, one which can return the program back to the calling location, wherever it may have been.

When executing the special return instruction, the processor must obtain the address from that structure and effect the branch back to the main sequence.

8.2 THE LAST IN FIRST OUT STACK (LIFO)

A simple structure to retain the return address from a subroutine would be a single register. However, with only one register, no subroutine could ever call another subroutine.

A group of several registers could accommodate several subroutine calls of other subroutines (*nested subroutines*). A *last in first out* stack of registers (a LIFO stack) would be required.

Upon completion of the innermost or most recently called subroutine, the last stored return address would be retrieved from the LIFO stack of registers.

Each register in the stack is numbered with an address. An auxiliary register called the stack pointer points to the last used register in the stack by holding the number of that register.

When the current subroutine is completed, the processor uses the stack pointer to locate the register containing the required return address.

After the processor retrieves the return address, the pointer is automatically changed to point to the next lower register in the stack in anticipation of the need for the next lower nested return address.

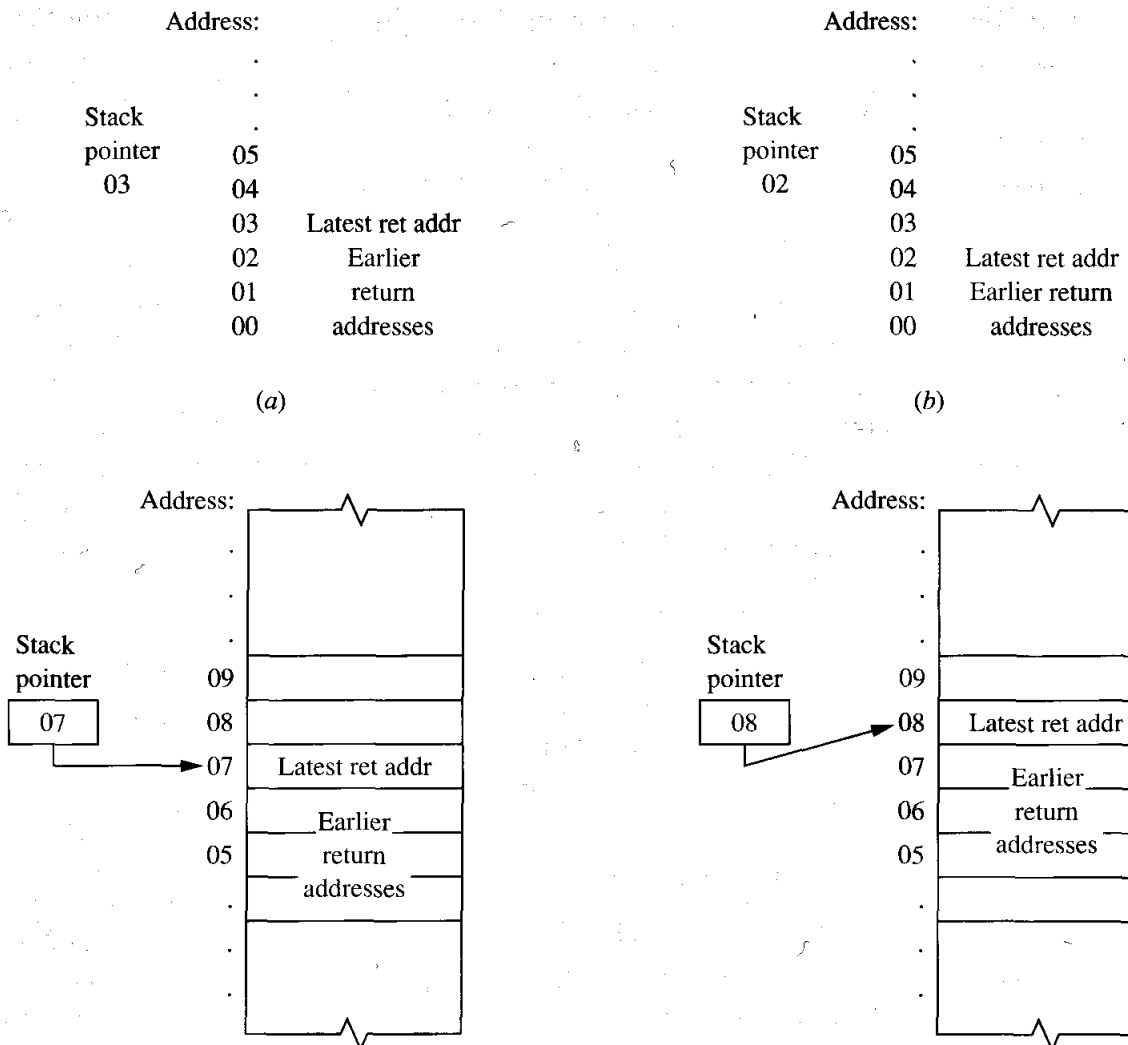


FIGURE 8.2

A LIFO register stack: (a) before a return, (b) after the return, (c) before a call, (d) after the call.

This is shown in Figure 8.2a and b. When a new subroutine call is executed, the processor saves the new return address on the top of the stack. The stack pointer is automatically incremented to point to this next higher register in the stack, as shown in Figure 8.2c and d. Thus, the stack pointer changes its content whenever a new return address is saved (pushed) onto the stack or retrieved (pulled) from the stack. In this way the stack pointer always points to the top of the stack, the last written register.

Although stacks may be implemented with processor registers as in Figure 8.2, it is more common to implement them in read/write memory.

8.3 STACK APPLICATIONS

The primary application for a LIFO stack :

- to store return addresses during subroutine executions.
- to control the transfer of data into or out of a computer uses an externally generated signal called an interrupt. Upon detecting the signal, the processor interrupts its current activity in order to attend to the external source of this signal.
- The status of the activity which was interrupted must be saved so that, after attending to the interrupt, the processor may resume where it left off. The stack provides a convenient place to store this status while servicing the interrupt.
- Thus, a processor may save the contents of the program counter, the condition code register, and other processor registers on the stack while it responds to an interrupt.
- **to** transfer parameters to a subroutine
- to save data temporarily from within the program.
-

Many processors provide a pair of instructions called *push/pull* or *push/pop*. When one of these instructions is executed the operand is pushed onto the stack or pulled off the stack. Push/pull instructions have an advantage over other types of load/store instructions in that they require no addressing specification.

The stack is also a convenient place for a subroutine to store intermediate results during its execution. Instructions accessing this temporary storage area need to include no absolute addresses. In addition, the storage area can be allocated to a routine when it is called and then deallocated upon its return, a process known as *dynamic allocation*.

Dynamic allocation of memory has another important application, however. It permits a routine to be interrupted by an input/output device and then to be called while that device is being serviced.

8.6 STACKS IN THE MC68000

In the MC68000 two address registers, A7 and A7', share the title of *system stack pointer (SP)*. The MC68000 operates in one of two privilege states, the *user* state and the *supervisor* state.

While in the supervisor state, all of the system resources are available to a program, any of the instructions may be executed, and reference to register A7 will access register A7', the supervisor stack pointer.

While in the user state, certain of the system resources may be blocked from access, several of the instructions may not be executed, and reference to register A7 will access register A7, the user stack pointer.

When in the user state, the processor can perform operations *using* the system stack, *but it is not permitted to read from or write to either system stack pointer*.

The user may define additional stacks for data storage by using the other address registers (A0 through A6) as stack pointers. A stack may be used to store data by using the address register indirect auto increment/decrement modes.

- to push the entire content of D0 onto the A4 stack, use the instruction `MOVE.L D0,-(A4)`.
- To pull the word off the top of the A1 stack into register D3, use `MOVE (A1)+,D3`.
- Since the MC68000 has no ordinary push/pull instructions, this technique must be used to save or retrieve data from any stack, including the system stack (A7 or SP).
-

The `MOVEM` instruction (move multiple registers) may be used to push or pull several registers in a single instruction.

The instruction `MOVEM A4/A6/D3-D5,-(A7)` would save the (word) contents of registers A4, A6, D3, D4, and D5 on the system stack. The order in which the registers are pushed onto the stack is independent of the order in which they are specified in the instruction.

The stacking order (push order) for MOVEM with the predecrement destination mode is as follows: address registers in reverse numerical order, and then data registers in reverse numerical order.

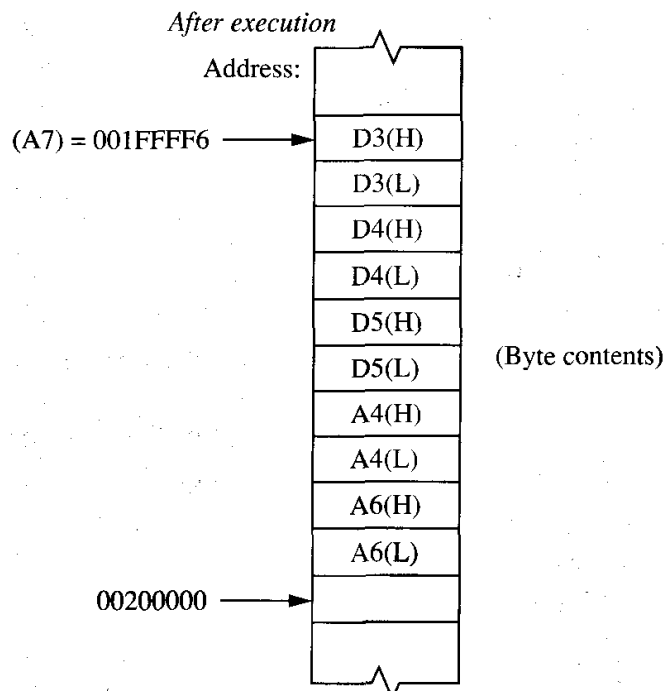
Instruction: MOVEM A4/A6/D3-D5, -(A7)

Before: A7

0020	0000
------	------

After: A7

001F	FFF6
------	------



The instruction MOVEM (A7)+, A4/A6/D3-D5 will restore the registers with the proper contents.

The MC68000 instructions for calling subroutines are BSR (branch to subroutine) and JSR (jump to subroutine). The BSR instruction uses relative addressing, specifying the target address in terms of its (8- or 16-bit) offset from the current program counter content. As such, it may not be capable of reaching distant subroutines. JSR allow the use of any of the control modes of addressing and is thus capable of reaching a subroutine anywhere in the program.

All subroutines must end with either the instruction RTS (return from subroutine) or RTR (return and restore condition codes).

RTS pulls the top four bytes off the system stack and loads them into the program counter.

RTR is used when the subroutine has saved the condition codes on top of the return address in the stack in order to preserve the status of the calling program. RTR pulls the top word of the stack into the condition code register and the next two words into the program counter.

Among the other MC68000 stack instructions are PEA (push effective address), LINK (link) and UNLK (unlink), RTI (return from interrupt), RTE (return from exception processing), will be discussed later.

8.7 EXAMPLES OF STACK OPERATIONS IN THE MC68000

8.7.1 Initialization of the MC68000 Stack Pointers

Prior to the use of any stack, the stack pointer register must be initialized to some address in read/write memory where the stack will have room to grow as it is used. Since stacks fill toward lower addresses, this initial address should be toward the upper end of the available space.

```
1.      ORG      $200000
2. INIT  MOVEA.L  #00F00000,SP 200000    2E 7C 00 F0 00 00
3.      JSR      DOIT          200006    4E B9 00 00 12 34
4. NEXT MOVE     D0,D4          20000C    38 00
```

FIGURE 8.15

MC68000 program segment to initialize and use the supervisor stack.

In line 2 the system stack pointer is initialized to 00F00000.

When the subroutine DOIT (with an assumed starting address of 001234) is called in line 3, the return address of 20000C is saved on the stack.

8.7.2 Passing Parameters in the MC68000

One way in which the calling program may pass parameters to a subroutine is to pick up or create the parameters in processor registers and allow them to remain there while calling the subroutine. The program segment in Figure 8.17 illustrates this.

```

MOVE    PARAM1, D0    PASS PARAM1 IN D0
MOVE    PARAM2, D1    AND PARAM2 IN D1
JSR     USEM

```

FIGURE 8.17 Passing parameters in MC68000 registers

Another way to pass parameters is to push them onto the stack prior to calling the subroutine. When this approach is used with the system stack the return address will be pushed on top of the parameters during the call. This requires that the subroutine reach down into the stack to access the parameters, as shown in the sample program listed in Figure 8.18.

In the calling program:

```

MOVE    PARAM1,-(A7)  STACK PARAM1
MOVE    PARAM2,-(A7)  STACK PARAM2
MOVE    PARAM3,-(A7)  STACK PARAM3
MOVE    PARAM8,-(A7)  STACK PARAM8
JSR     USEM

```

In subroutine USEM:

```

MOVE    4(A7),D1      PICK UP PARAM8 INTO D1
MOVE    12(A7),D3     PICK UP PARAM4 INTO D3

```

