Z8000™ CPU Programmers Guide

Zilog

Application Note

July 1981

Zilog

Z8000 CPU PROGRAMMER'S GUIDE

The purpose of this application note is to demonstrate how the features of the Z8000 CPU can be used to solve typical software problems. The first half focuses on specific programming techniques. In the second half, fully worked-out programs are presented for several important or illustrative problems.

PART I

PROGRAMMING TECHNIQUES

1.1 INTRODUCTION

A goal of programming is to allow computer users to deal with the high-level operations of their applications and to escape from the details of machine design and behavior. Many programming techniques have been designed with this goal in mind. This section introduces some widely used programming techniques and shows how they are implemented using the Z8000 architecture and instruction set.

1.2 DATA TYPES

All computer applications are based upon the interpretation of collections of bits--as numbers, text, logical flags and so forth. The term **data type** refers to a bit collection of specified size and interpretation.

Every computer provides direct support for some data types, and the programmer provides programs to support the manipulation of other desired data types. The Z8000 architecture provides direct support for several frequently used data types and the instructions for performing the operations associated with them. These are described below.

Bits. A two-valued logical flag is the simplest useful interpretation of a bit collection, and its natural size is one bit. Unlike many earlier computers, the Z8000 has instructions that allow any bit in memory or in any general-purpose register to be set, tested or cleared. Thus, any bit can be used as a logical flag, and flags can be packed into words or bytes without undue increase in processing overhead. An important application of this idea is a bit table, an array of 1-bit logical flags stored in consecutive bits of consecutive bytes of memory.

Digits. An important bit collection is a number, and an important special case of numbers is a decimal or hexadecimal digit. These are most conveniently represented by collections of four bits (occasionally referred to as nibbles). The Z8000 supports digits with the RRDB, RLDB, and DAB instructions, and the D and H bits in the Flags register.

Bytes. A collection of eight bits is called a byte. Almost all Z8000 instructions that take arguments have byte versions. (The Push, Pop, Multiply, and Divide instructions are the only important exceptions.) The two principal interpretations of bytes are as signed whole numbers and as codes for text characters. These interpretations are not enforced by the hardware, but some Z8000 features are designed with one or the other interpretation in mind. For example, the Translate and Test instruction and the P (parity) bit in the Flags register support the text data type, while the arithmetic instructions support signed whole numbers. The Z8000 has 16 byte registers.

Words. A collection of 16 bits is called a word. Almost all argument-taking instructions have word versions. (The Block Translate and Test instructions and the decimal arithmetic support instructions are the only exceptions.) The principal interpretations of words are as signed and unsigned whole numbers, Z8000 instructions, index values, and nonsegmented addresses. The Z8000 provides 16 word registers.

Long Words. A collection of 32 bits is called a long word. The principal interpretations of long words are as segmented addresses and as signed and unsigned whole numbers. The Z8000 provides long-word versions of its Load, Push, and Pop instructions and supports 32-bit signed whole numbers with long-word versions of its four main arithmetic operations, add, subtract, multiply and divide. The Z8000 provides eight long-word registers.

Quadruple Words. The Long Multiply and Long Divide instructions involve the use of 64-bit signed whole numbers. Four quadruple-word registers are provided for this purpose.

In addition to these data types, several other collections of bits are manipulated by certain Z8000 instructions.

Addresses. The LDA and LDAR instructions generate and save addresses. Addresses are words or long words, depending upon the segmentation mode of the CPU at the time of execution.

Register Sets. The LDM instruction manipulates register sets during the movement of information between general-purpose registers and memory. A register set consists of from 1 to 16 words stored in contiguous memory locations or in consecutive word registers.

Data blocks. The Z8000 block instructions manipulate data blocks, which can be from 1 to 65,536 words or bytes stored in contiguous memory locations. An important special case of a data block is a text string.

As subsequent examples illustrate, this large selection of data types offers Z8000 programmers simple approaches to solving a wide variety of programming problems.

1.3 ADDRESSING MODES

The Z8000 addressing modes were chosen and designed with the programmer's needs in mind. Here is a brief summary of the ideas behind these modes.

Direct Addressing. With direct addressing, the actual memory address of the argument is contained in the instruction. This is especially useful in programs assembled by hand and in "patches."

Register Addressing. This addressing mode allows fast access to intermediate results. Almost all two-operand instructions require the use of register addressing for one of the operands.

Immediate Addressing. Immediate addressing is similar to direct addressing, but the actual value of the argument rather than its address is contained in the instruction. Immediate addressing can only be used for source arguments.

Indirect Register Addressing. In this mode the address of the argument is in an address register (a word or long-word register, depending upon the segmentation mode). Its variants, the Autoincrement and Autodecrement modes, are used with the Push and Pop instructions to implement stacks, and with the block instructions to effect operations on sets of contiguous words or bytes in memory. Indirect register addressing is used when addresses are passed as arguments to subroutines and to implement more elaborate access techniques, such as linked lists. Following is a simple example of its use--a loop to read successive bytes of memory until a zero terminator is found and to replace each byte with a modified value:

```
       LDA RR2,X                !RR2 = address of text array!
LOOP:  LDB RH0,@RR2             !Fetch next character!
       TESTB RH0
         JR Z,ENDLP             !Done when NUL reached!
       !(Modify the character)!
       LDB @RR2,RH0             !Replace character by modified character!
       INC R3                   !Point at next character!
       JR LOOP
ENDLP: . . .
```

In this example, RR2 is used as an address register to point at (that is, contain the address of) successive bytes of a text string. Notice that the instruction

```
INC R3
```

is used to point to the next byte. This takes advantage of the way segmented addresses are stored in registers but assumes that the text string does not extend outside of the memory segment. A later example deals with arrays that extend beyond one segment.

Notice also that the instruction

```
LDA RR2,X
```

is used to set the contents of the address register RR2. An alternative instruction is

```
LDL RR2,#X
```

but it should be avoided, because it needlessly ties the code to a specific segmentation mode.

Indexed addressing. In the **Indexed Addressing** mode, a fixed address is stored in the instruction and a **displacement is** stored in a register. This is required when an array is **being processed using** a varying index. For example, consider the following FORTRAN **instructions:**

```
   DO 13 I=N1,N2
13 TABLE(I) = TABLE(I)+I
```

This can be implemented using indexed addressing **as shown in Figure 1-1.**

```
!Assume that the registers have been set:
        R0 contains N2
        R1 contains N1  (R1 will be I)
!
        LD R3,R1                !Use R3 for actual offset!
        SLA R3                  !Assume two-byte entries!
LOOP:   CP R1,R0                !Is I > N2 yet?!
          JR GT,DONE            !Done if so!
        LD R2,TABLE-2(R3)       !TABLE(I) - FORTRAN arrays start at 1!
        ADD R2,R1               !TABLE(I)+I!
        LD TABLE-2(R3),R2       !Replace original TABLE(I) value!
        INC R1                  !Increment I!
        INC R3,#2
        JR LOOP
DONE:   . . .
```

Figure 1-1

Two-dimensional arrays can be handled easily by a program that computes the offset associated with an index pair. For example, suppose that the M x N array of bytes TABLE is stored consecutively in memory as follows:

TABLE(1,1), TABLE(2,1),...,TABLE (M,1), TABLE(1,2),...

Each column is a one-dimensional array, and these one-dimensional arrays are stored end to end in contiguous bytes of memory. (This format is standard in FORTRAN.) A two-dimensional array can be viewed as a one-dimensional array of dimension MN, and the element TABLE(I,J) of the two-dimensional array is the element TABLE([J-1]*M+I) in the one-dimensional array. If R1 contains I-1, R2 contains J-1, and R3 contains M, then the following code loads TABLE(I,J) into RH0:

```
LD R5,R2                !RR4 = (xxx,J-1)!
MULT RR4,R3             !RR4 = (0,[J-1]*M)!
ADD R5,R1               !R5  = [J-1]*M+[I-1]!
LDB RH0,TABLE(R5)
```

This code assumes that MN $\leq$ 65,536. If this is not true, then indexed addressing cannot be used directly and the assumption that the columns of TABLE are stored end to end cannot be made. Instead, J is used as an index to a table of memory addresses (called a "dope vector"), and each of these addresses is the start of the corresponding column. If R1 contains I-1, R2 contains J-1 and the table of column base addresses is at an address contained in RR4, then the following code loads TABLE (I,J) into RH0:

```
        LD R3,R2
        SLA R3,#2               !R3 = 4*(J-1)!
        LDL RR6,RR4(R3)         !RR6 = address of Jth column!
        LDB RH0,RR6(R1)
```

This code uses base-indexed addressing (see below). It is so efficient that it can be used even when MN $\leq$ 65,536.

For nonsegmented operation, indexed addressing can be used to simulate based addressing (see below), since addresses and offsets are both 16 bits. For example,

```
ADD R0,8(R15)
```

adds the fifth word of the stack to R0. (NOTE: If separate data and stack spaces are used, this technique does not work. When R15 is used in the indexed addressing mode, the status outputs ST_3-ST_0 reflect data reference, not stack reference.)

In segmented mode, the same technique can be used if the segment number is known when the program is assembled. For example, if the stack has been assigned to segment 12 (that is, R14 contains 0C00), then

```
ADD R0,<<12>>8(R15)
```

adds the fifth word on the stack to R0.

Use of indexed addressing to simulate based addressing is helpful because based addressing is available only with the Load instruction.

Based Addressing. Based addressing specifies the address of an argument as the sum of a displacement contained in the instruction and a base address contained in an address register. For example,

```
LD R0,RR14(#8)
```

can be used in segmented mode to access the fifth word of the stack.

The Based Addressing mode is the key to a subroutine argument-passing convention that uses a stack (see Section 1.17).

Based addressing is useful in accessing items in records or more general data structures of predefined format, especially when the address of the record in memory is not known in advance. For example, if a number of 80-character records have been read into memory end to end starting at a location specified in RR2, then the code shown in Figure 1-2 steps through the records until one is found in which the seventy-third character is equal to 41H.

```
LOOP:    LDB RH0,RR2(#72)        !Get 73rd character!
         CPB RH0,#%41            !Compare with %41!
           JR EQ,ENDLP           !Done if equal!
         ADD R3,#80              !Otherwise, point at next record!
         JR LOOP
ENDLP:   . . .
```

Figure 1-2

Base Indexed Addressing. Base indexed addressing takes both the base address and the displacement from registers. One example of it was shown above in the code to handle large two-dimensional arrays. Other examples are shown in Part 2.

Base indexed addressing is also useful in a generalization of the record or data structure example given in Figure 1-2. For example, if the termination condition were the presence of 41H in any of positions 73 through 80, the code of Figure 1-2 would appear as shown in Figure 1-3.

```
LOOP:    LD R4,#72               !Set to 73rd position!
LOOP1:   LDB RH0,RR2(R4)         !Get R4th character!
         CPB RH0,#%41            !Compare with %41
           JR EQ,ENDLP           !Done if equal!
         INC R4                  !Otherwise, give R4 next index!
         CP R4,#80               !Compare position with last!
           JR LT,LOOP1           !If not past last, try next position!
         ADD R3,#80              !Otherwise, point at next record!
           JR LOOP
ENDLP:   . . .
```

Figure 1-3

Relative Addressing. This is a variant of based addressing in which the base register is always the Program Counter. It helps the programmer produce position-independent code (see Section 1.6), and it leads to more compact code in many cases. Also, if separate data and instruction memories are used, the LDR instruction is the only way to refer to a constant that is assembled as part of the program (except immediate data in instructions).

Further examples using the Z8000 addressing modes are given in the following sections.

1.4 STACKS

A stack is a last-in, first-out (LIFO) buffer of finite but unspecified size. It is like a stack of plates on a table in a room: plates can only be added to or removed from the top and while there is no preset maximum number of plates, the room does have a ceiling. Sometimes the metaphor used is a stack of plates on a spring in a well (as at a steam table); this accounts for the names PUSH and POP used for the operations of adding or removing items, but in the usual computer implementations the items stay fixed like plates on a table.

In the Z8000, stacks are implemented as arrays of declared fixed sizes, but an external memory-mapping facility allows stacks to be open ended, with additional memory allocations made as needed. The Push and Pop instructions are designed to work with stacks that grow downward; that is, the first item on the stack occupies the highest-numbered memory location. Programs, on the other hand, grow upward; that is, as each instruction is added to the program or as program modules are linked together, higher and higher-numbered addresses are used. This provides an efficient way for a program and a stack to share a given block of memory. The program can begin at the lowest-numbered address and grow upward as developments increase its size; the stack can begin at the highest-numbered location and grow downward as the program is executed. This is the most flexible and efficient use of the space. If there is room for both the program and the stack in memory, then memory is automatically allocated successfully.

A stack in the Z8000 uses an address register to keep track of the location of the top item (the lowest-numbered item). The stack register always contains the address of the top item because of the way PUSH and POP work. PUSH first decrements the stack register by 2 or 4, causing it to point at the next free-word or long-word location and then stores its argument at that location. POP first fetches the item pointed to by the stack register, then increments the stack register.

Reference to items on a stack can be made using the Based or Base Indexed Addressing mode. For example, if RR4 is a stack register, then RR4(#0), RR4(#2), and RR4(#4) refer to the top, second, and third words on the stack, respectively. Also, as previously explained, indexed addressing can be used to refer to stack items when the stack's segment number is known at assembly time. Reference to stack items is illustrated in the Section 1.7, Subroutines.

The most common use of stacks is for dynamic allocation of temporary storage space. The two pieces of code in Figure 1-4 show how a program can accumulate words for future processing. The first uses fixed temporary storage; the second uses a stack.

```
        !Accumulating words in a fixed buffer!

             CLR R4            !Word counter!
             LDA RR2, BUF      !RR2 always points at next free location!
        LP:  CALL GETWD        !Get next word!
               JR C,DONE       !If C set, no more to get!
             LD @RR2,R0        !Store word, increment pointer!
             INC R3,#2
             INC R4            !Count the word!
             JR LP

    DONE:    . . .

        !Accumulating words on a stack!

             CLR R4            !Word counter!
        LP:  CALL GETWD        !Get next word!
               JR C,DONE       !If C set, no more to get!
             PUSH @RR2,R0      !Store word, increment pointer!
             INC R4            !Count the word!
             JR LP

    DONE:    . . .
```

Figure 1-4

In the first piece of code, a buffer called BUF is allocated to the program at assembly time. Each time this code is executed, words are stored in this buffer, starting at the beginning of the buffer. The second piece of code has no storage of its own; every time it is executed it stores words on the stack controlled by RR2. It is assumed that the system initializes this stack before the process including this code begins running.

Using a stack in this way has several advantages:

- The total amount of space needed by the stack is usually less than the amount required by fixed allocation.
- Storage mangement is separate from the implementation of the function. This tends to simplify the implementation of functions.
- Program functions can be encoded in ROM more easily and management of RAM can be localized.
- It is easier to make program functions shareable (see below); in the preceding example, several different sets of words might have been accumulated in different parts of the stack by different calls on the code. This would not be possible with the fixed-buffer accumulation.

There are also some disadvantages to using stacks in this way. In general, programs that use a stack must leave it exactly as they found it; every item pushed onto the stack must be popped off before completion of the program. This is because the same stack used by the program that calls the given program is also used by programs called by the given program. For example, consider the following code:

```
PUSH @RR4,RO
CALL SUBR
POP RO,@RR4
```

This is a common means of saving a value, in this case RO, that would otherwise be destroyed by the intermediate operation, in this case CALL SUBR. But this procedure fails if the SUBR routine does not leave the stack controlled by RR4 exactly as it was found.

The requirement that each program regulate its stack use can make checkout difficult, since a subroutine's failure in stack management can lead to anomalies in the behavior of the calling program. The symptom and cause can be in seemingly unrelated portions of the program. Also, there is a dedicated stack register used for subroutine calling; failure in its management can cause symptoms that are difficult to recognize and usually interferes with the standard checkout procedures.

Dynamic allocation of temporary storage leads to another checkout problem: it is difficult to examine memory after the fact to look for the causes of anomalous behavior. A desired piece of information may have been overwritten, and it is difficult to determine where a given program stored its intermediate or temporary data.

In general, stack use is not as flexible as the use of dedicated storage. For example, in the preceding code, once the words are accumulated in BUF, they are processed any way the programmer desires. Indexed addressing of the form

```
ADD R1,BUF(R2)
```

makes the fixed buffer a random access memory. With a stack, on the other hand, only the top item is easily available. Other items can be accessed using based or base indexed addressing of the form

```
LD R1,RR6(#2)
LD R2,RR6(R3)
```

If the stack segment number is known when the program is assembled, the Indexed Addressing mode can be used, as in the preceding ADD example. For example:

```
ADD R1, <<seg no>>2(R7)
```

adds the next-to-last word received to R1.

The stack addressing methods described allow items in memory to be examined without giving up their places (as happens with POP), but the offsets (#2 or

R3 in the above lines) are measured from the top of the stack, that is, from the last item placed there. To process the items in a first-in, first-out (FIFO) order requires a complicated computation that can lead to errors. For example, referring to the sample code of Figure 1-4 for accumulating words on a stack, Figure 1-5 shows the code at DONE that allows the words to be examined in the order received.

```
DONE:     SLA R4            !Multiply by # bytes/word!
            JR Z,FINIS      !No words to examine!
GETNXT:   DEC R4,#2         !Convert count to offset!
          LD RO,RR2(R4)     !Fetch the word from the RR2 stack!
         !(Process the word)!
          TEST R4           !R4 contains # of bytes remaining!
            JR NZ,GETNXT

FINIS:    . . .
```

Figure 1-5

Stack initialization is straightforward. The stack register must be set to the address one word above (that is, at a higher-numbered address than) the first word to be used by the stack. This works regardless of whether words or long words are used. (In fact, there is no problem with mixing words and long words on a stack, as long as any item pushed with a PUSHL instruction is popped with a POPL instruction.) So, for example, if a stack uses locations F000-FFFF of segment 6, the first word used by the stack is at location FFFE. The stack register should be initialized to segment 6, offset zero.

Boundary protection has two aspects: overflow and underflow. Overflow occurs when all locations assigned to a stack have been filled and another push is attempted. Underflow results from an attempt to pop items from an empty stack. The Push and Pop instructions provide no direct support for boundary protection. This is achieved in software by using push and pop subroutines that check for overflow or underflow before pushing or popping. An external memory management facility can also help detect stack overflow.

The preceding discussion applies to all stacks in the Z8000. The Z8000 automatically uses stacks for subroutine calling and for saving CPU status on traps and interrupts, and for these purposes an implicit stack register is used. The implicit stack register is R15 for nonsegmented operation and RR14 for segmented operation. Furthermore, there are two copies of the implicit stack register, one for system mode operation and one for normal mode. In ordinary operation, each is referred to as R15 or RR14, but when referring to the normal mode stack register while operating in system mode, the LDCTL instruction is used with the argument NSP (in nonsegmented operation) or the arguments NSPSEG and NSPOFF (in segmented operation). It is not possible to refer to the system mode stack register while operating in normal mode.

There are several points about this implicit stack register that are important to understand:

- When the implicit stack register is used as an address register (that is, in a Push or Pop instruction in the Indirect Register mode) or as a base register in the Based or Base Indexed modes, the status lines ST_3-ST_0 reflect stack reference status rather than data reference status.

- An interrupt can occur between the execution of any two Z8000 instructions (or even between repetitions in the block instructions). The system mode implicit stack register is used for saving the CPU status, so it must never contain a higher-numbered address than that of any location containing stack data.

- The normal mode implicit stack register is not involved in the processing of interrupts, but it is used for saving subroutine return addresses in normal mode. Therefore, whenever a subroutine call is made while operating in normal mode, the normal mode implicit stack register must not contain a higher-numbered address than that of any location containing stack data.

Although the significance of these points may not be immediately obvious, they need to be considered when the stack is used other than as a last-in, first-out (LIFO) buffer accessed only with Push and Pop instructions.

One approach to processing stack items in an order other than last-in, first-out is to alter the value of the stack register temporarily. For example, after pushing five words onto the stack, one might wish to increment the stack register by 10 and step through the words in the order received, decrementing the stack register by two before each access. At the end of this process, the stack register returns to its correct value. This works with any other stack (assuming no pushes or pops are done on it during the processing), but with the system mode implicit stack register, any trap or interrupt causes CPU status to overwrite a portion of the five words being processed. This technique can be used with the normal mode implicit stack register provided that no subroutine calls are executed in the course of processing.

One approach to processing stack items that avoids these problems is to move the stack register contents into some other address register and then treat the stack data in question as an array (or other data structure) addressed by the new address register. Additional pushes and pops on the stack (such as those caused by traps, interrupts, or subroutine calls) are then handled correctly without affecting the processing of the stack elements. There are two potential problems with this approach:

- When the contents of the implicit stack register are moved into another address register and the other register is used for referring to the stack items, the status outputs ST_3-ST_0 will show data reference. Thus, this technique cannot be used without modification if the status outputs are used for directing references to separate data and stack memories.

- The programmer must be careful in using addresses that point into the stack. Since the stack storage is allocated dynamically, the same stack memory locations can be used in other ways that change their contents. Naturally, a change to the stack location contents before they are completely processed can only occur as the result of a programming error, but this sort of error is easy to make, especially if a stack management scheme is being used. Furthermore, there is no way to determine by examination of the saved stack address whether the contents are still valid.

A similar technique, subject to the same potential problems, is to use the stack for temporary storage of an array, character string, or other data structure and to pass the address of that structure to a utility subroutine for processing. The called program generally does not use the implicit stack register as an address register for processing the structure.

Since the Z8000 architecture does not allow words to be stored at odd addresses, and since an interrupt can occur at any time, the system mode implicit stack register must never contain an odd address. For this reason, pushes and pops of bytes cannot be allowed on the system mode implicit stack register. This is most easily done by providing for no byte Push and Pop instructions.

Saving byte registers can be accomplished by saving the entire word register. Restoring byte registers without disturbing the other half of the word register must be simulated. For example, if

```
PUSH @RR8,R0
```

is used to simulate PUSHB @RR8,RL0, then POPB RL0,@RR8 can be simulated by

```
LDB RL0,RR8(#1)
INC R9,#2
```

1.5 CONDITION CODES

Condition codes are names for logical combinations of flags bits. There are eight such combinations and an opposite for each, for a total of 16 condition codes. Of the eight, one is "always true"; four are single-bit combinations (C = 0, V = 0, S = 0, Z = 0), and three are multi-bit combinations [S XOR V = 0, Z OR (S XOR V) = 0, C OR Z = 0].

Because the condition codes are designed for use in a variety of Z8000 applications, some of these combinations have more than one name. Following are some typical applications and the condition code names associated with them.

Arithmetic Result Testing. An arithmetic operation (for example, ADD R0,R1) is performed and the result is used for conditional control (for example, a branch).

Code	**Meaning**	**Opposite Code**	
Z	Result is Zero	NZ	(Non-Zero)
MI	Result is negative (MInus)	PL	(Plus)
C	Carry (or borrow) occurred	NC	(No Carry)
OV	OVerflow occurred	NOV	(No OVerflow)

Logical Result Testing. A logical operation (for example, AND R0, R1) is performed and the result is used for conditional control.

Code	Meaning	Opposite Code
Z	Result is Zero	NZ (Non Zero)
PE	Parity is Even (byte op only)	PO (Parity Odd)

Arithmetic Comparison. Two arithmetic values are compared by CP a,b (for example, CP R0,R1). The relationship between the values is to be determined.

Code	Meaning	Opposite Code
EQ	$a = b$ (Equal)	NE (Not Equal)
LT	$a < b$ (Less Than)	GE (Greater or Equal)
LE	$a \geq b$ (Less than or Equal)	GT (Greater Than)

Unsigned Arithmetic Comparison. Two unsigned values (for example, addresses) are compared by CP a,b (for example, CP R0,R1). The relationship between the values is to be determined.

Code	Meaning	Opposite Code
EQ	$a = b$ (Equal)	NE (Not Equal)
ULT	$a < b$ (Unsigned Less Than)	UGE (Unsigned Greater or Equal)
ULE	$a \leq b$ (Unsigned Less than or Equal)	UGT (Unsigned Greater Than)

Miscellaneous Situations. There are many Z8000 instructions (for example, MREQ, shift instructions, block instructions) that set specific flags bits in other ways. Also, the programmer can use the flags bits for passing information between routines. SETFLG and RESFLG are provided for this purpose and any of the 16 combinations can be tested using any of the available names.

Code	Meaning	Opposite Code
LT	S XOR V = 1	GE
LE	(S XOR V) OR Z = 1	GT
ULE	C OR Z = 1	UGT
OV,PE,V*	V=1	NOV,PO,NV*
MI,S*	S=1	PL,NS*
Z,EQ	Z=1	NZ,NE
C,ULT	C=1	NC,UGE

(*V, NV, S, NS not recognized by all assemblers)

It is important to understand the operation of the Test instruction. TEST sets S and Z to reflect the value of its argument; that is, S is set if the high-order bit of the argument is set, and Z is set if the value of the argument is not set zero. The only other bit set is P/V. For byte arguments it is set to reflect the parity, for long-word arguments it is undefined, and for word arguments it is unaffected. C is always unaffected by TEST.

MI and EQ are the only condition codes solely dependent upon Z and S, so there is no easy way to determine whether the tested argument is less than or equal to zero. There are several ways around this:

- CP _a_, #0 can be used instead of TEST a, and C, Z, S, and V will be set according to their arithmetic meanings. This works for byte, word, or long-word arguments.

- For word arguments only, if V is clear, TEST _a_ can be used, and if $\underline{a} \leq 0$, then LE is true.

- TEST _a_ can be followed by two tests:

```
    TEST a
      JR LT,X
      JR EQ,X
    (come here if a > 0)
    .
    .
    .
X:  (come here if a ≤ 0)
```

  This works for byte, word, or long-word arguments.

It is often desirable to postpone the testing of a condition until after the execution of instructions that must be performed regardless of the outcome of the test. For this reason, Z8000 instructions do not change the flags bit settings except to report the outcomes of their operations. In particular, the transfer instructions (CALL, CALR, JP, JR, RET) and the data-moving instructions (CLR, LD, EX, SET, TCC, etc.) do not affect the flags bits. For example, in the code of Figure 1-6, the result of the addition is stored via the pointer RR4, regardless of the FLAGS settings.

If the LD @RR4,R0 instruction affected the flags bits, it could not be placed before the tests. Instead, a LD @RR4,R0 instruction would have to appear at each of the four locations to which control might pass as a result of the testing, and the code would take the form shown in Figure 1-7.

```
    ADD R0,R1
    LD @RR4,R0
      JR OV,W
      JR Z,X
      JR MI,Y
  !(otherwise come here)!
```

Figure 1-6

```
        ADD R0,R1
          JR OV,W
          JR Z,X
          JR MI,Y
        LD @RR4,R0
        .
        .
        .
  W:    LD @RR4,R0
        .
        .
        .
  X:    LD @RR4,R0
        .
        .
        .
  Y:    LD @RR4,R0
        .
        .
        .
```

Figure 1-7

If, however, the example in Figure 1-6 required the unconditional execution of

```
INC R5,#2
```

after the LD instruction (to point RR4 at the next word of storage), the INC instruction could not have been placed before the conditional JR instructions, since INC affects Z, S, and V. (However, POP R0,@RR4 would solve that difficulty.)

To avoid duplicating the increment instruction at each of four locations in the program, the Flags register can be saved and restored as follows:

```
LDCTLB  RH6,FLAGS
INC  R5,#2
LDCTLB  FLAGS,RH6
```

The saving and restoring of the Flags register is not a privileged operation.

One important use of flags bits is based upon the ability to postpone testing: passing information back from subroutines. For example, consider the routine in Figure 1-8.

```
        !Test the ASCII character in RL0 to see whether it is a hex digit.

          CALL TSTHEX; RL0 = the character
          Return with registers unchanged and C=0 if a digit, C=1 if not.
        !
        ASCZER=%30; ASC9=ASCZER+9   !0-9 range!
        ASCA=%41;   ASCF=ASCA+5     !A-F range!

        TSTHEX:  CPB RL0,#ASCZER    !Compare with "zero"!
                   JR ULT,NOTHEX    !All digits are > "zero"!
                 CPB RL0,#ASC9      !In "0" to "9" range?!
                   JR ULE,ISHEX     !Yes--success!
                 CPB RL0,#ASCA      !Now try "A" (> "zero")
                   JR ULT,NOTHEX    !Between "9" and "A"--fail!
                 CPB RL0,#ASCF      !In "A" to "F" range?!
                   JR ULE,ISHEX     !Yes--success!
        NOTHEX:  SETFLG C           !Return C=1!
                 RET
        ISHEX:   RESFLG C           !Return C=0!
                 RET
```

Figure 1-8

This routine might be called in a sequence like

```
        LP:      CALL GETCH         !Get next char into RL0!
                 CALL TSTHEX        !Is it hex?!
                   JR C,X           !C=1 means "no"!
                !(Code for the case:  char is hex)!
                 JR LP
        X:      !(Code for the case:  char not hex)!
                 JR LP
```

There are several advantages in using condition codes this way:

- Registers are undisturbed. The flags bits are usually available, since they cannot be used for long-term storage. If registers are used to pass this kind of information, additional instructions are necessary for saving and restoring previous register values.

- The calling routine can ignore the information if it is irrelevant to the specific case. This is in contrast to the commonly used technique of signaling different conditions by returning to different locations (for example, to the first or second word after the call). This difference is especially important if the return of an error condition is being added to an existing routine. In this case, existing calls do not need to be modified immediately.

- The use of flags bits takes advantage of the Z8000's conditional instructions. Any scheme other than returns to different locations has to be followed by a testing procedure, which would involve the use of flags bits anyway.

The technique of using flags bits to return information from subroutines can be adapted for use with "system call" routines as well, so a sequence such as the following is possible:

```
SC #HXTEST
 JR C,X
```

This sequence cannot be accomplished by using the SETFLG and RESFLG instructions in the system routine. System routines called through the SC mechanism behave like interrupt routines: CPU status (including FLAGS) is saved on the R15 or RR14 stack when the SC is executed, and it is restored from the stack when the IRET is executed. Therefore, the copy of FLAGS saved on the stack must be modified to reflect the desired returned settings. Modification of stack locations by called programs is tricky. For example, when the SC trap first occurs, the saved FCW is the second word on the stack; it can be accessed as R15(#2) or RR14(#2). If the SC handling program then calls the subroutine corresponding to the given index (#HXTEST in the example above), the subroutine return is stored on the stack. Access to the saved FCW is then done as R15(#4) or RR14(#6). If the called subroutine begins by saving registers, the offset changes again. For example, after a

```
PUSHL @15,RR0
```

or a

```
PUSHL @RR14,RR0
```

the new offsets become R15(#8) and RR14(#10). Similarly, each time the processing routine calls a subroutine or uses the stack for temporary storage, the situation changes.

Not only is changing the FCW value saved on the stack potentially error prone, but the type of error that can occur is serious. Thus, change to the saved FCW value is better done by the SC-dispatch routine, the routine whose address appears in the program status area entry corresponding to the SC trap. An SC-dispatch routine to accomplish this is shown in Figure 1-9.

Many variations on this dispatch mechanism are possible, depending on the system in which it functions. This example illustrates the use of condition codes, but is not a model SC dispatcher.

1.6 POSITION-INDEPENDENT PROGRAMS

A position-independent program is one that can be moved to different locations in memory without changing its behavior. The instructions and program constants are in a fixed order, but their behavior does not depend upon the actual addresses of the memory locations where they are stored.

An example of a position-independent program is the subroutine TSTHEX of Figure 1-8. Figure 1-10 contains an assembled version of this subroutine starting at location 1000H.

```
SCDISP:  EX R13,@RR14            !Save RR12, get "reason" into R13!
         PUSH @RR14,R12
         PUSH @RR14,R0           !Use RL0 to pass saved FLAGS!
         LDB RL0,RR14(#7)        !(Offset of FLAGS is 7 after
                                                  above saves)!
         !(Code to compute processing subroutine
           address from "reason" and leave it in RR12)!
         CALL @RR12              !Call processing routine!
           JR C,NOMSG            !Dont use updated FLAGS!
         LDB RR14(#7),RL0        !Update flags on stack!
NOMSG:   POP R0,@RR14            !Restore R0!
         POP R12,@RR14           !Restore RR12!
         LD R13,@RR14
         IRET
```

Figure 1-9

```
1000    0A08 3030     TSTHEX:  CPB RL0,#ASCZER
1004    E709                    JR ULT,NOTHEX
1006    0A08 3939              CPB RL0,#ASC9
100A    E308                    JR ULE,ISHEX
100C    0A08 4141              CPB RL0,#ASCA
1010    E703                    JR ULT, NOTHEX
1012    0A08 4646              CPB RL0,#ASCF
1016    E302                    JR ULE,ISHEX
1018    8D81          NOTHEX:  SETFLG C
101A    9E08                   RET
101C    8D83          ISHEX:   RESFLG C
101E    9E08                   RET
```

Figure 1-10

Because of relative addressing, the hex values of the instructions remain the same wherever the program is assembled. This is true despite the fact that the symbols NOTHEX (at location 1018) and ISHEX (at location 101C) are referred to by instructions in the program. To understand this, consider the two instances of the instruction

```
JR ULT,NOTHEX
```

The hex values corresponding to these two instances are not the same, because NOTHEX is used in these two instructions simply as a convenience to the programmer. They are actually two different instructions:

```
JR ULT,$+%14

JR ULT,$+%8
```

In other words, these instructions do not rely on the fact that NOTHEX is at 1018H. Instead they require the destination to be 14 or 8 locations after the location containing the instruction.

Position-independent programs contribute in several ways to achieving modularity. One way is by using "silicon software." Imagine a set of programs, each available on a ROM, that provide a variety of software tools, such as a debugger, an editor, a text-formatting program. If each of these programs is position-independent, the system designer can select from among these ROMs and assign a set of memory addresses to each, thus building a custom-tailored system. A variation of this idea is a "demand loading" memory system that loads position-independent programs from secondary storage into any available RAM area whenever calls are made on them.

As another example, consider a debugging program that can be loaded into RAM wherever space is available. For example, it could reside in a buffer area while the initialization code was executing and then move to overlay the initialization code while the program used the buffers.

These examples show some of the uses of position-independent programs. When writing position-independent programs, the main rule is, "Don't use addresses in instructions." Addresses in instructions are generally used in the direct and indexed addressing modes and as immediate arguments. Direct and indexed addressing cannot be used in position-independent programs except when indexed addressing is used as previously described to simulate based addressing. The use of addresses as immediate arguments should be avoided. The same result can be achieved with the LDA and LDAR instructions.

Relative addressing--the CALR, JR, LDR, and LDAR instructions--is the principal tool available to the programmer writing position-independent programs. Another important tool is the use of fixed-location utilities called from position-independent programs. For example, in a demand-loading scheme, segment zero might be dedicated to routines that are always resident. If so, the first 256 bytes of segment zero can consist of subroutine entry points, and calls can be made on these subroutines by using direct or indexed addressing from position-independent programs. (The first 256 bytes of each segment can be addressed by using a short segmented address.) The system call trap can also be used to access system routines from position-independent programs.

Many variations on these ideas are possible, depending on what is to be fixed and what is to be position independent. Use of the stack for temporary storage automatically achieves position independence of the data. If the stack is not used, position independence of data can be achieved using the LDAR instruction, the Indirect Register, or the Based and Base-Indexed Addressing modes.

The kind of position independence discussed here is an independence from the particular range of addresses assigned to the program. Another kind of position independence is provided by an external memory-mapping facility, which allows a given address range to correspond to different physical memory locations.

1.7 SUBROUTINES

The principal property of Z8000 subroutines is that they use RET as an exit so that they can be called from more than one place. Invocation of subroutines is accomplished with the CALL (or CALR) instruction. CALL and RET perform complementary functions. When a CALL (or CALR) instruction is executed, the address of the following memory location is saved on the RR14 or R15 stack. Then transfer is made to the address specified in the CALL instruction. When a RET instruction is executed, the address on top of the RR14 or R15 stack is popped into the PC; that is, it is removed from the stack and a transfer to that address is made.

In this way, the programmer can encode commonly used functions in one place and then make use of them by CALLs whenever they are needed. The CALL of the given subroutine is like another instruction added to the CPU's instruction set. This is the most important tool of the assembly language programmer, because it allows instructions to be used that are relevant to the application at hand, thereby simplifying and clarifying assembly language programs.

The CALL and RET instructions provide the subroutine calling mechanism but do not dictate a specific means of argument passing. For example, if a subroutine is needed to compute the square root of a number, the programmer must decide how to specify that number to the subroutine. The programmer must also decide how the subroutine will report the answer.

There are three commonly used methods for argument passing:

- In a register
- On a stack
- In the program, in locations following the call

Each of these methods can be used to pass actual arguments or to pass the address of an argument table.

The return of answers to the calling program has four commonly used options:

- In a register
- On a stack
- By returning to addresses at varying offsets from the CALL
- By manipulating flags bits

The use of registers for subroutine argument passing and result returning is the most popular and most efficient option. For example, to implement the FORTRAN statement Y=SQRT(X) the following code can be used:

```
                LDL RR0,X               !Get X!
                CALL SQRT               !Compute square root!
                LDL Y,RR0               !Store in Y!
```

Here the subroutine SQRT takes its argument in RR0 and returns the answer in RR0.

The code for a SQRT routine that takes arguments and returns results on a stack might be:

```
                PUSHL @RR6,X
                CALL SQRT
                POPL Y,@RR6
```

This assumes that a stack controlled by RR6 is available for use in argument passing.

There are times when passing arguments on a stack is preferable to using registers. There might be more arguments than can be accommodated in the registers, or it might be desirable to make the subroutine re-entrant (see Section 1.8). When a stack is used for passing arguments, the subroutine usually uses the Based or Base-Indexed Addressing modes to refer to them. For example, suppose that the subroutine BIGSQRT accepts an array of 14 numbers on the RR6 stack and replaces each with its square root. The code might look like that of Figure 1-11.

```
        BIGSQRT:  LDK R2,#14            !Set argument Counter!
                  CLR R3                !Initialize index!
        LOOP:     LDL RR0,RR6(R3)       !Get next arg!
                  CALL SQRT             !Compute square root!
                  LDL RR6(R3),RR0       !Store it back!
                  INC R3,#4             !Arguments are 4 bytes!
                  DJNZ R2,LOOP          !Loop if more arguments!
                  RET
```

Figure 1-11

In nonsegmented operation or in segmented operation when the stack segment number is known at assembly time, indexed addressing can also be used to refer to stack items. The passing of arguments by including them in the program following the CALL, and the return of status information by returning to addresses at varying offsets from the CALL are illustrated in the following code:

```
        CALL SQRT                   !Compute square root!
        X                           !Adr of argument!
        Y                           !Adr at which to store result!
        JR NEGX                     !Error return: X was negative!
        (Execution resumes here if no error)
```

The subroutine SQRT used with this sort of call might look like the one in Figure 1-12.

```
SQRT:    LDL RR12,@RR14          !Get saved return!
         LDL RR2,@RR12           !Get address of X!
         LDL RR0,@RR2            !Then get X itself!
         INC R13,#4              !Step over adr of X!
         LDL RR2,@RR12           !Get address of Y!
         INC R13,#4              !Step over adr of Y!
         TESTL RR0               !Test X!
           JR MI,ERREX           !Error if X ≤ 0!
         INC R13,#2              !Step over error exit!
        !(Compute square root)!
         LDL @RR2,RR0            !Store in Y!
ERREX:   LDL @RR14,RR12          !Put updated Return adr on stack!
         RET
```

Figure 1-12

The code makes it apparent that this means of passing information is awkward. It was originally developed for computers that had few registers and no multiple-word instructions and that stored their return addresses in the subroutines rather than on a stack. It is not well suited to the Z8000.

Often it is convenient to use an argument table whose address is passed to the subroutine. The subroutine refers to the table elements as it would to arguments on a stack--it uses based or base-indexed addressing. An example of such a table is given in Section 2.4.

The flags bits provide a convenient means of passing error or status information back from a subroutine. Since RET does not affect any flags bits, a condition can be set in a subroutine and tested in the calling program. For example, the SQRT routine might use C to indicate that an error condition prevented it from computing a square root. The calling program might look like this:

```
         LDL RR0,X               !Get the argument!
         CALL SQRT               !Compute the square root!
           JR C,ERREX            !C set if error!
         LDL Y,RR0               !Store the result!
```

1.8 RE-ENTRANT PROGRAMS

Often in computer systems, two or more distinct processes seem to be running simultaneously. Actually, the computer alternates between these processes, dropping each one in turn, then picking it up at the point at which it was dropped. Since the CPU's most fundamental resources are generally not duplicated, the two processes share them. For example, the values of the FCW and the PC being used for one process must be saved before they are set to the values appropriate for the next process. There are other resources that may need to be saved, such as the general-purpose registers and memory. The context of the processes is the total set of registers and memory that needs to be saved for each process when it is suspended and later restored. The operation of saving one context and restoring another is called context switching.

A re-entrant program is a program that can be used simultaneously by two or more processes. A program is re-entrant if and only if it refers only to registers and memory locations that are included in the process contexts.

One example of concurrent processes arises when interrupts are used. In this case, the CPU provides for the automatic saving of the PC and FCW. Let us assume that we are working with a system in which every interrupt-processing routine saves and restores RR0 and RR2. Figure 1-13 shows three pieces of code that form the basis of an extended illustration of how re-entrancy is achieved.

The routine MULTEN is re-entrant, since it refers only to registers and memory locations in the context assumed above. The references to RR14(#4) are to a location in the context. This is because the contents of RR14 (or R15 in the nonsegmented case) are implicitly saved and restored in switching to and from interrupt processing, and all memory locations at a positive offset from the base defined by RR14 are in effect separate copies of that portion of the context.

The example shows how the execution of MULTEN, called from the code starting at 100, is interrupted to allow the interrupt-processing routine IROUT to run. In turn, IROUT calls MULTEN, so MULTEN must work in two contexts simultaneously.

This example follows the changing contents of the registers R0, R1, R2, R3, RR14, PC, and FCW, and shows the section of the stack used during execution of this portion of the program. Figure 1-14 lists the assumed initial values.

As the first instruction, at 100 of segment 6, is executed, the stack register value changes to (0400,0098) and stack location 98 contains 0003, the contents of R3. The PC is incremented to (0600,0102), and everything else is unchanged. The next instruction is the call to MULTEN. Figure 1-15 shows the status following that call.

Calling Program (in segment 6)

```
    100 93E3                          PUSH @RR14,R3     !Put argument on stack!
    102 5F00 0600 2000                CALL MULTEN       !Multiply it by 10!
    108 97E3                          POP R3,@RR14      !Return argument to R3!
    10A                               -------------
```

MULTEN Program (in segment 6)

```
   2000 31E1 0004         MULTEN:    LD R1,RR14(#4)    !Get argument!
   2004 BD2A                         LDK R2,#10        !Constant Multiplier!
   2006 9920                         MULT RR0,R2       !10 x argument!
   2008 33E1 0004                    LD RR14(#4),R1    !Replace arg with result!
   200C 9E08                         RET
```

Interrupt-Processing Program (in segment 8)

```
    600 91E0              IROUT:     PUSHL @RR14,RR0   !Save!
    602 91E2                         PUSHL @RR14,RR2   !  registers!
    604 31E0 0008                    LD R0,RR14(#8)    !Get "reason"!
    608 93E0                         PUSH @RR14,R0     !Compute!
    60A 5F00 0600 2000               CALL MULTEN       !  10 x "reason"!
    610 97E0                         POP R0,@RR14      !R0 gets 10 x "reason"!
    ------------------              !(Perform other tasks)!
    630 95E2                         POPL RR2,@RR14    !Restore!
    632 95E0                         POPL RR0,@RR14    !  registers!
    634 7B00                         IRET
```

Figure 1-13
Re-entrant MULTEN Routine, a Calling Program, and an Interrupt-Processing Program

Registers Name	Contents	Stack Address	Contents
R0	0000	(none used yet)	
R1	1111		
R2	2222		
R3	0003		
RR14	(0400,009A)		
PC	(0600,0100)		
FCW	D880		

Figure 1-14
Initial Values for MULTEN Routine

Registers		Stack		
Name	Contents	Address	Contents	
R0	0000	98	0003	argument
R1	1111	96	0108	saved PC
R2	2222	94	0600	
R3	0003			
RR14	(0400,0094)			
PC	(0600,2000)			
FCW	D880			

Figure 1-15
Values After Call to MULTEN from 102

Figure 1-16 shows the situation after the first two instructions of Multen have been executed. Suppose at this point that an interrupt occurs and that IROUT is the processing routine. Figure 1-17 shows the status immediately following the interrupt. The first two instructions of IROUT push RR0 and RR2 onto the stack. Then the "reason" is fetched and pushed onto the stack as an argument for the call to MULTEN. MULTEN is called, and after the first two instructions of MULTEN have been executed, we are exactly where we were before the interrupt. Figure 1-18 shows the new status.

Register		Stack	
Name	Contents	Address	Contents
R0	0000	98	0003
R1	0003	96	0108
R2	000A (10 = %A)	94	0600
R3	0003		
RR14	(0400,0094)		
PC	(0600,2006)		
FCW	D880		

Figure 1-16
Status Before the Interrupt

The stack locations 7E, 80, and 82 in Figure 1-18 play the same role as did 94, 96, and 98 in Figure 1-16. If the contents of stack locations 84 thru 98 in Figure 1-18 are covered up, there would be no essential difference between the two figures. The only record of the first execution of MULTEN is stored in these stack locations. Conversely, in Figure 1-16, if the portion of the stack with addresses 100 through 114 were shown (nothing tells us where the stack originally started), the context of a previous execution of MULTEN might be found.

Registers		Stack		
Name	Contents	Address	Contents	
R0	0000	98	0003	
R1	0003	96	0108	
R2	000A	94	0600	
R3	0003	92	2006	saved PC
RR14	(0400,008C)	90	0600	
PC	(0800,0600)	8E	D880	saved FCW
FCW	D800*	8C	0005	"reason"*

*To make the example concrete, assume a value of 0005 for "reason" and an FCW value of D800 associated with the interrupt.

Figure 1-17
Status Immediately Following the Interrupt

Assume that execution proceeds without further interrupts. MULTEN computes 5 x A and stores the result at stack location 82 [at RR14(#4)]. Its RET causes the contents of 7E and 80 to be popped into the PC and execution resumes in IROUT, where the 0032 (5 x A) is popped into R0 and presumably is used in the "perform other tasks" section of IROUT.

Register		Stack		
Name	Contents	Address	Contents	
R0	0000	98	0003	Argument & return address for first
R1	0005	96	0108	(interrupted) execution of MULTEN
R2	000A	94	0600	
R3	0003	92	2006	CPU status & "reason" pushed auto-
RR14	(0400,007E)	90	0600	matically when the interrupt
PC	(0600,2006)	8E	D880	occurred
FCW	D800	8C	0005	
		8A	0003	RR0,RR2 values saved by IROUT (contain
		88	0000	register values set during first exe-
		86	0003	cution of MULTEN)
		84	000A	
		82	0005	Argument & return address for second
		80	0610	execution of MULTEN
		7E	0800	

Execution of MULTEN is at the exact point reached before the interrupt (Figure 1-16). Every value in Figure 1-16 is somewhere on the stack in this figure.

Figure 1-18
Current and Saved Contexts for MULTEN

When execution in IROUT reaches 630, the RR2 and RR0 values are restored from the stack. At this point, status is exactly as shown in Figure 1-17, except that the PC (and possibly the FCW) has a different value. The execution of the IRET restores the saved values of PC and FCW, leaving the status originally shown in Figure 1-16.

Execution of MULTEN proceeds at 2006 as if there had never been an interrupt. The result of 3 x A (1E) is stored in stack location 98 [R14(#4)]. The RET at 200C causes the saved PC to be restored from stack locations 94,96. Execution of the original program then resumes at 108 of segment 6, where the result of the multiplication is popped into R3. The status at this point is shown in Figure 1-19. All of the values here are exactly as they would have been if the execution of MULTEN had not been interrupted.

Registers			**Stack**	
Name	**Contents**		**Address**	**Contents**
R0	0000		(none still in use)	
R1	001E	Result of MULT RR0,R2		
R2	000A	Result of LDK R2,#10		
R3	001E	Result of POP R3,@RR14		
RR14	(0004,009A)			
PC	(0600,010A)			
FCW	D800	FLAGS set by MULT RR0,R2		

Figure 1-19
Final Values for MULTEN Routine

This example also illustrates how the definition of re-entrancy depends upon the properties of the surrounding system. If RR4 and RR6 instead of RR0 and RR2 had been preserved by interrupt-processing routines, then MULTEN would not be re-entrant and it could not be called from interrupt-processing routines.

The MULTEN example illustrates context switching triggered by interrupts. Another instance of re-entry, for which it is harder to provide a simple illustration, is a program shared by a number of concurrent processes, each doing approximately the same thing. For example, a BASIC or PASCAL time-sharing system might have one copy of the interpreter that works on the users' programs "concurrently," switching from one to the next either at the expiration of a "time slice" or when the user's program pauses for I/O. Each user would have an interpretable program and a temporary storage stack. These would be in the user's private memory and would be addressed using a base register and an offset (pseudo-PC) register for the interpretable program and a stack register for the stack. These registers and the other general-purpose registers used by the interpreter constitute the context to be switched. The re-entry of the interpreter depends upon its reference to the users' memory areas only through the use of the registers making up the context.

1.9 CONTEXT SWITCHING

In Section 1.7, we defined the context of a process to be the values of all registers and memory locations that need to be saved before another process running "at the same time" can have its turn at using them. In general, the context of a process consists of the entire register and memory contents, but in most applications measures are taken to keep the size of the context to a minimum. Fixed storage locations can be avoided, and the times at which context switches occur can be controlled.

Fixed storage locations must become part of the context of a process if some other process can change the contents between the time its value is set and the time it is no longer needed. On the other hand, a process that "ties up the loose ends" before another process can run can have a small context, even though it may use and abandon many registers and locations during the period in which other processes cannot run. The recursive subroutine QUICK presented in Section 2.6 is an example of this phenomenon.

In most context-switching schemes, the stack is used for storage of all or part of saved process contexts, as illustrated in Section 1.8. Saving registers on the stack is accomplished efficiently by using the LDM instruction. For example,

```
        DEC R15,#16             !Can't decrement by 28!
        DEC R15,#12             ! all at once!
        LDM @RR14,R0,#14
```

causes registers R0 through R13 to be saved on the RR14 stack.

Saving control registers, if necessary, is accomplished by loading them into registers and then saving the registers. If it is necessary to save the FCW explicitly, care must be taken that the saving operations do not affect the flags bits before they are saved or after they are restored. For example, the DEC instruction affects V, Z, and S, so after the above instructions have been used to save the registers, it is too late to save FLAGS. A variation on the preceding code that saves FLAGS is:

```
        PUSHL @RR14,RR12        !Make room to work!
        LDCTL R12,FCW           !Get FCW into R12!
        SUB R15,#24             !Finish saving registers!
        LDM @RR14,R0,#12
        PUSH @RR14,R12          !Save FCW!
```

Of the control registers, the Normal Stack Pointer is the one most likely to be part of a process context in a multi-processing system. To save it, the following instructions are added to the above:

```
                        LDCTL R12,NSPSEG
                        LDCTL R13,NSPOFF
                        PUSHL @RR14,RR12
```

If fixed locations are part of the process context, their contents also must be saved. In the code shown in Figure 1-20, assume that RR2 contains the address of a list of fixed word locations whose contents must be saved. Assume that the list is terminated by a double word, -1. This code causes the contents of these locations to be saved on the stack, each accompanied by the corresponding address.

```
LOOP:   LDL RR4,@RR2            !Get next item!
        CPL RR4,#-1             !Test for terminator!
          JR EQ,DONE            !Done if -1 encountered!
        LD R0,@RR4              !Get contents!
        PUSH @RR14,R0           !Save both!
        PUSHL @RR14,RR4
        INC R3,#4               !Increment list pointer!
        JR LOOP
DONE:   -----
```

Figure 1-20

1.10 INTERRUPTS

An interrupt forces a context switch. Since there is almost no control of the time a switch to the interrupt context occurs, interrupt routines must save and restore the values of any registers, control registers, or memory locations they use. (An exception is a memory location purposely changed by the interrupt routine, such as a flag indicating that output of a given line of text is finished.)

Before interrupts can be used, the linkage between the interrupt and the processing routine must be established. This is done using the program status area (PSA) and the program status area pointer (PSAP). The format of the program status area is described in the Z8000 CPU Manual (document 00-2010-C). In the PSA, a CPU status (FCW and PC) is specified for every allowed interrupt type. In contrast with machines that used fixed memory locations for such interrupt response definition, the PSA of the Z8000 can be anywhere in Program memory so long as it is on a 256-byte block boundary (that is, the last eight bits of its address are zero). This means that the PSA can be assembled with the program without conflicting with the loader's use of the interrupt facility. The only thing remaining to be done at initialization time is to set up the PSAP and then to enable interrupts. If the PSA begins at PSALOC, setting up the PSAP can be done by:

```
LDA RR0,PSALOC
LDCTL PSAPSEG,R0
LDCTL PSAPOFF,R1
```

The use of interrupts for input or output of data requires communication between the program requesting the input or output and the associated interrupt-processing routines. Furthermore, an interrupt-processing routine must communicate with itself; that is, whenever an interrupt occurs, it must know exactly what it is doing and how far along it is.

The solution to providing communication between interrupt and application routines and to provide temporary storage for the interrupt routines is a set of fixed memory locations (often called a process status block or context block) containing pointers, counters, flags, etc.

When the application routine needs to perform an I/O operation, it calls on an initiator routine. For example, it may need to send the ASCII characters for "HELLO" to a CRT screen. The initiator program sets a pointer in the context block to the zero-terminated string of ASCII characters for "HELLO" provided by the application program, and it sets a flag in the context block to "BUSY." Then it does whatever is necessary to assure that output interrupts for the CRT screen begin to occur. As each interrupt occurs, the processing routine transmits another character of the string and advances the pointer in the context block. When the pointer reaches the terminating zero, the interrupt routine sets the flag in the context block to "DONE." Meanwhile, the application program can be doing other things. If it needs to output another string, it waits for the flag to change from "BUSY" to "DONE." It can enter a loop in which all it does is test the flag, or it can do other things while the output proceeds.

This sort of communication between tasks proceeding under interrupt and application programs is sometimes used to implement an event-driven timesharing system. Instead of entering a loop to wait for the flag to change from "BUSY" to "DONE," the program defers to other tasks, allowing them to execute until they too are held up waiting for an I/O operation to be completed.

1.11 INITIALIZATION

For the programmer responsible for the entire CPU instead of simply providing programs to run under some system, the sequence of operations following a cold start (reset) is important.

Execution begins when the CPU fetches its CPU status (FCW and PC) from inst uction memory addresses beginning at segment 0, offset 2. The FCW is at ffset 2, the PC at offset 4. If an external memory-mapping unit is in use, it must be capable of dealing properly with these initial fetches, even before any code is executed to establish memory-mapping parameters.

The PC value at location 4 is the address of the first instruction to be executed. The FCW value should leave the CPU in system mode and segmented operation (unless the CPU is a Z8002) with all maskable interrupts disabled. The nonmaskable interrupt (NMI) should be disabled at this point also, but that is impossible, so the system must be designed so that the NMI cannot occur immediately after a reset.

The initialization code first sets the PSAP to point at the previously assembled PSA. The implicit stack register (R15 or RR14) must then be set. If an external memory mapping facility is used, its parameters are set up as soon as possible. Until then, it must continue to handle all instruction, data and stack references properly. Once the stack register and the PSAP are properly initialized, interrupts can be enabled. If the Refresh register is to be used, it is initialized during this sequence.

1.12 PROGRAMMING FOR BOTH SEGMENTATION MODES

It is important for Z8000 programmers to know how to write programs for operation in one segmentation mode that can be adapted for use in the other segmentation mode with minimal alterations. The only way two modes differ is in the format of addresses--in instructions, in general-purpose registers, in the PC, in control registers, and on the stack after subroutine calls, traps, or interrupts. Therefore, the solution to this lies in finding mode-independent ways of handling addresses. Addresses are manipulated by programs in many ways. The most common are:

- Loading them into registers.
- Performing arithmetic on them.
- Using them in the Indirect Register, Based and Base-Indexed Addressing modes.
- Moving them out of registers and into memory or onto the stack.

The two program fragments shown in Figure 1-21 are segmented and nonsegmented versions of the same algorithm. If symbolic definitions are given for the address registers, the code takes the form shown in Figure 1-22.

```
Non-Segmented          Segmented

LDA R2,XYZ             LDA RR2,XYZ
LD R0,@R2              LD R0,@RR2
INC R2,#2              INC R3,#2
LD R1,@R2              LD R1,@RR2
PUSH @R15,R2           PUSHL @RR14,RR2
LD R4,R2               LDL RR4,RR2
```

Figure 1-21

```
ADREG = R2; ADOFF = R2             ADREG = RR2; ADOFF = R3
SAVREG = R4                        SAVREG = RR4
SR = R15                           SR = RR14

LDA ADREG,XYZ                      LDA ADREG,XYZ
LD R0,@ADREG                       LD R0,@ADREG
INC ADOFF,#2                       INC ADOFF, #2
LD R1,@ADREG                       LD R1,@ADREG
PUSH @SR,ADREG                     PUSHL @SR,ADREG
LD SAVREG,ADREG                    LDL SAVREG,ADREG
```

Figure 1-22

With the symbolic definitions, the two pieces of code are very similar. The remaining problem is the "L" in the mnemonics. If there were an assembler that recognized the perfectly unambiguous source statements

```
LD RR4,RR2
PUSH @RR14,RR2
```

and generated the long-word versions of the instructions, then at the source code level the segmented and nonsegmented programs would be identical. Without such an assembler, the only other possibility is conditional assembly. Except for very small programs, this is unlikely to be workable, unless the conditional instructions are built into a set of address-manipulation macros.

For example (following no particular macro syntax), an address pushing macro could be defined as follows:

```
APUSH x,y =
      if y is an RR, then
                "PUSHL @x,y"
      else
                "PUSH @x,y"
```

Then,

```
APUSH SR,ADREG
```

is the next-to-last line of either of the programs in Figure 1-22.

PART II

PROGRAMMING EXAMPLES

Part I showed how specific features of the Z8000 are related to standard programming techniques. This section presents some complete examples to give a clearer picture of how Z8000 instructions and features are used.

2.1 ADDING AN ARRAY OF NUMBERS

Problem: To find the sum of an array of 16-bit signed numbers.

Solution: The items are added one at a time to an initially cleared accumulator. Any occurrence of a V indication following any of the additions is registered, and V is set upon completion if overflow occurs at any point during the operation.

Notice that the sum can be correct even if overflow occurs; for example, let the array be (32,765, 8, -25). The first sum, 32,765 + 8, yields -32,763 and an overflow indication. The second sum, (-32,763) + (-25) yields 32,748 and another overflow indication. The final answer is correct:

$$32,748 = 32,765 + 8 + (-25).$$

An overflow indication is set upon completion of the addition, and the programmer can choose to take action. Alternatively, the addition program might take action on overflow (such as by terminating the process), but the programmer calling the function has more information about the intended use of the sum and the nature of the data. The code for this appears in Figure 2-1.

```
!Addition subroutine
   CALL SUM with RR2 = array address
                 R0  = number of words in the array (0 to 32,767)
   Returns sum in R1; V is set on return if an arithmetic overflow
   occurred in any of the addition operations used in forming the
   sum.
   The contents of R0, R2, R3 and R4 are lost.
!
SUM:    CLR R1           !Initialize sum to zero!
        CLR R4           !R4 saves any V's!
LOOP:   CP R0,#0         !Done when R0 no longer!
          JR LE,ENDLP    ! greater than zero!
        ADD R1,@RR2      !Add in the next!
        TCC OV,R4        !Save overflow indication!
        INC R3,#2        !Increment array pointer!
        DEC R0           !Decrement loop counter!
        JR LOOP
ENDLP:  RESFLG V
        TEST R4          !V=0 if no overflow!
          RET Z
        SETFLG V         !Otherwise V=1!
        RET
```

Figure 2-1

Notes:

1. Notice that the test for the loop termination condition is done first; this allows the program to behave properly if the initial value of R0 is zero--it returns a sum of zero. Also notice that the test is for LE instead of for EQ. This is simply a precaution. If the count becomes negative, then there is a programming error somewhere, and it is best to stop immediately.

   Given that we wish to test for a counter value less than or equal to zero, we use

   ```
   CP R0,#0
   ```

   instead of

   ```
   TEST R0
   ```

   Because the TEST instruction leaves the V bit unaffected, while the definition of LE is Z OR (S XOR V).

   An alternative to

   ```
   CP R0,#0
   ```

   is the sequence

```
RESFLG V; TEST R0
```

but this sequence does not work with the TESTB instruction, because TESTB uses V to report the parity of the byte (since P = V), and this is unrelated to the sign.

2. Notice the use of the TCC instruction. Initially R4 is cleared; if V is clear (no overflow occurred in the ADD), then

   ```
   TCC OV,R4
   ```

   leaves R4 unaffected. If V is set (overflow did occur), then

   ```
   TCC OV,R4
   ```

   causes the low-order bit to be set. This means that if overflow ever occurs, R4 will be non-zero for the remainder of the time, since the only instruction affecting R4 after it is initially cleared is the TCC, which either sets it or leaves it unaffected.

   It is important to note that the TCC instruction does not set the destination value to zero if the specified condition code is false.

3. Notice the use of

   ```
   INC R3,#2
   ```

   to increment the array pointer RR2. This is done because the segmented address arithmetic is done separately on the segment and offset portions of the address.

   As written, the program wraps around the end of a segment, treating the word at offset zero as the successor to the word at offset 65,534 (FFFE). If this is to be treated as an error, a test can be made for this condition. Z can be used, but this does not necessarily work with larger increments; if long words are being added, for example, INC R3,#4 might change R3 from FFEE to 2. Another approach is to test for this condition on entry, using the initial values of RR2 and R0.

2.2 DETERMINING THE PARITY OF A BYTE STRING

Problem: To find the parity of an arbitrarily long byte string and to set P (PE true) if the total number of bits in the string of bytes is even, to clear P (PO true) if the total number of bits is odd.

Solution: The parity of a byte string is, by definition, the sum of its bits modulo 2. Since addition is associative (i.e., the sum is the same if the items are grouped, subtotals computed for the groups, and the subtotals added), the parity of the byte string is the sum of the parities of its bytes.

Furthermore, if a and b are binary numbers (in particular, if they are bytes), then the parity of (a XOR b) equals the parity of a plus the parity of b. (It suffices to prove this for one-bit arguments a and b, since the parity of an n-bit binary number is the sum of the parities of its n bits. The proof for one-bit arguments is accomplished by considering the four possible bit combinations.) Therefore, the total parity can be determined as follows:

- Initialize a register to zero.
- For each byte of the string, compute the XOR of the byte with the current contents of the register.
- Test the parity of the final contents of the register.

The code for this appears in Figure 2-2.

```
!Subroutine to test the parity of an arbitrarily long
 byte string
        CALL BIGPAR with RR2 = address of the byte string
                         R1 = number of bytes (0 to 32,767)
        Returns with P set (PE true) if parity is even, P
        clear (PO true) if parity is odd.  Contents of R0,
        R1, R2 are lost.
!
BIGPAR: CLRB RL0            !Accumulate parity in RL0!
LOOP:   CP R1,#0            !All tested yet?!
          JR LE,ENDLP       !If so, determine final parity!
        XORB RL0,@RR2       !XOR this byte with RL0!
        INC R3              ! and set up!
        DEC R1              !  for next byte!
        JR LOOP
ENDLP:  TESTB RL0           !Final parity!
        RET
```

Figure 2-2

Notes:

1. Notice that by initially clearing RL0, we assure that a zero-length string has even parity.

2. If we wish to allow for from 0 to 65,535 bytes in the string, we replace

   JR LE,ENDLP

   with

   JR EQ,ENDLP

In this case we are using the contents of R1 as an unsigned number in the range 0 to $2^{16}-1$ instead of as a signed number in the range -2^{15} to $2^{15}-1$.

3. If we wish to allow for from 1 to 65,536 bytes in the string, we remove the instructions

```
CP R1,#0
 JR LE,ENDLP
```

   and move the label LOOP down to the XORB instruction. The instructions

```
DEC R1
JR LOOP
```

   become

```
DJNZ R1,LOOP
```

   and the label ENDLP is no longer be needed.

4. For long byte strings, the efficiency of this routine can be increased by using the XOR instruction to process whole words at a time. Special tests have to be included to handle strings that begin at an odd byte or end at an even byte.

2.3 ACCESSING AN ARRAY LARGER THAN 65,536 BYTES

Problem: To manage a one-dimensional array that is too large to fit within one memory segment and has too many elements to be indexed by a 16-bit word.

Solution: Two solutions to this problem are presented. One provides high efficiency but little flexibility, and the other provides great flexibility, but at a substantial cost in processing overhead.

The high-efficiency scheme uses an arbitrary segmented address as the address of the first array element and assumes that the array is stored contiguously in memory. Segmented address (N+1, 0) is assumed to follow address (N,65,535); that is, consecutively numbered segments are treated as contiguous pieces of the address space. If the segment number bits were bits 6 through 0 of the high-order segmented address byte, this interpretation would be achieved automatically simply by treating segmented addresses as 32-bit unsigned integers. Since this is not the case, the addition of an offset to the starting address of the array must include an operation that takes bits 6-0 of the high-order word of the result and adds them to the segment number field, which is in bits 14-8.

A subroutine is provided to take the base segmented address in one long-word register and an offset in another long-word register. The offset must be less than 2^{23}. The algorithm used causes a wraparound from segment 127 to segment 0, so the full 2^{23} bytes of segmented address space are used, regardless of the base segmented address. The code for this version appears in Figure 2-3.

```
!Address-mapping subroutine (high-efficiency version)
   CALL ADMAP with RR2 = virtual address (23 bits)
                   RR4 = starting segmented address
   Returns with RR2 = segmented address corresponding
 to the given virtual address and RR4 preserved.
!
ADMAP:  ADD   R3,R5
        ADCB  RL2,RH4
        EXB   RH2,RL2
        RET
```

NOTE: The EXB instruction is unnecessary if the result is returned in RR4. The code for this is:

```
ADMAP:  ADD R5,R3
        ADCB RH4,RL2
        RET
```

The longer version given above allows the array base to be maintained in RR4 at all times.

Figure 2-3

The high-flexibility scheme also uses a 23-bit offset, or virtual address, but instead of a starting segmented address and a contiguous array, it uses a virtual-to-segmented address mapping scheme that works as follows (see Figure 2-4):

- An array of "virtual" addresses in ascending numerical order (V_1, V_2, ...,V_n is provided. A segmented address (S_0, S_1,..., S_{n-1}) is associated with each. Virtual addresses 0 through V_1-1 are mapped into a contiguous block of segmented addresses, starting at S_0. V_1 through V_2-1 are mapped into a block at S_1, and so on.

- The given virtual address, v, is compared with each of V_1, V_2,..., V_n until the first V_i is found for which $v < V_i$. If $v \geq V_n$, an error indication is returned.

- The segmented address $S_{i-1} + (v-V_{i-1})$ is returned (V_0 is assumed to be zero).

The address calculation $S_{i-1} + (v-V_{i-1})$ is performed as for the high-efficiency scheme above, so that consecutively numbered segments are treated as contiguous, and wraparound occurs from segment 127 to segment 0.

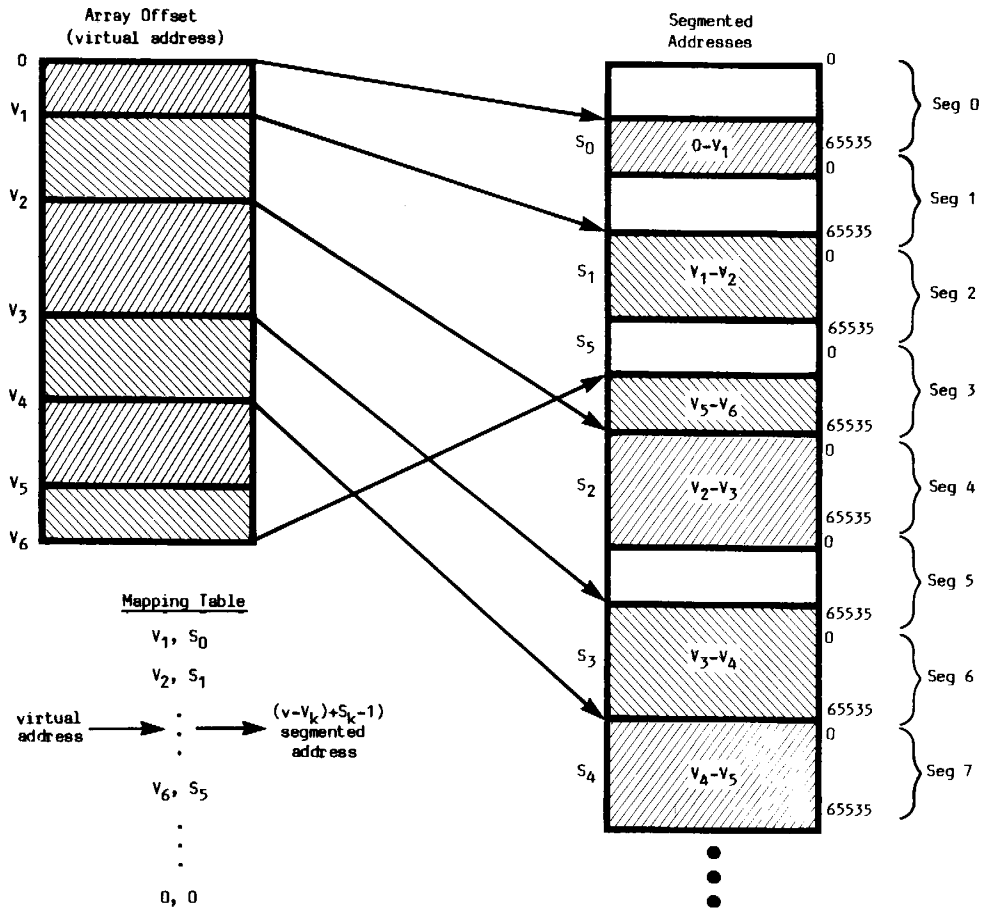

Figure 2-4

As an example, suppose we have an array of 200,000 bytes that we wish to store in memory in three sections:

```
      0 -  84,999 starting at segment  6, offset 30,000
 85,000 - 131,071 starting at segment 14, offset 0
131,072 - 199,999 starting at segment 19, offset 45,000
```

The subroutine is called with the address of the virtual-to-segmented address mapping table in one double-word register and the virtual address in another. For this example, the mapping table takes the form shown in Figure 2-5.

```
MAPTAB:     0;     0     !V0=0!
         %600;30000      !S0=(6,30000)!
            1;19464      !V1=85,000 (= 2^16 + 19464)!
         %E00;     0     !S1=(14,0)!
            2;     0     !V2=131,072!
        %1300;-20536     !S2=(19,45000)!
            3; 3392      !V3=200,000 (= 3 x 2^16 + 3392)!
            0;     0     !Two 32-bit zeros terminate!
            0;     0
```

The means of expressing the 32-bit constants and segmented addresses depends upon the specific assembler used. Simpler ways are possible with some assemblers.

Figure 2-5

In this example, suppose that RR2 contains a virtual address, that is, an index between 0 and 199,999. It can be translated into a segmented address with the following code:

```
LDA RR4,MAPTAB
CALL ADMAP
```

The code for the high-flexibility solution appears in Figure 2-6.

```
!Address-mapping subroutine (high-flexibility version)
   CALL ADMAP with RR2 = virtual address (23 bits)
                   RR4 = address of mapping table
   Returns with C=0 and RR2 = segmented address; or
                C=1 if virtual address out of range
   The contents of RR4 are lost.
!
ADMAP:  INC R5,#8        !Step to next entry!
        TESTL @RR4       !Terminator?!
          JR Z,ERREX     ! Yes-out of range!
        CPL RR2,@RR4     !Compare v with Vi!
          JR GE,ADMAP    ! If v ≥ Vi, try next!
FIND:   DEC R5,#8        !Back up to Vi-1!
        SUBL RR2,@RR4    !RR2 = v-Vi-1!
        INC R5,#4        !Step to Si-1!
        ADDL RR2,@RR4    !RR2 = Si-1 + (v-Vi-1)!
        ADDB RH2,RL2     !Carry overflow to segment field!
        CLRB RL2         !Clear "reserved" bits!
        RESFLG C; RET    !C=0 for success return!
ERREX:  SETFLG C; RET    !C=1 for out-of-range return!
```

Figure 2-6

Notes:

1. The algorithms here are designed for random access. A loop to step through a byte array addressed for the high-efficiency version uses the following sort of address computation:

```
                LDL RR2,RR4      !Start at the beginning!
        LOOP:   !If at end of array, exit!
                !Perform operation on the array element!
                INC R3           !Step to next address!
                  JR NZ,LOOP     !Still in the segment!
                INCB RH2         !New segment!
                JR LOOP
```

2. Notice the use of the V_0 entry in the MAPTAB table. Even though V_0 can only be zero, the program is simplified by including an entry for it in the table.

3. There is no error checking performed in either routine. Several errors can occur: RR2 can contain a virtual address of greater than 23 bits, or MAPTAB can be incorrectly formed or can define an array that overlaps itself.

   The checking of RR2 in either version must be done dynamically. The checking of MAPTAB can be done once when the table is created or each time it is changed. A special routine can be provided for this purpose.

4. The DEC R5,#8 and INC R5,#4 instructions in the mapping computation are required because the Based Addressing mode cannot be used with the ADD and SUB instructions. If it could, the code at FIND might be

```
        FIND:  SUBL RR2,RR4(#-8)
               ADDL RR2,RR4(#-4)
```

   In the nonsegmented mode, indexed addressing can be used to simulate based addressing (see Section 1.2 Addressing Modes), but of course, this program would not be used in nonsegmented operations.

5. Many applications using large arrays do not need to have the entire array in memory at all times. The high-flexibility version of address mapping can be used to implement a demand-loading scheme. For this, the code at "FIND:" must recognize a special value for the base segmented address S_{i-1} that signifies that the array section in question is not currently present in main memory. ($S_{i-1}=2^{31}-1$ is a good value for this purpose.) At this point a call can be made on a demand-loading routine that loads the section in question and passes back its actual segmented address for storage in the address-mapping table.

2.4 REMOVING TRAILING BLANKS

Problem: To replace a fixed-length array of text (such as a card image) by a possibly shorter array containing the initial segment of the original array up to and including the last non-blank character. This type of operation is useful when a set of fixed-length arrays (for example, a card deck) is to be read into memory. Elimination of trailing blanks allows more records to fit into a buffer of given size.

Solution: The Z8000 block instructions that use the autodecrement mode are designed to handle this sort of problem. The array is scanned backward until the first non-blank character is found. The code for this appears in Figure 2-7.

```
!Subroutine to remove trailing blanks
   CALL STRIP with RR2 = address of the array
                    R1 = # of bytes in the array (1 to 65,536)
   Returns with R0 = number of bytes in stripped array.
   The contents of R0, R1 and RR2 are lost.
!
BLANK=32            !ASCII Code for blank!

STRIP:              LDB RL0,#BLANK          !Comparison character!
                    ADD R3,R1               !Set RR2 to point!
                    DEC R3                  ! at end of array!
                    CPDRB RL0,@RR2,R1,NE    !Scan backward to non-blank!
                    LD R0,R1                !Remaining count (Z not affected)!
                      RET NZ                !If all-blank return R0=0!
                    INC R0                  !Count the final non-blank!
                    RET
```

Figure 2-7

Notes:

1. Notice the computation to set RR2 to point at the last byte of the array. R3 is the offset portion of the address in RR2. Adding R1 (the number of bytes in the array) to R3 leaves RR2 pointing at the first byte following the array. DEC R3 brings RR2 back to its array. (R1 = 0 means 65,536 bytes.)

2. The Block Compare instruction terminates when the count in R1 reaches zero or when one of the CPB RL0,@RR2 operations causes the NE condition to be true. R1 is decremented for each comparison, whether or not there is a match. Therefore, if a match occurs (which the block compare instruction signals by setting Z), the count remaining in R1 is one less than the number of bytes in the stripped array. If no match occurs, R1 is decremented to zero, which is equal to the number of bytes in the stripped array.

2.5 DETERMINING WHETHER A 16-BIT WORD IS A BIT PALINDROME

Problem: To determine whether or not a given 16-bit word satisfies the condition

Bit n = Bit (15-n)

for n = 0, 1, 2, ..., 15. A word meeting this condition is called a bit palindrome, since it reads the same frontwards and backwards.

Solution: This problem illustrates the use of the Z8000 bit-testing instructions that allow the bit number to be specified in a register. The solution given here is the straightforward one: comparing bit n with bit (15-n) for n = 0, 1, 2, ..., 7. The code appears in Figure 2-8.

```
        !Subroutine to test for bit palindromes
         CALL BITPAL with R0 = 16-bit word to be tested.
         Returns with C=1 if not a bit palindrome, C=0 if it is.
         Register use: R1 = n; R2 = 15-n; RH3 = loop count; RL3 = scratch.
        !
        BITPAL: CLR R1          !Set n = 0!
                LDK R2,#15      ! and 15-n = 15!
                LDB RH3,#8      !Set loop counter!
        LOOP:   CLRB RL3
                BIT R0,R1       !Test Bit n and!
                TCCB NZ,RL3     ! move it into RL3!
                RLB RL3         !Make room for Bit 15-n!
                BIT R0,R2       !Test Bit 15-n and!
                TCCB NZ,RL3     ! move it into RL3!
                TESTB RL3       !Bit n = Bit 15-n if and!
                  JR PO,NOTPAL  ! only if parity of RL3 is even!
                INC R1          !Increment n!
                DEC R2          !Decrement 15-n!
                DBJNZ RH3,LOOP  !Loop until count exhausted!
                RESFLG C; RET   !Success: set C=0 and return!
        NOTPAL: SETFLG C; RET   !Failure: set C=1 and return!
```

Figure 2-8

Notes:

1. This example illustrates the operation of the Bit Test instruction. A more efficient solution to the problem involves a direct comparison of the two bytes of R0 after reversing one of them with a loop like:

```
        LDK R2,#8
LOOP:   RLCB RL0
        RRCB RL1
        DJNZ R2,LOOP
        RLCB RL0
```

2. The condition code NZ is used in the TCC instructions. BIT sets Z if the bit is clear and clears Z if the bit is set.

3. TCC instructions are used to save the bit values, and TEST is used to compare them by testing the parity of the byte into which they have been stored. Both simplify the flow of control. Not using these techniques results in the sort of jumping around shown in Figure 2-9.

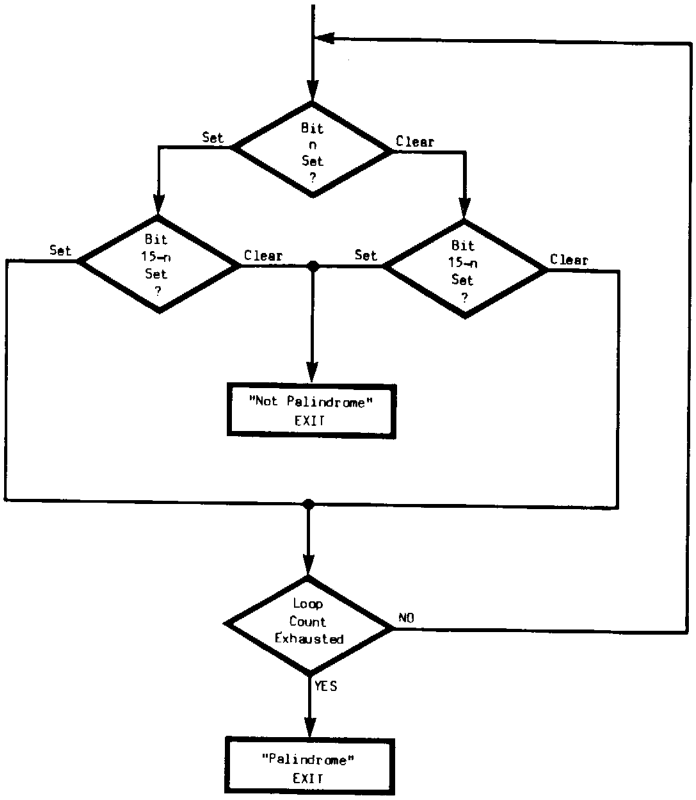

Figure 2-9. **A Poor Alternative to the Use of TCC**

2.6 SORTING

Problem: Given an array A of "items" and an order relation "$\leq$", rearrange the items of A in such a way that for integers i, j and items a_i, a_j, $a_i \leq a_j$ whenever $i \leq j$. The items of A can be integers, floating point numbers, character strings or any other data type. The order relation can be any ordering appropriate to the given data type, for example, dictionary order for character strings.

Solution: An adaptation of the "quicksort" algorithm of C.A.R. Hoare is used. A program is written to sort an array of 16-bit 2's complement integers in ascending numerical order. The organization of the program into subroutines indicates how other items and orderings can be used.

Assume that A is an array indexed from 0 to N. Quicksort is a recursive procedure that begins by arbitrarily selecting one of the items of A as the "pivot" value. Then a preliminary rearrangement of A is made as follows: For some i, $0 \leq i \leq N$, a_i is the pivot value and $a_k \leq a_i$ if $0 \leq k \leq i$, $a_k \geq a_i$ if $i < k \leq N$. That is, all items less than or equal to the pivot are moved into the "left half" of the array and all those greater than or equal to the pivot are moved into the "right half."

Once this is done, the same process is performed on each of the two array segments a_0 to a_{i-1} and a_{i+1} to a_N. These segments are usually not of equal size. Implementation of the algorithm requires a minimum of stack storage if at each stage the smaller segment is sorted first.

In this example assume that array offsets are 23-bit numbers in the range of 0 to 8,388,607 and that the array elements are 16-bit signed integers. A base segmented address and an address computation similar to that of the high-efficiency version of ADMAP (Section 2.3) are used. The generalization to other types of element is straightforward. The code for this appears in Figures 2-10 through 2-15.

```
        !Subroutine Quicksort
         CALL QUICK with RR12 = array address
                         RR10 = U (offset of upper limit)
                         RR8  = L (offset of lower limit)
         Returns with array elements at offsets between L and U (inclusive)
         sorted.  L and U are 23-bit integers in the range 0 to 8,388,607.
         Register use:
                RR14: Stack Register
                RR12: Always contains starting segmented address of array
                 RQ8: (L,U) on call; shorter (L,U) range returned by SHORT
                 RQ4: longer (L,U) range returned by SHORT
                 RQ0: used by subroutines of QUICK
        !
         QUICK: CPL RR8,RR10     !Compare L,U!
                  RET GE         !Return if L > U!
                CALR PART        !Partition: RQ4, RQ8 get ranges!
                CALR SHORT       !Put shorter range in RQ8, longer in RQ4!
                DEC R15,#8       !Save RQ4 - longer (L,U) range!
                LDM @RR14,R4,#4
                CALR QUICK       !Recursive call to sort the shorter range!
                LDM R8,@RR14,#4  !Restore longer range - into RQ8!
                INC R15,#8
                CALR QUICK       !Recursive call to sort the longer range!
                RET
```

Figure 2-10

```
!Subroutine of QUICK to put shorter range first
 CALL SHORT with RQ4 = one (L,U) range
                 RQ8 = another (L,U) range
 Returns with shorter range in RQ8, longer in RQ4
 Register use: as for QUICK.  RR0 contents are lost.
!
SHORT:  LDL RR0,RR6      !RR0 = U-L for RQ4!
        SUBL RR0,RR4
        PUSHL @RR14,RR0  !Save first U-L!
        LDL RR0,RR10     !RR0 = U-L for RQ8!
        SUBL RR0,RR8
        CPL RR0,@RR14    !Compare lengths!
        POPL RR0,@RR14   !Clear the stack!
          RET LE         !Return if RQ8 length ≤ RQ4 length!
        EX R4,R8         !Exchange RQ4 & RQ8!
        EX R5,R9
        EX R6,R10
        EX R7,R11
        RET
```

Figure 2-11

```
!Partitioning subroutine of QUICK
 CALL PART with registers as for QUICK
 Returns with array segment between L and U partitioned
 around a pivot element with index I.  Returns the two
 ranges to be sorted: (L,I-1) in RQ8 & (I+1,U) in RQ4.

 Register use: RQ8 = (L,U); RQ4 = (I,J).  On return,
 RQ4,RQ8 are new ranges.  RQ0 is used by subroutines.
!
PART:   CALR SETPIV      !Choose pivot; initialize pivot routines!
        LDL RR4,RR8      !Set I = L!
        LDL RR6,RR10     !Set J = U!
        CALR DECI        !Decrement I: I=L-1!
LPI:    CALR UPI         !Increment I until a_I ≥ pivot value!
        CALR DOWNJ       !Decrement J until a_J ≤ pivot or J ≤ I!
          JR C,MOVPIV    !J ≤ I: only pivot remains to be moved!
        CALR EXCHIJ      !Exchange a_I and a_J values!
        JR LPI
MOVPIV: CALR EXCHIP      !Exchange a_I and pivot values!
        LDL RR6,RR10     !Move I to end of RQ4 (where J was)!
        LDL RR10,RR4     !Move I to end of RQ8 (where U was)!
        CALR DECI        !Decrement I: RR4 = I-1!
        EX R4,R10        !Exchange RR4,RR10:        !
        EX R5,R11        !Now RQ8 = (L,I-1); RR4 = I!
        CALR INCI        !Increment I: Now RQ4 = (I+1,U)!
        RET
```

Figure 2-12

```
!Subroutines of PART for moving I and J
 CALL UPI: returns with I incremented until aI ≥ pivot value
 CALL DOWNJ: returns with J decremented until aJ ≤ pivot
 or J ≤ I; returns C=1 if J ≤ I, otherwise C=0
 Register use: As for PART.
!
UPI:    CALR INCI          !Increment I!
        CALR CPPI          !Compare pivot value with aI!
          RET LE           !Return if pivot value ≤ aI!
        JR UPI             !Otherwise keep incrementing!

DOWNJ:  CALR DECJ          !Decrement J!
        CPL RR4,RR6        !Compare I,J!
          JR LT,DJ1        !I < J: proceed!
        SETFLG C; RET      !J ≤ I: return C=1!
DJ1:    CALR CPPJ          !Compare pivot with aJ!
          JR LT,DOWNJ      !Keep decrementing if pivot value ≤ aJ!
        RESFLG C; RET      !Otherwise return with C=0!

!Routines to increment or decrement I or J.
ESIZE = 2                  !Entries are words: two bytes!
INCI:   ADDL RR4,#ESIZE
        RET
DECJ:   SUBL RR6,#ESIZE
        RET
DECI:   SUBL RR4,#ESIZE
        RET
```

Figure 2-13

```
!Pivot Setting and Comparison Subroutines
   CALL SETPIV - chooses pivot & saves its value in a
 register
 CALL CPPI - compare pivot value, aI. Set FLAGS.
 CALL CPPJ - compare pivot value, aJ. Set FLAGS.
 Register use: as for PART. R0 = temp. R1 = saved pivot
 value.  RR2 = calling argument and actual address
 returned by ADCOMP
!
SETPIV: LDL RR2,RR10       !RR2 = U!
        CALR ADCOMP        !RR2 = actual address of aU!
        LD R1,@RR2         !Choose aU as pivot value!
        RET

CPPI:   LDL RR2,RR4        !RR2 = I!
        JR IJM
CPPJ:   LDL RR2,RR6        !RR2 = J!
IJM:    CALR ADCOMP        !RR2 = adr of item to be compared!
        CP R1,@RR2
        RET
```

Figure 2-14

```
!Exchange Subroutines
 CALL EXCHI - exchange $a_I$ and pivot values.
 CALL EXCHIJ - exchange $a_I$ and $a_J$ values.
 Register use: as for PART. R0 = temp. R1 = saved pivot
 value.  RR2 = calling argument and actual address
 returned by ADCOMP
!
EXCHIJ: LDL RR2,RR4       !RR2 = I!
        CALR ADCOMP       !RR2 = actual address of $a_I$!
        LD R0,@RR2        !R0 = $a_I$!
        PUSHL @RR14,RR2   !Save address of $a_I$!
        LDL RR2,RR6       !RR2 = J!
        CALR ADCOMP       !RR2 = actual addresss of $a_J$!
        EX R0,@RR2        !Exchange: R0=$a_J$, $a_J$ replaced by $a_I$!
        POPL RR2,@RR14    !Restore $a_I$ address!
        LD @RR2,R0        !Replace $a_I$ by $a_J$!
        RET

EXCHIP: LDL RR2,RR4       !RR2 = I!
        CALR ADCOMP       !RR2 = actual address of $a_I$!
        EX R1,@RR2        !Exchange $a_I$ with saved pivot value!
        LDL RR2,RR10      !RR2 = U (offset of pivot element)!
        CALR ADCOMP       !RR2 = actual address of $a_U$!
        LD @RR2,R1        !Replace $a_U$ by $a_I$!
        RET

ADCOMP: ADDL RR2,RR12     !Add array base to offset!
        ADDB RH2,RL2      !Carry overflow into segment field!
        CLRB RL2          !Clear reserved bits!
        RET
```

Figure 2-15

Notes:

1. This code falls into two principal categories: the code to implement the algorithms and the code to manipulate the indices and data items. The algorithm is implemented by the routines QUICK, PART, SHORT, UPI, DOWNJ and SETPIV. The manipulation and comparison of data items and the arithmetic on array indices occur in the routines INCI, DECI, DECJ, CPPI, CPPJ, EXCHIP, EXCHIJ, and SETPIV. The mapping of array offsets into actual memory addresses occurs in ADCOMP.

   The organization used here facilitates the alteration of QUICK for other applications. For example, a nonsegmented version can be produced simply by changing all instances of @RR2 to @R3 and keeping the nonsegmented array address in R13 with a zero in R12. All references to RR14 also have to be changed to refer to R15. The resulting code is less efficient than a tailor-made nonsegmented version, but this does not matter in many applications.

As another example, QUICK could be changed so that it sorts bytes by redefining the symbol ESIZE to take the value 1. Instead of using R0 as a temporary location and R1 for the saved pivot value, the routines SETPIV, CPPI, CPPJ, EXCHIP, and EXCHIJ need byte registers. Then the four LD instructions, the CP instruction, and the two EX instructions in those routines must be changed to byte versions.

Sorting on the basis of other ordering relations is facilitated by this program orgnization. For example, decreasing numerical order could be used simply by replacing the instruction CP R1,@RR2 with:

```
LD R0,@RR2
CP R0,R1
```

in the CPPI/CPPJ routine (CP @RR2,R1 is not a legal instruction). The program could have byte constants representing the various flags combinations it wishes to return. For example, the less than condition can be returned by the following sequence of instructions at the end of the subroutine:

```
LDB RH0,#LTVAL
LDCTLB FLAGS,RH0
RET
```

The symbol LTVAL might have the value %20, corresponding to C = 0, Z = 0, S = 1, V = 0, D = 0, H = 0.

2. The CPPI and CPPJ routines illustrate the useful programming technique of multiple entry points. An alternative organization is

```
CPPI: LDL RR2,RR4
      CALR IJM
      RET
CPPJ: LDL RR2,RR6
      CALR IJM
      RET
```

The code at IJM in both organizations is shared. The objective of this is not principally to save memory space but rather to assure that these two related functions are carried out according to a common algorithm.

3. The SETPIV routine is mainly concerned with data manipulation, but it also implicitly embodies a part of the quicksort algorithm, the choice of a pivot element. Use of a_U for the pivot is inefficient if the array is already sorted. Other algorithms can be used to make the choice.

4. The use of 23-bit indices stored in long-word registers simplifies index comparisons such as those that occur in QUICK and SHORT. To use the same code for one-word registers, the index values would have to be restricted to 15 bits. If 16-bit indices are used, the comparisons must be the unsigned versions. In that case, special tests must be made for the case L > U, in both SHORT and QUICK. In particular, the case U = -1, L = 0, a termination condition for QUICK, needs further special handling.

2.7 POLYNOMIAL EVALUATION

Problem: Given a set of coefficients a_0, $a_1,\ldots,a_n$ and a variable x, compute

$$f(x) = a_0 + a_1x + a_2x^2 + \ldots a_nx^n.$$

Solution: The coefficients $a_0,\ldots,a_n$, the variable x, all of the products a_kx^k, the intermediate sums, and the final sum are assumed to be within the range of 32-bit signed integers, -2^{31} to $2^{31}-1$.

A subroutine is provided that accepts as its arguments the variable x and the address of a parameter table describing the array. The table has the following format:

```
n  (1 word)
a0 (2 words)
 .
 .
 .
an (2 words)
```

The subroutine returns the value f(x). In addition, the results of computations are checked at each stage to verify that they remain within the stated bounds. If the bounds are exceeded at any stage, V is set when the subroutine returns its final result. The code of this routine appears in Figure 2-16.

The code is arranged so that multiplications are required at two places. In each case, the arguments are manipulated in the registers so that the required instruction is

```
MULTL RQ8,RR6
```

A subroutine is provided to execute this instruction and to verify that the result fits into RR10, the low-order half of RQ8. If not, a bit is set in an error-flag register that is initially cleared to zero by the main routine. The code for the multiply and check routine is shown in Figure 2-17.

```
!Subroutine to perform polynomial evaluations!
   CALL POLY with RR0 = x
                  RR2 = adr. of table (n, a0, ..., an)
   Returns with   RR4 = f(x), contents of RR2 and R6-R13 lost
                  V = 0 if all values in bounds, 1 otherwise.
   Register use:  RR0, RR2 -- calling arguments
     RR4 -- running sum       R12 -- coefficient counter
     RR6 -- x^k (k=0,1,...,n) R13 -- error flag
     RQ8 -- scratch
!
POLY:    POP R12,@RR2     !Get n from table to set counter!
         LDL RR6,#1       !Initialize: x^k = 1 (i.e., k = 0) !
         LDL RR4,#0       !            f(x) = 0              !
         CLR R13          !            Error flag = 0        !
LOOP:    POPL RR10,@RR2   !Get ak from the table!
         CALR MULCH       !RR10 = akx^k!
         ADDL RR4,RR10    !f(x) = f(x) + akx^k!
         TCC OV,R13       !Remember overflow, if any!
         DEC R12          !Decrement coefficient counter!
           JR MI,POLEX    !  Done if < 0!
         LDL RR10,RR0     !Get x!
         CALR MULCH       !RR10 = x^k+1!
         LDL RR6,RR10     !Replace x^k by x^k+1 (i.e., increment k)!
         JR LOOP          !Perform computation for new k!
POLEX:   RESFLG V
         TEST R13         !Were there any overflows?!
           RET Z          !  No -- return with V = 0!
         SETFLG V; RET    !  Yes -- return with V = 1!
```

Figure 2-16

```
!Multiply and check subroutine!
MULCH:   MULTL RQ8,RR6    !Perform the multiplication!
         PUSHL @RR14,RR8  !Save high-order 32 bits!
         EXTSL RQ8        !Set high-order 32 bits to proper value!
         CPL RR8,@RR14    !Was it already OK?!
         TCC NE,R13       !If not, then overflow occurred!
         INC R15,#4       !Discard saved RR8!
         RET
```

Figure 2-17

Notes:

1. Notice the structure of the loop in POLY. There is no test at the beginning, so the loop is always executed at least once. The effect of this is that tables with negative values of n will be treated as if they had n = 0.

   There is also no test at the end of the loop. Instead, the decrement of the coefficient counter and the test for termination appear immediately following the latest update of the running sum and before the computation of x^{k+1}. The overall length of the program can be shortened by moving this test to the end of the loop, but then x^{n+1} is always computed unnecessarily. In addition to the wasted computation, this leads to an erroneous overflow indication if x^{n+1} exceeds the 32-bit limitation.

2. The subroutine MULCH illustrates the use of the multiplication and sign extension instructions. The instruction

   ```
   MULTL RQ8,RR6
   ```

   causes the contents RR10 (the low-order half of RQ8) to be multiplied by the contents of RR6 and the resulting value to be stored in RQ8. The original contents of RQ8 (the high-order half of RQ8) are irrelevant.

   The instruction

   ```
   EXTSL RQ8
   ```

   causes the contents of RQ8 to be replaced by a number whose value is the same as that of RR10 but which has twice as many bits. Assuming that all results are within the range of signed 32-bit numbers, the EXTSL instruction should cause no change to RR8. This explains the test performed in MULCH.

3. The use of the TCC instruction to remember the occurrence of overflows is similar to its use in Section 2.1.

2.8 PSEUDO-RANDOM NUMBER GENERATION

Problem: To provide a subroutine that returns an "unpredictable" 16-bit number.

Solution: The solution presented is sometimes referred to as the power residue method. A large positive number N with few prime factors is chosen. The values returned by the function RND on successive calls 1,2,... are defined as follows:

$$RND_1 = N^2 \pmod{2^{16}}$$

$$RND_k = (RND_{k-1} \text{ AND } (2^{15}-1)) \times N \pmod{2^{16}}$$

for k = 2,3,...

The algorithm used requires that the routine know at each stage the value it returned when last called. The storage space for remembering this value is provided by the caller in a table whose address is passed to the routine each time it is called. An initializing routine is provided for setting up the table. Figure 2-18 shows the code for the initializing routine and the pseudo-random number generator.

```
!Random-number routines
   CALL INRAND with RR2 = address of 2-word temp storage table.
   Returns with table "initialized," R1 lost, and R0 = N.

   CALL RAND with RR2 = address of the table.
   Returns with R0 = "random" number & table updated.

   Register use:
     RR0:  Dest for multiplication; R0 returns the random number.
     RR2:  address of table.
!
N = 15419                   !N = 17*907!

RAND:    LD R1,RR2(#2)      !R1 = RNDk-1!
         RES R1,#15         !R1 = RNDk-1   AND 2^15-1!
         MULT RR0, @RR2     !RR0 = (RNDk-1 AND (2^15-1))*N!
         LD R0,R1           !R0 = RNDk!
         LD RR2(#2),R0      !Remember RNDk for next call!
         RET

INRAND:  LD R0,#N
         LD @RR2,R0         !Save N in table!
         LD RR2(#2),R0      !RND0 = N!
         RET
```

Figure 2-18

Notes:

1. This is a quick and dirty pseudo-random number generator. For a thorough discussion of random-number theory and algorithms, refer to Chapter 3 of "The Art of Complete Programming, Volume 2: Seminumerical Algorithms," by Donald E. Knuth.

2. Similar routines can be used for 32-bit random numbers. In fact, RAND could be generalized to take its argument size from the table. The desired size could be passed to INRAND, which would set up the table accordingly.

3. The choice of the number N could be made by the caller and passed, possibly as an option, to INRAND.

4. Note the use of the instruction

```
RES R1,#15
```

as an alternative to

```
AND R1,#%7FFF.
```

5. Note that the use of an argument table makes RAND a re-entrant routine.