Z8000 PLZ/ASM
Assembly Language Programming Manual

Preface

This manual describes the PLZ/ASM assembly language for the Zilog Z8000 microprocessor, and serves as the primary reference manual for the assembly language programmer. It is one in a series of documents describing the Z8000 and associated hardware and software. Information on the use of the PLZ/ASM Assembler can be found in the publication:

Z8000 Assembler User's Guide

It is intended that this manual be read the first time in sequence, from beginning to end.

Chapter 1 provides an overview of the architecture of the Z8000, with special emphasis on those features of interest to the assembly language programmer. A detailed description of the architecture and hardware-related features can be found in the publication:

Z8000 CPU Technical Manual (To be published)

Chapter 2 introduces assembly language conventions, including the format of statements and addressing mode specifications.

Chapter 3 contains the detailed instruction set of the Z8000.

Chapter 4 introduces the high-level PLZ/ASM statements needed to construct a complete program, while Chapter 5 contains the detailed description of these statements.

The Appendices contain a summary of the instruction set, high-level statements and assembler directives, plus a table of the ASCII character set.

Contents

CONTENTS (continued)

CONTENTS (continued)

LIST OF TABLES

LIST OF ILLUSTRATIONS

CONTENTS (continued)

Section 1
Architectural Overview

1.1 INTRODUCTION

Zilog's Z8000 microprocessor has been designed to accommodate a wide range of applications, from the relatively simple to the large and complex. Depending on the Z8000 configuration chosen, the programmer can directly address from 64 kilobytes (64K or 65,536 bytes) to 8 megabytes (8M or 8,388,608 bytes) of memory.

The Z8000 achieves high throughput with a relatively low clock rate (standard: 4MHz) and can, therefore, use memories with a comparatively long access time. Built-in random-access memory (RAM) refresh with a programmable refresh rate permits the use of a wide variety of dynamic memories.

Z8000 central processing unit (CPU) resources include sixteen 16-bit general-purpose registers, seven data "types" (lengths from single bits to 32-bit long words), eight addressing modes, and a repertoire of 105 instructions. These resources are similar in form to comparable features of the Z8-microcomputer and Z80-microprocessor families, allowing users of these systems to upgrade easily to the Z8000.

Over 410 meaningful combinations of instructions, data types, and addressing modes are available with the Z8000. The instruction set also includes signed multiplication and signed division (implemented in hardware) for both 16-bit and 32-bit values.

The Z8000 provides several sophisticated features not found on other microprocessors. Since reference to these features are made throughout this manual, a brief description of each of the major concepts and terminology follows:

- Segmentation

  To facilitate the management of a large address space, the Z8000 allows program and data to be accessed as part of variable-length logical groupings called "segments". Segmentation provides hardware assistance for relocation of program and data, as well as protection against accidental or malicious damage to system or user information (see Section 1.4).

- Stacks

  The Z8000 has several instructions which facilitate the handling of stacks (Section 1.5.3). A "stack" is an area of memory used for temporary storage or re-entrant procedure call/interrupt service linkage information, and is managed in a "last-in, first-out" manner; that is, several items may be "pushed" or added to the top of a stack and then "popped" or removed from the stack in reverse order. On the Z8000, a stack starts at the highest address allocated for it and grows linearly downward to the lowest address as items are pushed. A general-purpose register is used to "point" to the current top of the stack; that is, the register contains the address of the most recently-pushed item.

- Interrupts and Traps

  Two events can alter the normal execution of a Z8000 program: asynchronous hardware "interrupts" which occur when a peripheral device needs service, and synchronous software "traps" which occur when an error condition arises such as an attempted use of an unimplemented instruction. The Z8000 handles both of these cases in a similar manner (Section 1.6.4). The current status of the processor is pushed on a stack, and then program control is automatically "vectored" to a handler procedure for the particular class of interrupt or trap. The address of the handler is determined by indexing a table (or "vector") of program status blocks which is referred to as the Program Status Area, and extracting the appropriate entry that includes the address of the handler procedure. When the handler procedure is finished, control can be returned to the interrupted program through the status information saved on the stack. (Non-vectored interrupts are also possible, with the burden placed on the program to determine which device needs servicing.)

- Normal and System Operating Modes

  Z8000 programs run in one of two operating modes: Normal mode or System mode. Normal mode is the operating mode during normal program execution. System mode exists primarily to protect the operating system kernel. Instructions that alter the machine state (such as I/O operations, changes to control registers, etc.) can be issued only in System mode and are referred to as "privileged" instructions.

Attempting to execute privileged System mode instructions while in Normal mode initiates a trap. This is an important feature in a multi-user environment. Programs running in normal mode cannot take over or monopolize control of the Z8000 or damage other users' programs or system software. System mode is also in effect following an interrupt or trap.

Compiler and assembler code, the program code output by these translators, and operating system code, all run efficiently on the Z8000. Large address spaces, memory relocation, system and normal stacks, special instructions, and a sophisticated interrupt and trap structure add to this efficiency.

The Z8000 can also be part of a multi-microprocessor system using its exclusion and synchronization software instructions and its Micro Input and Micro Output hardware controls. The large address space of the Z8000 augments the overall data-processing capability of multi-microprocessor configurations.

From a programmer's point of view, the basic Z8000 system configuration consists of the Z8000 microprocessor and its various memory and I/O address spaces. These are described in the remainder of this section. For detailed architectural and configuration data, see the Z8000 Technical Manual.

NOTE

The Z8000 is designed to provide for future extensions to its architecture which would require several of the currently unused (but reserved) bits in some of the instruction and data encodings. Any shaded or hatched areas in the figures throughout this manual are considered reserved and should not be used by the programmer. Reserved bits must be zero.

1.2 MEMORY ADDRESS SPACES

To allow for a wide range of applications, two versions of the Z8000 microprocessor device are available: a 40-pin package (Z8002) with an address range of 0-64K bytes, and a 48-pin package (Z8001) with an address range of 0-8M bytes (Figure 1-1). The Z8002 version is called nonsegmented and the Z8001 is the segmented version (see Section 1.4.1).

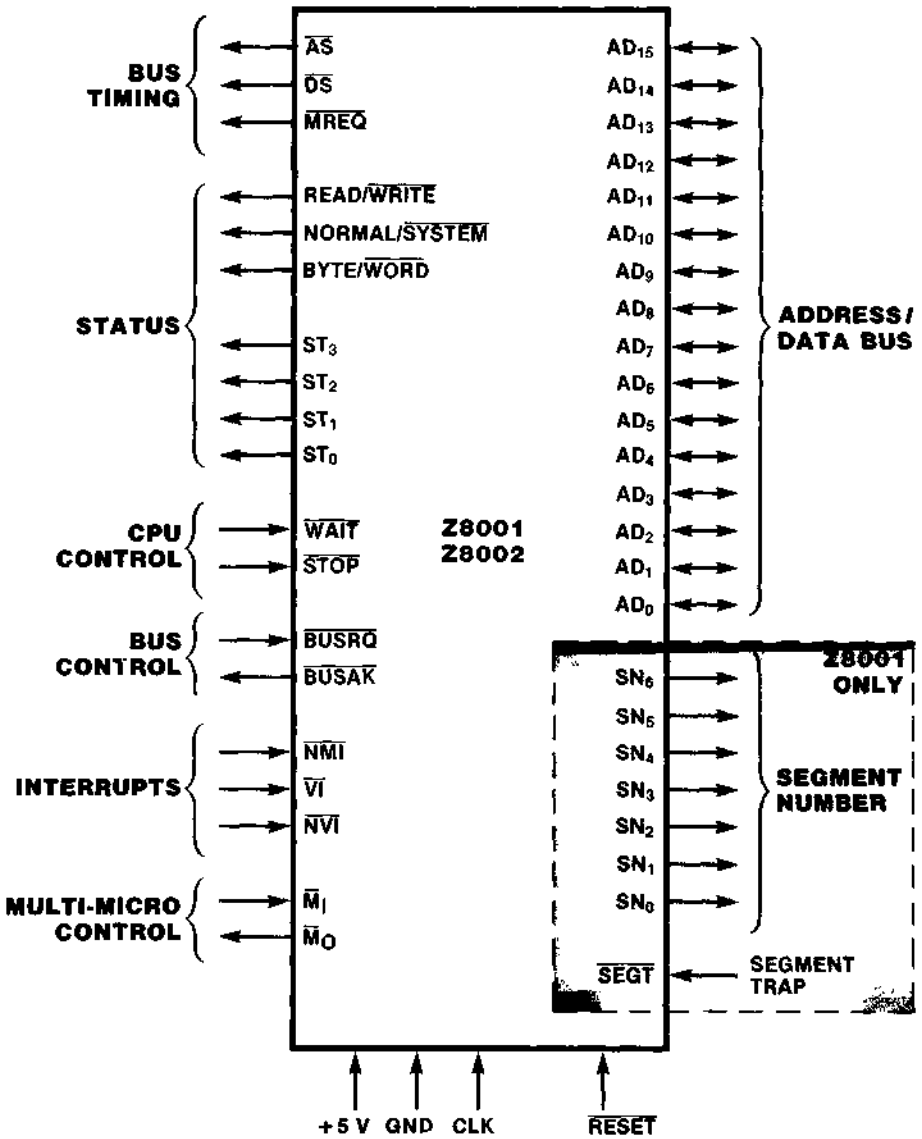


Figure 1-1 Z8000 Pin Functions

1.2.1 Four Memory Address Spaces

When memory is addressed, the Z8000 CPU distinguishes between

- CODE memory, containing program instructions, and
- DATA memory, including stack memory,

both of which can be accessed in Normal and System operating modes. Thus, the programmer can address four distinct memory address spaces: system code memory, normal code memory, system data memory, or normal data memory.

The specific space addressed is determined by the combination of outputs from the Z8000 status pins, ST0-ST3 (Figure 1-1). The status pins indicate various CPU operations such as memory or I/O references, interrupt or trap acknowledgements, memory refresh or internal operations. Data memory reference, stack memory reference, instruction fetch of the first word of an instruction (IF1), and instruction fetch of the nth word of an instruction (IFn) each correspond to different status pin outputs, so that each address space can be distinguished by the memory hardware.

Each of the four address spaces has a range as great as the addressing capability of the processor. For the user with a nonsegmented Z8002 microprocessor, this means each address space can have up to 64K bytes, for a total system capacity of 256K bytes of directly-addressable memory. Similarly, a segmented Z8001 provides 32 megabytes of directly-addressable memory (up to 8M bytes for each of the four address spaces).

1.2.2 Addressing Memory Spaces

Each memory address space can be viewed as a string of bytes numbered consecutively in ascending order. This "byte address" is the basic addressing element for the memory portion of a Z8000 system. The different methods for arriving at the byte address are discussed later in this section (Section 1.5.4).

The byte address is used not only to address bytes, but also to address bits, 4-bit Binary Coded Decimal (BCD) digits, 16-bit words, and 32-bit long words.

In the case of a bit or digit, the memory address designates the byte or word that contains it. For example, a bit can be addressed by specifying a byte address and the number of the bit within the byte (0-7), or by specifying a word address and the bit number within the word (0-15). Bits are numbered right-to-left; that is, least-to-most significant (Figure 1-2).

In the case of data types longer than one byte, the memory address designates the leftmost, or high-order byte. In other words, the high-order byte has the lowest memory address of the bytes within a word or long word (Figure 1-2).

Program instructions residing in code memory are always addressed as 16-bit words. Words and instructions are always aligned (the high-order byte must have an even-numbered address). Aligned words improve access speed and double the range of instructions that use relative addresses (JR, DJNZ, DBJNZ, CALR) -- see Section 1.5.4. The memory address of a long word must also be an even-numbered byte address.

NOTE

Word quadruples (64 bits) cannot be addressed in memory. It is possible to address 64-bit register quadruples, however. This is covered later, in Section 1.5.2.

1.3 INPUT/OUTPUT ADDRESS SPACES

The Z8000 has an I/O address space separate from the memory address space. I/O address references can be distinguished from memory addresses using the status pin outputs from the Z8000 (Figure 1-1).

I/O addresses are 16-bit addresses allowing a range of 0-64K bytes to be addressed. I/O addresses are multiplexed with the byte or word data accessed by the I/O operation. (See pins AD0 - AD15 of Figure 1-1.) I/O data references have an automatic wait cycle included for each load or store, which results in a 4-cycle read or write.

The Z8000 instruction set includes two groups of I/O instructions: a standard complement of input, output, and block transfer instructions, plus a "special" group of I/O instructions. The latter are generally used to load and examine the Memory Management Unit, a device used only in the segmented Z8000 system configuration (Section 1.4.2). Special I/O references can be distinguished from standard I/O references using the status pins (Figure 1-1.)

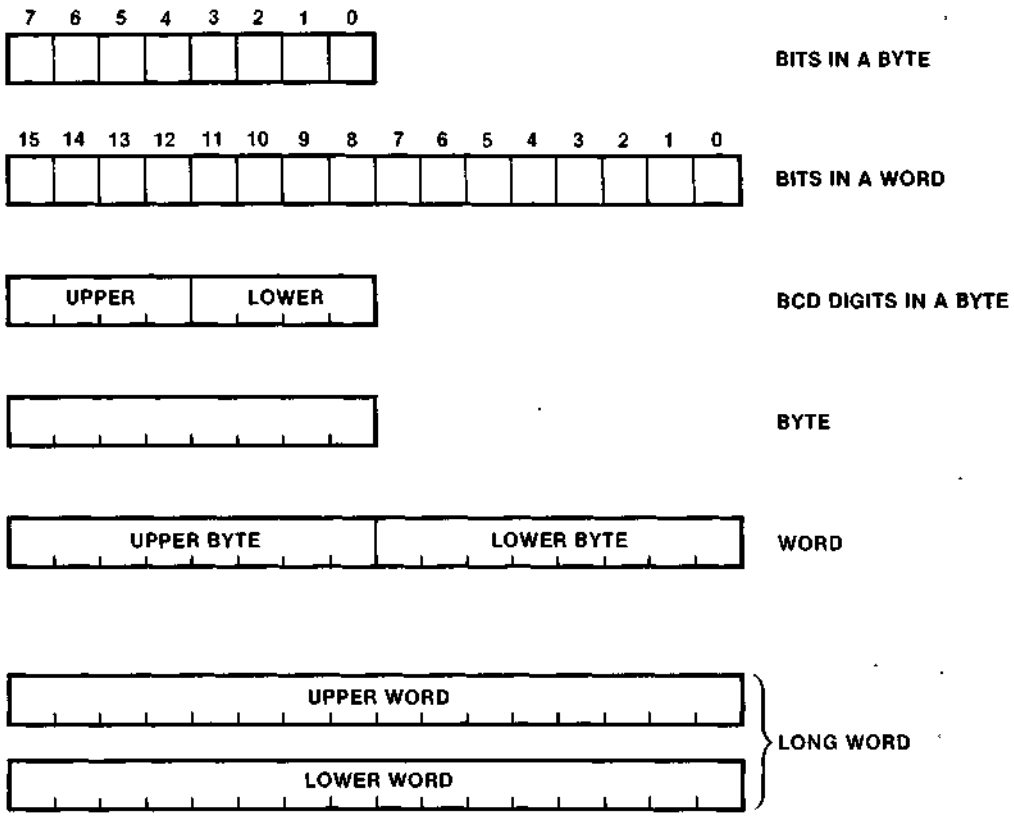


Figure 1-2 Addressable Data Elements

1.4 SEGMENTATION

The segmented Z8000 memory address space may be separated into several (up to 128) variable-length "segments". Each segment may vary in size, independently from other segments, from 0 to 64K bytes. Addresses consist of both a segment number and an offset. The segment number is used as an index into a table of base addresses for each segment. The table is kept in a separate package called the Memory Management Unit (MMU). The corresponding table entry provides a base to which the offset is added, thus providing the final address. The term "logical address" is used to indicate the segmented address used by the programmer to construct his program. The term "physical address" is used to indicate the translated address which is passed to the physical memory hardware. This address translation is accomplished completely in hardware and does not affect the instruction execution time.

Segmentation provides:

1) Dynamic relocation of program and data memory without necessitating the modification of addresses. Since all instructions use addresses which are relative to the segment base registers, code and data segments can be relocated anywhere in memory simply by moving the information and setting the segment base registers (in the MMU) to new values. This provides great flexibility for multi-user or multi-task operating systems in efficiently managing memory allocations.

2) Hardware-assisted memory protection to insure that only valid addresses within the bounds of the user's code and data segments are accessed. Since segments can be of variable length, only the amount of memory needed for each segment needs to be specified (independently from other segments).

3) Large logical address space without necessitating large physical address space ("virtual memory"). Since logical addresses are 23 bits, up to 8M bytes per address space can be accessed. This space usually exceeds the size of the available physical memory address space. However, by using a combination of the MMU, some external hardware and some operating system software, a virtual memory scheme can be employed which detects accesses to logical memory which are not currently mapped to physical memory, thus allowing the loading of segments from external secondary memory devices on demand.

4) Controlled sharing of memory with several Z8000 CPUs. Logical segments may be shared by several microprocessors by using the same MMU, or the same physical memory segments may be shared with several MMUs mapping different microprocessors'

logical address spaces into the same physical addresses. Relocation, protection and/or virtual memory facilities may be shared by several processors as well.

The nonsegmented Z8002 uses 16-bit addresses that can be manipulated as words. This version can directly address 64K bytes of memory in each of its four address spaces. The segmented, Z8001 version uses 23 bits to address directly up to 8M bytes in each of its four address spaces. The basic difference between programs running in nonsegmented or segmented modes is the number of bytes used to form addresses in instructions or registers.

Code written for a nonsegmented Z8000 can run in one segment of a segmented Z8000. This is called "running the segmented Z8000 in nonsegmented mode". The converse is not possible; i.e., code written for a segmented Z8000 will not run properly on a nonsegmented Z8000 due to differences in the instruction formats, and the use of register pairs for addressing modes. The functionality of the two versions is identical in all respects other than memory addressing.

1.4.1 Nonsegmented and Segmented Addresses

All nonsegmented addresses are represented as 16-bit words, whether they reside in a register, in memory, or as part of an instruction.

Segmented addresses require 23 bits. Each 8-megabyte address space is divided into 128 segments from 0 to 64K bytes each. Thus, to address a byte in one of these spaces, two pieces of information are needed:

- the segment number (0-127), which can be expressed in 7 bits, and
- the offset from the beginning of the segment (i.e., the 0-64K byte address within the segment), which can be expressed in 16 bits.

The two parts of a segmented address may be manipulated separately. Word functions, including 16-bit arithmetic, can be performed on the offset portion.

Figures 1-3 and 1-4 show internal representations of segmented addresses. The two words shown in Figure 1-3 could represent a long word in memory (Section 1.2.2) or a register pair in the CPU (Section 1.5.2). Within instructions, segmented addresses can have a long 16-bit offset or a short 8-bit offset. As Figure 1-4 indicates, bit 7 of the segment-number byte is used to differentiate between these two formats.

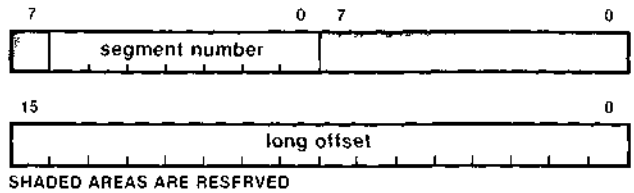


Figure 1-3 Segmented Address (Register Memory)

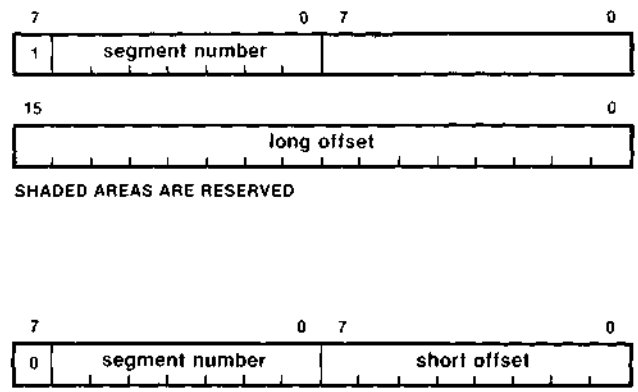


Figure 1-4 Segmented Address Within Instruction

1.4.2 Memory Management

The segmented Z8000 configuration usually includes a separate package called the Memory Management Unit (MMU). The MMU pin functions are pictured logically in Figure 1-5 and described below. These functions are divided between:

- segment address relocation, and
- memory protection.

The MMU relocates segment addresses by converting them from "logical" to "physical" addresses. Logical addresses are those manipulated by the program or specified in instructions, and output by the Z8000 CPU. Physical addresses are actual hardware locations. This address translation occurs automatically in the MMU and requires no programmer intervention (see Figure 1-6).

As Figure 1-5 illustrates, the inputs to the MMU include the segment number and the upper eight bits of the offset. Each segment number is associated with a 24-bit "base" address equivalent to the first physical address in the segment. The 16-bit offset is added to this base to complete the 24-bit physical address. Note that the lower eight bits of the offset are passed directly to the physical memory, so that the MMU need only store the upper 16 bits of each base address (bits 8-23). Figure 1-7 is another representation of the addition done by the MMU to form the 24-bit physical address.

The other major function of the MMU is memory protection. By interpreting its four status lines (Chip Select, Address Strobe, Data Strobe, and Read/Write), the MMU can check

- System vs. normal segment status
- Code vs. data status
- Read/write vs. read-only status
- Invalid entry
- Segment size

During each memory reference, the attributes for each segment are checked against the corresponding Z8000 status lines. If a mismatch is detected, a trap is generated using the Segment Trap line ($\overline{\text{SEGT}}$).

The MMU functions constantly while memory is referenced, but its translation and protection tables are loaded and examined as an I/O peripheral. The Z8000's special I/O instructions (SIN, SOUT, and their variations -- see Section 3) can load or examine the MMU.

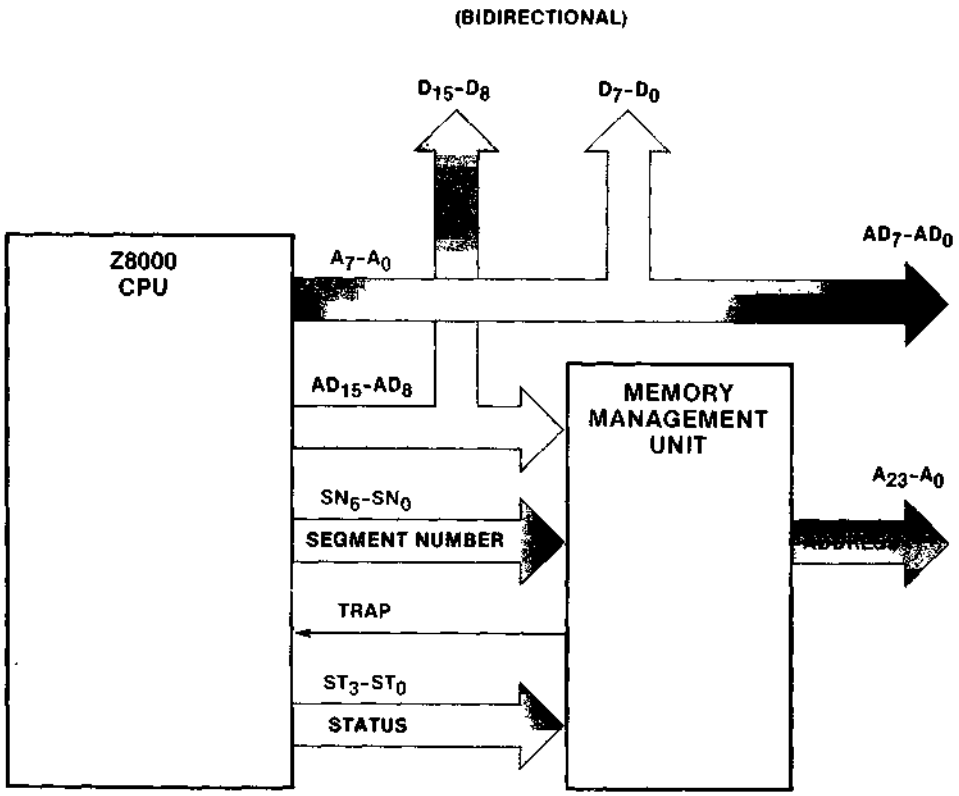


Figure 1-5 The MMU Connection

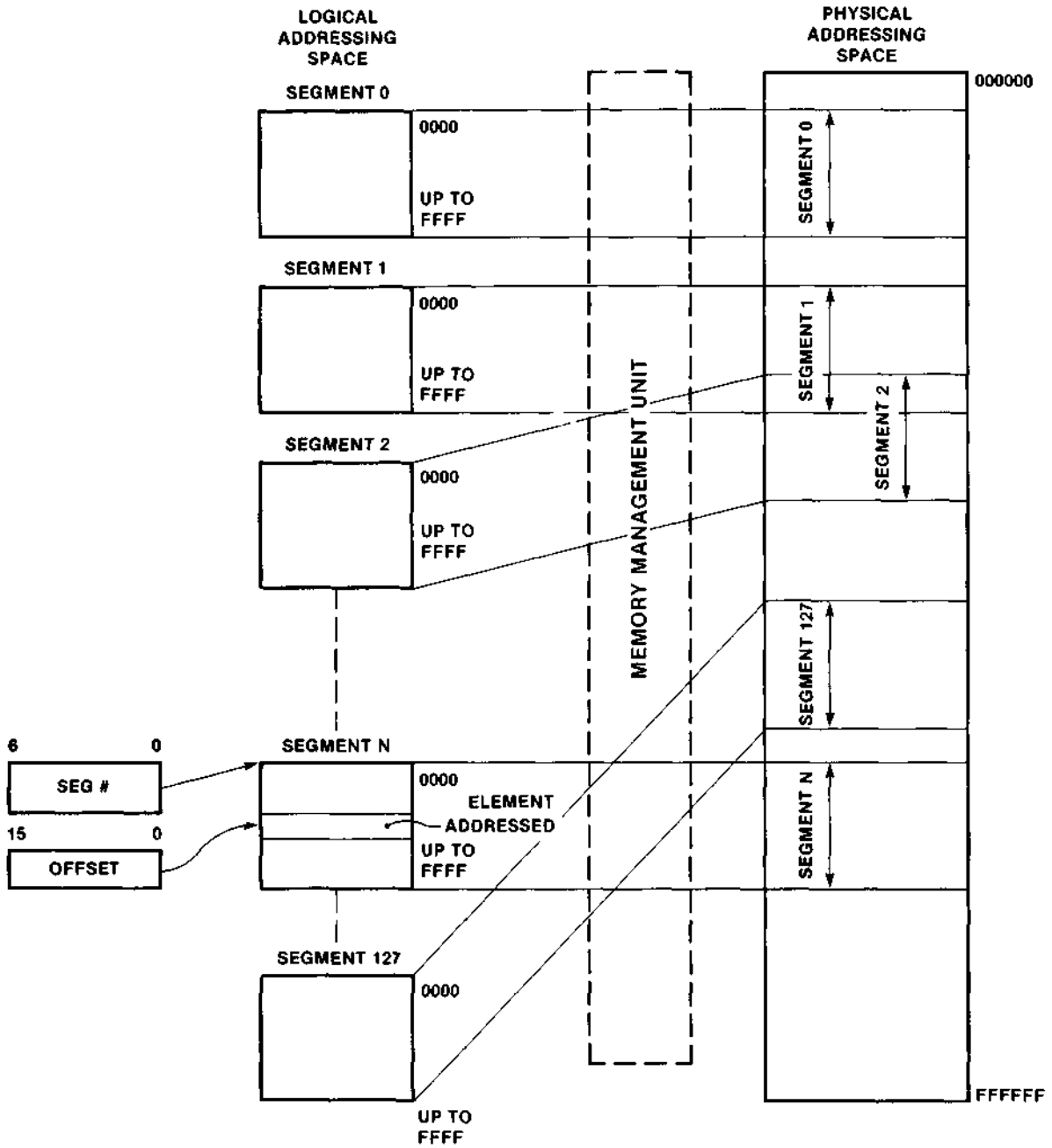


Figure 1-6 Logical to Physical Address Translation

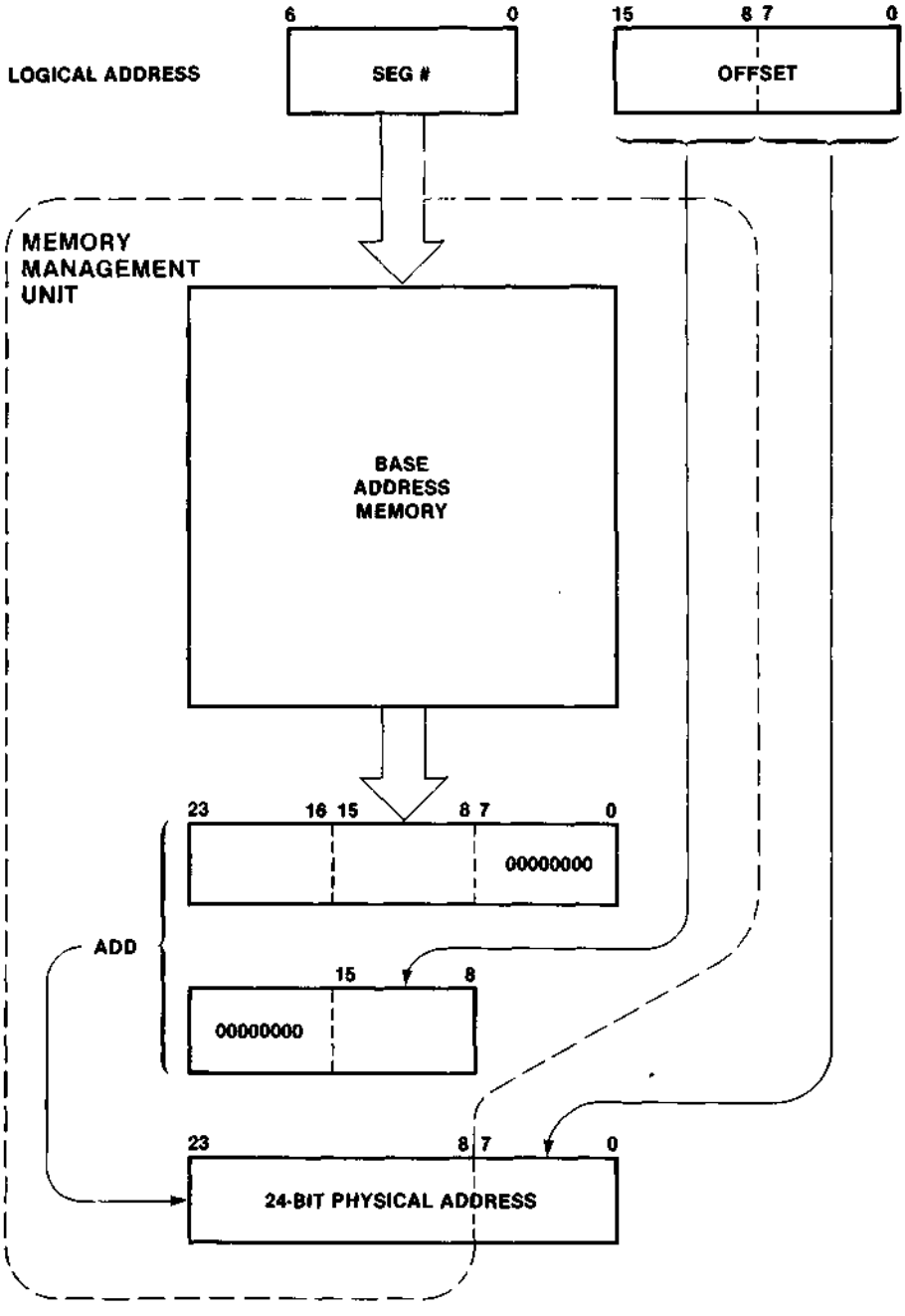


Figure 1-7 Segmented Address Relocation

1.5 DATA MANIPULATION

This section describes the Z8000 CPU data manipulation features which are relevant to the assembly-language programmer. For information on CPU architectural or hardware features, see the Z8000 CPU Technical Manual.

1.5.1 Data Types

Z8000 instructions allow the programmer to work with seven different data types (or data lengths); five of these were described in Section 1.2.2 and Figure 1-2:

- Bits
- 4-bit BCD digits
- 8-bit bytes
- 16-bit words
- 32-bit long words

Each of these can be addressed by specifying a byte address -- the byte containing a bit, digit or byte, or the high-order byte of a word or long word. For these five data types, the length to be used is implied by the operation.

All five of these data types can also be processed in CPU registers (Section 1.5.2). In this case, the instruction designates:

- a byte-register number (for a bit, digit, or byte data type),
- a word-register number (for a word data type), or
- a register-pair number (for a long word data type)

Z8000 instructions allow bits to be set, cleared, and tested. Digits are used in BCD arithmetic operations. Bytes are used to handle characters or small integers (in the range 0 to 255 if unsigned, or in the range -128 to 127 if signed). Words can contain larger values (in the range 0 to 65535 if unsigned, or in the range -32768 to 32767 if signed), instructions, or nonsegmented addresses. Long words contain large values (in the range 0 to 4,294,967,295 if unsigned, or in the range -2,147,483,648 to 2,147,483,647 if signed) and segmented addresses.

The two remaining data types, not discussed thus far are:

- Strings of bytes
- Strings of words

Strings can be stored and referenced only in memory. They are referenced by designating either their lowest or their highest byte address plus their length in bytes or words. They are used with Z8000 "autoincrement" or "autodecrement" instructions, which automatically increase or decrease the pointer to a byte or word address (Figure 1-8). Note that when strings are used, the programmer must designate the length of the data element explicitly, as one of the instruction operands.

1.5.2 General-Purpose Registers

The Z8000 CPU data manipulation capabilities are based on a powerful set of sixteen 16-bit, general-purpose registers which are used to hold temporary data and address calculations during the processing of data in the I/O and memory address spaces. The registers are specified in assembly language statements as R0 through R15, which means they can be addressed by four bits. All sixteen registers can be used as accumulators. In addition to their use as accumulators in arithmetic and logical operations, 15 of the 16 registers may be used in addressing mode calculations as either indirect, index or base-address registers. Because the instruction format encoding uses the value 0 to differentiate between various addressing modes, register R0 (or the register pair RR0) cannot be used as an indirect, index or base-address register.

The programmer can also address registers as groups of 8, 32, or even 64 bits (Figure 1-9). These registers are specified using the following assembly language symbols:

- RH0, RL0, RH1, RL1, ..., RH7, RL7 for 8-bit registers. ("H" stands for high-order byte, and "L" stands for low-order byte within a word register). These registers overlap 16-bit registers R0 - R7. All 8-bit registers can be used as accumulators.

- RR0, RR2, RR4, ..., RR14 for 32-bit register pairs.

- RQ0, RQ4, RQ8, and RQ12 for 64-bit register quadruples. These registers are used only by a few instructions such as Multiply, Divide, and Extend Sign.

1.5.3 Stacks and Stack Pointers

The Z8000 is a register-oriented machine. It also has sophisticated stack-oriented instructions and includes separate hardware-maintained stacks for its two operating modes -- a system stack and a normal stack, residing in system data memory and normal data memory, respectively. In addition, since any general-purpose register (except R0) can be used as a

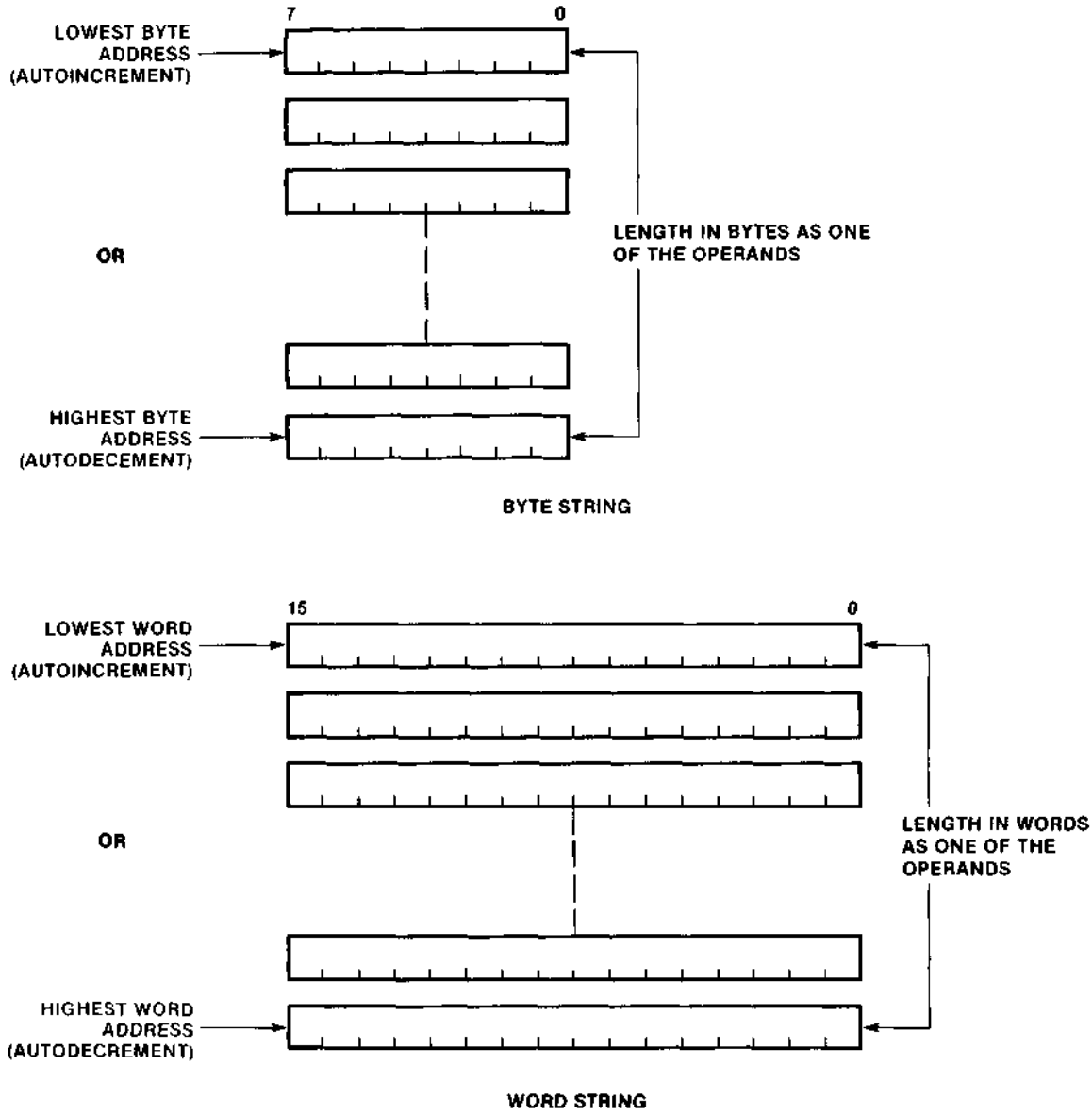


Figure 1-8 Byte and Word Strings

programmer-maintained stack pointer with instructions such as Push and Pop, multiple stacks can be efficiently supported. The stack pointers (pointers to stack locations in data memory) are kept in general-purpose registers and, therefore, can be altered using standard instructions. No special stack-pointer instructions are needed.

The processor stack is used implicitly by certain operations such as the Call and Return instructions or the interrupt and trap mechanisms. Depending on the System or Normal mode, these operations use a hardware-defined stack pointer to access either the system processor stack or the normal processor stack.

For the nonsegmented Z8000, the system and normal processor stacks can be addressed by 16 bits. Register R15 is used for this purpose. For the segmented version, 32-bit pointers are required and register pair RR14 is used (Figure 1-9). The format of the 32-bit address is shown in Figure 1-3.

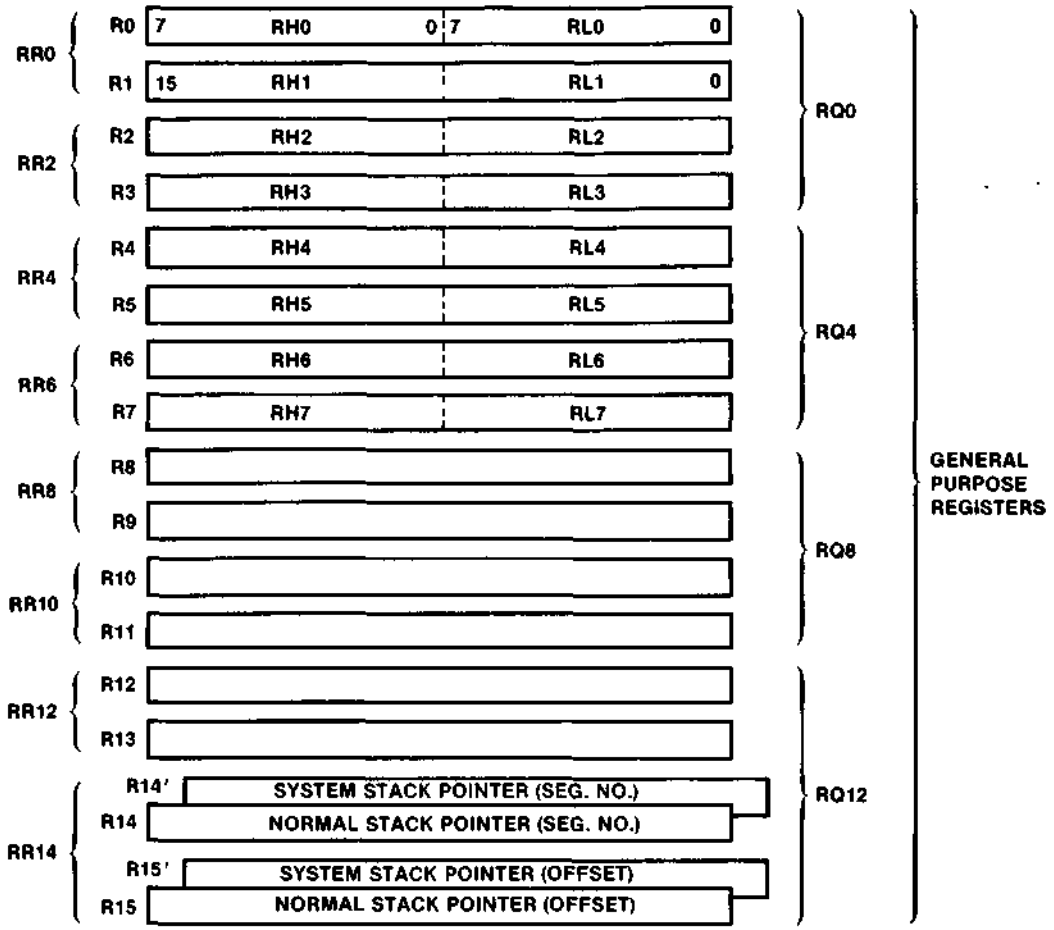
As Figure 1-9 also shows, two copies of the stack pointer are needed -- one for the system stack and one for the normal stack. Although the stacks are separated in memory, the normal stack pointer (NSP) register can be accessed while in System mode using the Load Control Register (LDCTL) instruction.

The two sets of stack pointers make task-switching easier. To make sure the normal stack is always free of system information, the normal program status data saved when an interrupt or trap occurs is pushed onto the system stack before the new program status is loaded (see Sections 1.6.4 and 1.6.5).

1.5.4 Addressing Modes

An assembly-language statement consists generally of an operation to be performed (instruction) and the data to be operated upon (operands). The latter include the source of the data and the destination where the result of the operation is to be stored.

In its simplest form, an operand can be either the specific data to be processed (immediate data) or the name of a register that holds the data. Data can be specified in more complex ways, also. For example, an operand may name a register whose contents are added to the contents of another register to form the address of the memory location containing the source data (based indexed addressing). Data can be specified by eight distinct addressing modes:



NOTE: Register R14 is not used as a stack pointer in the nonsegmented Z8000.

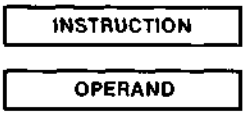
Figure 1-9 General-Purpose Registers

- Immediate Data (IM)
- Register (R)
- Indirect Register (IR)
- Direct Address (DA)
- Indexed Address (X)
- Relative Address (RA)
- Based Address (BA)
- Based Indexed Address (BX)

Depending on the operation, the addressing mode to be used might be implied by an instruction or spelled out explicitly. When stated explicitly, the addressing mode usually refers to a register or memory location. Implied addressing modes usually refer to program (code) memory or the I/O address space.

Immediate Data

Although considered an "addressing mode" for the purpose of this discussion, Immediate Data is the only mode that does not indicate a register or memory address. The data processed by the instruction in this case is the value supplied as the operand.

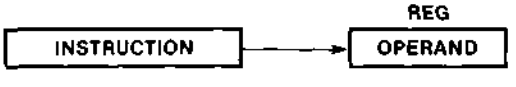


THE OPERAND VALUE IS IN THE INSTRUCTION

Immediate Data mode is often used to initialize registers. The Z8000 is optimized for this function, providing several short immediate data instructions to reduce the byte count of programs.

Register Addressing

In Register addressing mode, the instruction processes data taken from a specified general-purpose register. Storing data in a register allows shorter instructions and faster execution.

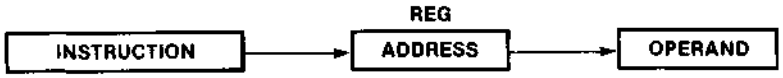


THE OPERAND VALUE IS THE CONTENTS OF THE REGISTER

The register length (byte, word, register pair, or register quadruple) is implied by the instruction.

Indirect Register Addressing

In Indirect Register addressing mode, the data processed is not the value in the specified register. Instead, the register holds the address of the data.



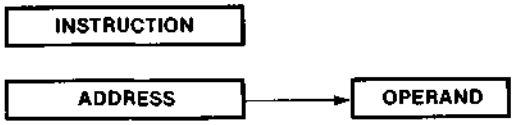
THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS IN THE REGISTER

A single word register is used to hold the address in Nonsegmented mode, while a register pair must be used in Segmented mode. Any general-purpose word register (or register pair in Segmented mode) can be used except R0 or RR0. This mode is also used by the I/O instructions to specify a "port" and always indicates a 16-bit I/O address held in a single word register.

The Indirect Register mode may save space and reduce run time when consecutive locations are referenced. This mode can also be used to simulate more complex addressing modes, since addresses can be computed before the data is accessed.

Direct Addressing

In Direct Address mode, the data processed is found at the address specified as an operand.

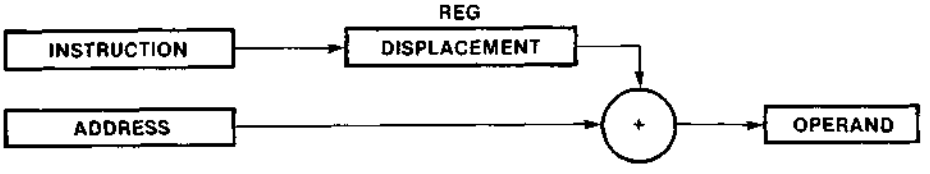


THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS IN THE INSTRUCTION

This mode is also used by the I/O instructions to specify a "port" and always indicates a 16-bit I/O address. This mode is also used by Jump and Call instructions to specify the address of the next instruction to be executed (actually, the address serves as an imediate value that is loaded into the program counter). The address specified is limited to one word in Nonsegmented mode and in Segmented mode using the short offset (Figure 1-4). Segmented addresses using a long offset require a long word.

Indexed Addressing

In Indexed Address mode, the instruction processes data located at an indexed address in memory. The indexed address is computed by adding the address specified in the instruction to a "displacement" or "index". The index is contained in a word register, also specified in the instruction. Indexed Addressing allows random access to tables or other complex data structures where the address of the base of the table is known, but the particular element index must be computed by the program.

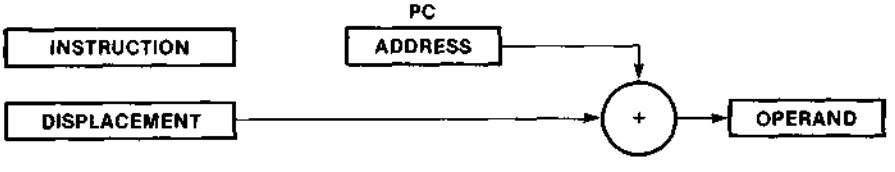


THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE ADDRESS IN THE INSTRUCTION, OFFSET BY THE CONTENTS OF THE REGISTER

Any word register can be used as the index register except R0. The address operand is limited to one word in Nonsegmented mode, but can be one or two words in Segmented mode, depending on whether a long or short offset is used in the address (Figure 1-4).

Relative Addressing

In Relative Address mode, the data processed is found at an address relative to the current instruction. The instruction specifies a two's complement displacement which is added to the program counter to form the target address. The program counter setting used is the address of the first instruction following the current relative instruction.

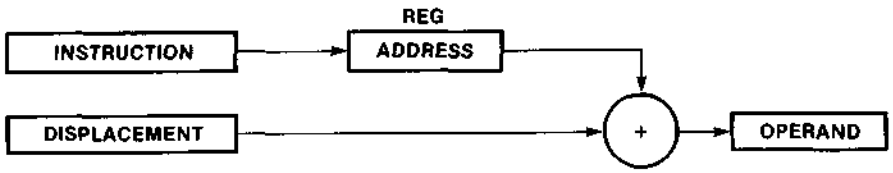


THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE CONTENTS OF PC OFFSET BY THE DISPLACEMENT IN THE INSTRUCTION

As with Direct Address mode, Relative Address is also used by certain program control instructions to specify the address of the next instruction to be executed (specifically, the result of the addition of the program counter and the displacement is loaded into the program counter).

Based Addressing

Based Address mode is similar to Indexed Address mode. In Based Address, however, the register operand specifies the base address and the displacement is expressed as a 16-bit value. The two are added and the resulting address points to the data to be processed. This addressing mode may only be used with the Load instructions. Based Address, as a complement to Indexed Address, allows random access to tables or other data structures where the displacement of an element within the structure is known, but the base of the particular structure must be computed by the program.

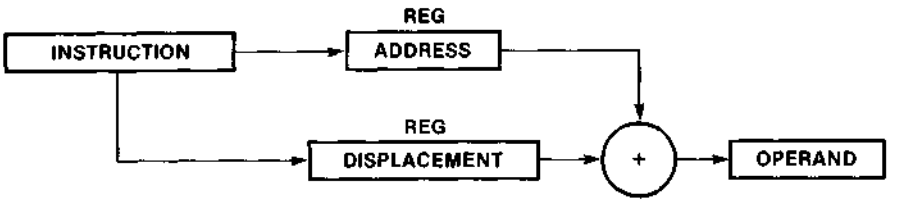


THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE ADDRESS IN THE REGISTER, OFFSET BY THE DISPLACEMENT IN THE INSTRUCTION.

Any word register (or register pair in Segmented mode) can be used for the base address except R0 or RR0. In Segmented mode, Based Address allows access to locations whose segment numbers are not known at assembly time.

Based Indexed Addressing

Based Indexed Address is an extension of Based Address, and may only be used with the Load instructions. In this case, both the base address and index (displacement) are held in registers.



THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE ADDRESS IN THE REGISTER, OFFSET BY THE DISPLACEMENT IN THE REGISTER.

Any word register (or register pair in segmented mode) can be used as the base address except R0 or RR0. Any word register can be used as the index except R0.

1.5.5 Instruction Formats

Z8000 instructions may occupy one to four words, depending on the number of operands and the number of words needed to form operand addresses or immediate values. Figure 1-10 shows some of the formats for typical instructions and the main fields within these formats.

Regardless of an instruction's length, its first byte contains the operation code (also referred to as the "opcode"). In addition to specifying the operation, this field also indicates the addressing mode (bits 14-15), and the word or byte data type (bit 8), as applicable. Instructions can designate zero or more operands explicitly. If the operands designate general-purpose registers, the register address(es) are usually specified in the second byte of the instruction. The format of these fields are detailed in Section 3.

1.6 PROGRAM CONTROLS

In addition to the general-purpose registers described in Section 1.5.2, the Z8000 CPU has several control registers containing status flags, control bits, the Program Counter, a pointer to the Program Status Area, and a memory refresh register.

The status flags, control bits, and program counter are referred to collectively as the Program Status (PS). The Program Status is contained in two or four words, depending on whether the nonsegmented or segmented version of the Z8000 is used (Figure 1-11). The status flags and control bits are referred to collectively as the Flags and Control Word (FCW).

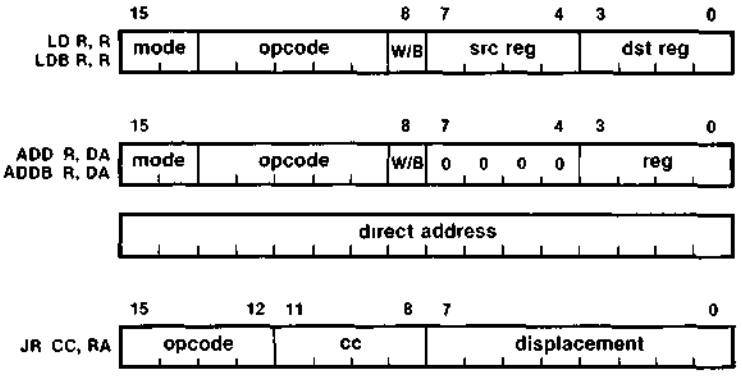


Figure 1-10 Typical Instruction Formats (Nonsegmented)

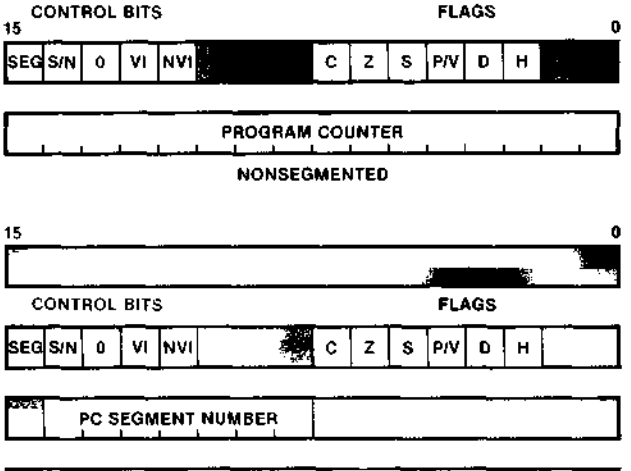


Figure 1-11 Program Status Blocks

1.6.1 Program Counter

The Program Counter (PC) contains the address in program memory of the next instruction to be executed. As shown in Figure 1-11, the Program Counter is 16 bits for the nonsegmented version and 32 bits for the segmented version. The segmented address format is described in Section 1.4.1.

1.6.2 Status Flags

Status Flags (FLAGS) can be used to determine the outcome of certain operations and to redirect the flow of the program as necessary. The program status has six flags for the use of the programmer and the Z8000 processor:

- Carry (C)
- Zero (Z)
- Sign (S)
- Parity/Overflow (P/V)
- Decimal Adjust (D)
- Half Carry (H)

Z8000 CPU control instructions allow the programmer to set, reset (clear), or complement any or all of the first four flags. The half-carry and decimal-adjust flags are used only by the Z8000 for BCD arithmetic corrections.

The FLAGS register can be separately loaded by the Load Control Register (LDCTLB) instruction without disturbing the control bits in the other byte of the FCW. The C, Z, S, and P/V flags can also be used with branching instructions to provide up to 21 conditional tests and as loop controls in string instructions.

The carry (C) flag, when set, generally indicates a carry out of the high-order bit position of a register being used as an accumulator. For example, adding two 8-bit numbers causes a carry out of bit 7 and sets the carry flag:

```
                    Bit
                  7 6 5 4 3 2 1 0

    225           1 1 1 0 0 0 0 1
+    64           0 1 0 0 0 0 0 0

    289      ┌─   0 0 1 0 0 0 0 1
             └─>  1 = carry flag
```

The zero (Z) flag is set when the result register's contents are zero following certain operations.

The sign (S) flag is set to one when the most significant bit of a result register contains a one (a negative number in two's-complement notation) following certain operations.

The overflow (V) flag, when set, indicates that a two's-complement number in a result register has exceeded the largest or is less than the smallest number that can be represented in a two's-complement notation. This flag is set as the result of an arithmetic operation. Consider the following example:

```
                    Bit
                    7 6 5 4 3 2 1 0

    120             0 1 1 1 1 0 0 0
+   105             0 1 1 0 1 0 0 1

    225         ┌─  1 1 1 0 0 0 0 1
                │
                └─> 0 = carry flag
```

The result in this case (-95 in two's complement notation) is incorrect, thus the overflow flag would be set.

The same bit acts as a parity (P) flag following logical instructions for byte operands only. The number of one bits in the register is counted and the flag is set if the total is even (that is, P=1). If the total is odd, the flag is reset (P=0).

The decimal-adjust (D) flag is used for BCD arithmetic. Since the algorithm for correcting BCD operations is different for addition and subtraction, this flag is used to specify what kind of instruction was executed so that the subsequent Decimal Adjust (DAB) instruction can perform its function correctly.

The half-carry (H) flag indicates a carry out of, or a borrow into bit 3 as the result of adding or subtracting two BCD digits. This flag is used by the DAB instruction to convert the binary result of a previous decimal addition or subtraction into the correct decimal (BCD) result.

Neither the decimal-adjust nor the half-carry flag is normally accessed by the programmer. The specific operations affecting the flags are detailed in Section 3 and listed in Appendix A.

1.6.3 Control Bits

The control bits are used to enable various interrupts or operating modes. The nonsegmented Z8000 uses three control bits; the segmented version has four (Figure 1-11). These bits are:

- Vectored Interrupt Enable (VI)
- Non-Vectored Interrupt Enable (NVI)
- Segmentation Mode (SEG), used only by the segmented Z8000
- System/Normal Mode (S/N)

The two interrupt control bits (VI and NVI) are set and reset by the Enable Interrupt (EI) and Disable Interrupt (DI) instructions. When the control bit is set to one, the appropriate interrupt is enabled; otherwise, it is disabled. Interrupts and interrupt handling are described further in the following two sections.

The Segmentation mode bit has meaning only for the segmented version of the Z8000. The setting of this bit indicates whether the Z8000 is running in Segmented (=1) or Nonsegmented mode (=0). The bit is set and reset by the LDPS or LDCTL instructions.

The System/Normal bit indicates the operating mode of the program. When set to one, System mode is in effect; when reset to zero, Normal mode is in effect. This bit can be explicitly set or reset by the LDPS or LDCTL instructions, or implicitly changed by the occurrence of an interrupt or trap (Section 1.6.4) when a new program status is loaded. The programmer can also enter System mode from Normal mode (and change the setting of this control bit) by issuing a System Call (SC) instruction.

Any control bit can be changed by the occurrence of an interrupt or trap, and then restored to its previous setting by terminating the interrupt handler procedure with an Interrupt Return instruction (IRET). The Interrupt Return pops the saved program status off the system stack.

1.6.4 Interrupts and Traps

Interrupts are asynchronous events and are typically triggered by peripheral devices needing attention. The three kinds of interrupts are:

- Non-Maskable interrupt (NMI)
- Vectored interrupt (VI)
- Non-Vectored interrupt (NVI)

Non-Maskable interrupts cannot be disabled, and are usually reserved for critical external events that require immediate attention. Vectored interrupts cause a 16-bit value output by the interrupting device to be read from the data bus (AD0-AD15). This "vector" value is used to select a particular interrupt procedure to automatically branch to (Section 1.6.5). Non-Vectored interrupts are all handled by the same interrupt procedure, which may "poll" the external devices to determine which one requires attention.

Traps, on the other hand, are synchronous events and usually indicate some special programming error or condition. They are triggered by specific instructions and recur each time the instruction is executed with the same set of data. The four kinds of traps are:

- Unimplemented instructions
- Privileged instructions in Normal mode
- Segmentation violations
- System call

During the execution of an instruction, one of three kinds of error conditions can arise. An "illegal instruction exception" signifies that that the binary code of the current instruction is an illegal value for the architecture of the Z8000. The result of this error is undefined, so the programmer must not use binary instruction values other than those defined in the instruction set of Section 3.

An "unimplemented instruction exception" signifies that the binary code of the current instruction is defined by the Z8000 architecture, but is not currently implemented by the hardware (Section 3.4). In this case, an Unimplemented Instruction trap occurs, which allows system software to either simulate the execution of the instruction or abort the program.

An "operation exception" signifies that the binary code of the current instruction is valid, but the operand specification or execution of the instruction is architecturally invalid. For example, an attempt to execute a privileged instruction in Normal mode will cause a trap. A segmentation violation, such as using an offset larger than the defined length of the segment, will cause the MMU to signal a Segmentation trap. (Segmentation traps occur only with the segmented Z8000.) All other operation exceptions, such as specifying an odd address for a word data value, are considered programmer errors and the results are undefined.

The System Call instruction (SC) provides a controlled access from Normal mode software to System mode software, and is handled in a manner similar to the other traps.

Because it is possible for several interrupts or traps to occur simultaneously, a priority for selecting which kind of interrupt or trap is honored first is established by the Z8000 architecture. The descending priority order of the traps and interrupts is: internal trap (unimplemented instructions, privileged instructions and the System Call instruction), Non-Maskable interrupt, Segmentation trap, Vectored interrupt, and Non-Vectored interrupt.

Interrupts and traps are handled similarly by the Z8000 hardware. At the start of the interrupt or trap sequence, the Z8000 is forced into System mode. In addition, the segmented version of the Z8000 is forced into Segmentation mode, regardless of the current mode. The program status information in effect just prior to the interrupt or trap is pushed onto the system stack. An additional word, indicating the "identifier" for the interrupt or trap, is also pushed onto the system stack, where it can be accessed by the interrupt or trap handler (Figure 1-12). The new program status is set up as described in the following section (which includes loading the program counter with the starting address of the procedure for servicing the interrupt or trap), and then control is transferred to the service procedure.

For internal traps, the "identifier" stored in the system stack is the first word of the instruction causing the trap. For interrupts and external traps (such as a segmentation trap), the "identifier" stored is the 16-bit value read from the data bus (AD0-AD15) at the start of the interrupt acknowledge sequence. In the case of Vectored interrupts, this value is also used to select the appropriate service procedure, as explained in the following section.

1.6.5 Program Status Area

As part of a system software configuration, the user must provide service procedures to handle the various interrupts and traps. The procedures are accessed through their respective program status blocks (Figure 1-11) which determine the new program status set up when program execution is interrupted. These status words reside in a reserved area of memory called the Program Status Area, also established by the programmer, and should be arranged as shown in Figure 1-13. The ordering is important, because the specific program status block selected (and consequently the service procedure selected) is determined implicitly from the kind of interrupt or trap that occurred. Note that the size of each program status block depends on the version of the Z8000 (two words for the nonsegmented and four words for the segmented version).

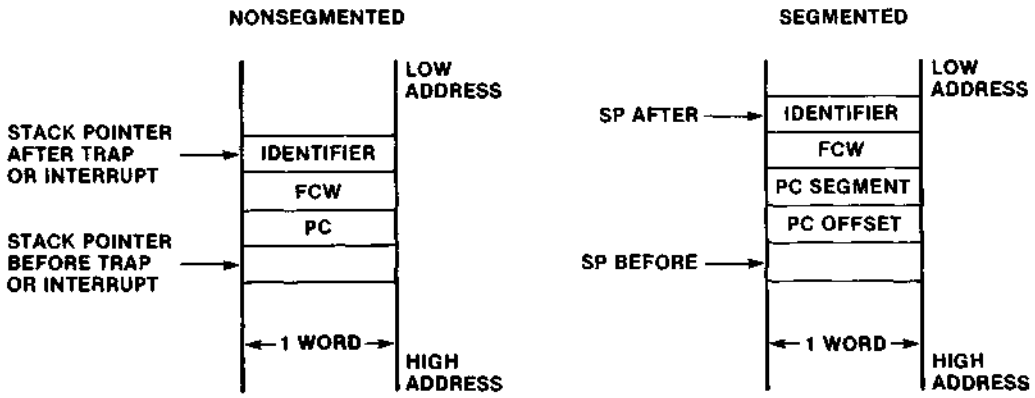
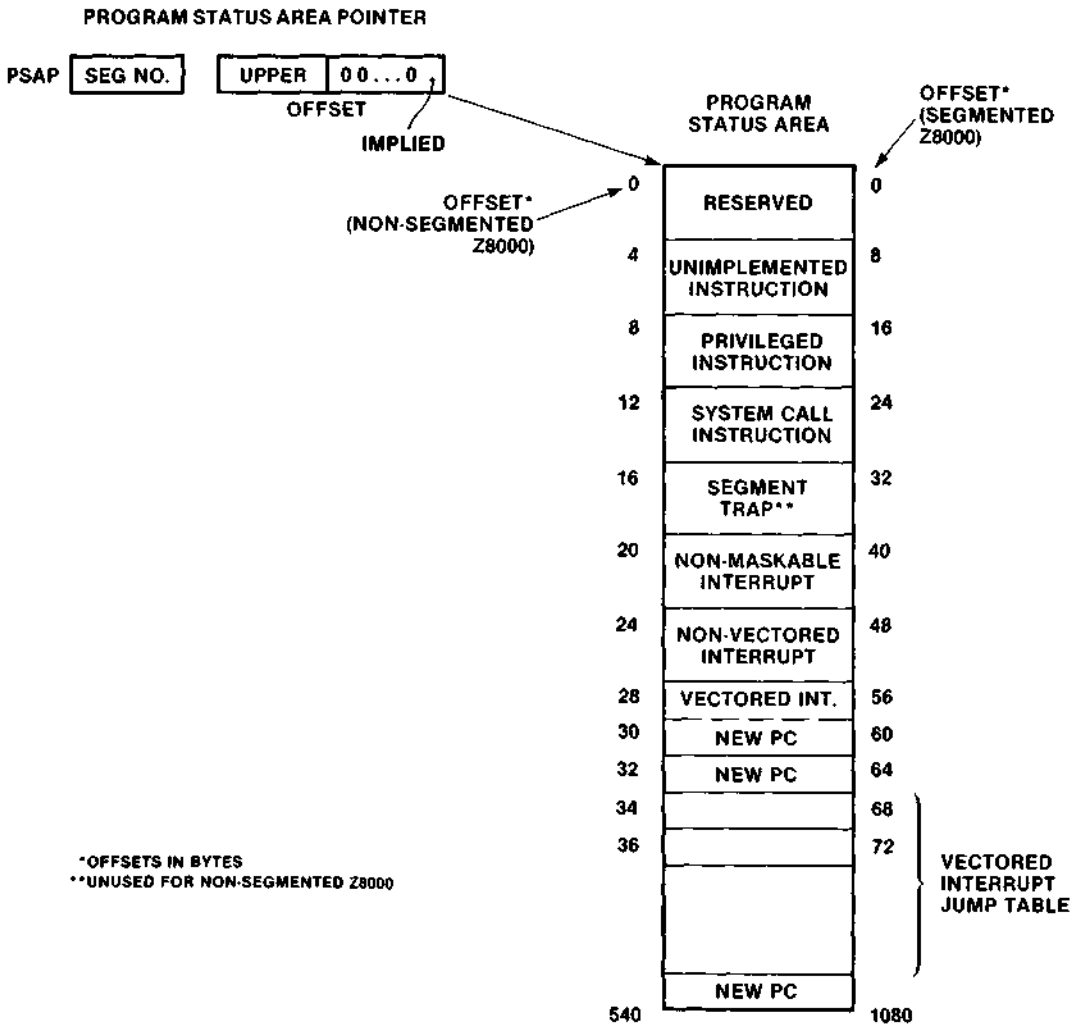


Figure 1-12 Format of Saved Program Status in the System Stack



SEG/NSEG with MODE
same Vers - 1bh.

Figure 1-13 Program Status Area

For each kind of interrupt or trap other than a Vectored interrupt, there is a single program status block that is automatically loaded into the Program Status registers (which includes the Flags and Control Word and the Program Counter). Control is then passed to the service procedure whose starting address is contained in the Program Counter.

For all Vectored interrupts, the same Flags and Control Word (FCW) is loaded from the corresponding program status block. However, the appropriate Program Counter (PC) value is selected from up to 256 different values in the Program Status Area. The low-order eight bits of the "identifier" placed on the data bus by the interrupting device is used as an index into the Program Status Area following the FCW for Vectored interrupts. The "identifier" value 0 selects the first PC value, the value 1 selects the second PC, and so on up to the "identifier" value 255.

The Program Status Area is addressed by a special control register, the "Program Status Area Pointer", or PSAP. This pointer is one word for the nonsegmented and two words for the segmented Z8000. As shown in Figure 1-14, the pointer contains a segment number (if applicable) and the high-order byte of a 16-bit offset address. The low-order byte is assumed to contain zeroes, thus the Program Status Area must start on a 256-byte address boundary. The programmer accesses the PSAP using the Load Control Register instruction (LDCTL).

1.6.6 System Reset

A system reset overrides all other conditions, including all other interrupts or traps. When a reset sequence is begun, a 4-word program status is fetched from segment 0, offset 0 for the segmented Z8000; for the nonsegmented version, the program status occupies two words at address 2.

During the reset sequence, the status pins and other outputs indicate System mode; in addition, Segmentation mode is in effect for the segmented Z8000. No information is saved on the system stack when a reset occurs since the stack pointer has not yet been initialized.

1.6.7 Memory Refresh

The refresh control register (REFRESH) is a 16-bit counter used to refresh dynamic memory automatically. A special refresh memory access is made at progammable intervals and is otherwise invisible to the currently executing program. This register has the format shown in Figure 1-15 and, like the other control registers, can be programmed using the Load Control Register (LDCTL) instruction.

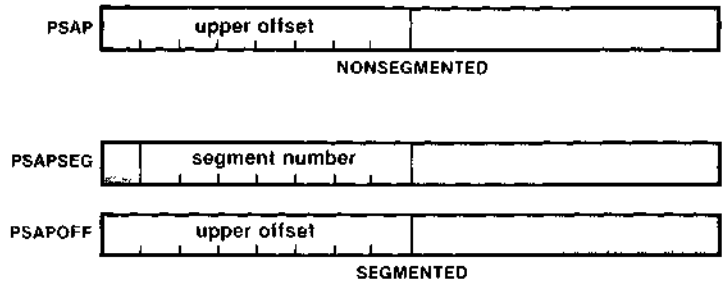


Figure 1-14 Program Status Area Pointer

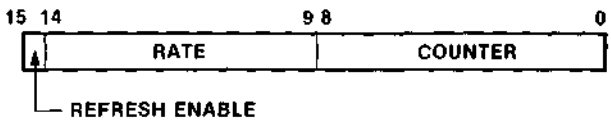


Figure 1-15 Refresh Register

The refresh rate (the time between successive refreshes) is determined by the 6-bit prescaler. This is a modulo-n counter (n = 1-64) driven at one-fourth the system clock rate. Thus, with a 4 MHz clock, the refresh period is programmable from 1 to 64 microseconds.

The refresh row counter is 9 bits and is incremented by 2 each time the prescaler times out (reaches zero). This allows up to 256 rows for future high-density memories. (Currently available 16K dynamic RAMs have 128 rows.)

Memory refresh can be totally disabled, if necessary, by setting bit 15 of the refresh register to zero.

1.7 ADDRESS ARITHMETIC

1.7.1 Nonsegmented Addressing

In the nonsegmented Z8000, all addresses are 16-bit values. When addressing mode arithmetic is performed, the result is always taken modulo 65536. No carry is generated or any other indication that the result of addition or other addressing operations may have overflowed 16 bits. Thus in Indexed, Relative, Based and Based Indexed addressing modes, or in autoincrement and autodecrement instructions, the resulting address always remains within the 0 to 65535 addressing space.

1.7.2 Segmented Addressing

In the segmented Z8000, all memory addresses are 23-bit values, consisting of a 7-bit segment number and a 16-bit offset. The short offset format (Figure 1-4) contains an 8-bit offset which is extended to include 8 high-order zero bits before further addressing computations are performed. When addressing mode arithmetic is performed, only the 16-bit offset is used so that the result is taken modulo 65536 as in nonsegmented addressing. No carry out of the 16-bit offset is generated, nor is any other indication that the result of addition or other addressing operations may have overflowed 16 bits. In other words, the segment number is not affected by addressing arithmetic. Thus in Indexed, Relative, Based and Based Indexed addressing modes, or in autoincrement or autodecrement instructions, the resulting address always remains within the same segment with the offset in the range 0 to 65535.

Nonsegmented programs can be run in one segment of the segmented Z8000 by setting the Segmentation mode control bit to zero with the LDPS or LDCTL instructions. In this case, the current value of the Program Counter's segment number is used for all memory accesses. While "running the segmented Z8000 in Nonsegmented mode", the rules of nonsegmented addressing mode arithmetic apply.

All I/O addresses are 16-bit values, regardless of whether memory addresses are segmented or nonsegmented. Therefore, all I/O address arithmetic follows the rule for nonsegmented addressing.

Section 2
Z8000 Assembler Conventions

2.1 ASSEMBLER OVERVIEW

The Z8000 microprocessor is programmed in a symbolic assembly language (PLZ/ASM). This marks a significant improvement over coding in binary notation. The operation codes for assembly-language statements are easily memorized (for example, SUB for Subtract and LD for Load). In addition, meaningful symbolic names can be assigned to program addresses and data (for example, ALLOCATE as the label of the first statement in a storage allocation procedure).

A Z8000 source module is made up, for the most part, of such assembly language statements. These statements are then translated by the Z8000 assembler into an object module that can either be separately executed by the Z8000 microprocessor, or can be linked with other object modules to form a complete program. Because the assembler has some high-level features, a source module can also include PLZ constructs such as DO and IF statements. The user can also embed assembler directives, which control the operation of the assembler, in the source module. High-level statements and assembler directives are discussed in Sections 4 and 5.

Depending on the assembler directives used, addresses within an object module or program can be absolute (meaning addresses in the source program correspond exactly to Z8000 logical memory addresses) or relocatable (meaning addresses can be assigned relative to some logical base address at a later time). Object modules should be made relocatable wherever possible. This facilitates both the ability to link with other object modules as well as the ability to load object programs anywhere in memory. It also allows the creation of libraries of commonly used procedures (including math or input/output routines) that can be linked selectively into several programs as desired.

Operation of the assembler, module linkage, address relocation, and program execution are the subject of the Z8000 Assembler User's Guide.

2.2 ASSEMBLY LANGUAGE STATEMENT FORMAT

The most fundamental component of a PLZ/ASM program is the assembly language statement consisting of an instruction and its operand(s). The instruction describes an action to be taken; the operand(s) supplies the data to be acted upon.

An assembly language statement can include four fields:

- Statement label(s)
- An instruction
- Operand(s)
- Comments

The statement label and comment fields are always optional. The statement has zero or more operands, depending on the instruction selected. The following statements have the same effect in a Z8000 program, but the second is more descriptive (and consequently more helpful in program debugging).

```
Label        Instruction   Operand(s)      Comment

             LD            COUNT, #255
INITCOUNT:   LD            COUNT, #255     !Load COUNT with
                                            initial value!
```

Each of the elements of a PLZ/ASM program must be separated from other elements by one or more delimiters. A delimiter is one of the characters: space (blank), comma, semicolon, tab, carriage return, line feed, or form feed. Note that carriage return is treated just like any other delimiter, so that a single statement may span several lines, or several statements may appear on a single line. The delimiter used in a specific situation is up to the programmer. For the sake of illustration, this manual uses blanks to separate statement fields and commas to separate operands.

2.2.1 Program Labels and Identifiers

Any assembly language (or high-level) statement in a Z8000 program can be preceded by any number of labels. Any statement referenced by another statement must be labeled. A label consists of an identifier followed by a colon (:) in the form:

```
label1:  label2: ... labeln:  statement
```

A PLZ/ASM identifier can contain up to 127 characters, of which the first must be a letter. The remaining characters can be letters, digits, or the special character underscore (_). Letters can be capitalized or lower-cased, but each time an identifier is used, it must be written in exactly the same way. The following are valid identifiers:

```
START_UP_ROUTINE
Program_Initialization
A
Loop_12
N1
sort
```

In addition to their statement-labeling function, identifiers also serve as symbolic names for constants (Section 2.3.2), data variables (Section 2.3.3), and procedures (Section 5.2.2). Certain identifiers serve as PLZ/ASM keywords and should not be used as programmer-defined identifiers (see Appendix D).

An identifier can be associated with only one item within the scope of its definition. Section 4.2.5 explains the scope of identifiers, including the scope of labels. Labels are accessible within the module in which they are defined, and are not accessible outside that module unless specifically declared to be GLOBAL or EXTERNAL.

2.2.2 Instruction

The instruction is the assembly-language mnemonic describing a specific action to be taken. The instruction must be separated from its operand(s) by a delimiter.

```
LD    R5, R10        !Load register 5 from register 10!

CLR   R10            !Clear register 10!
```

Many of the operations of the Z8000 can be applied to word, byte, and long operands. A simple naming convention has been adopted to distinguish the size of the operands for these particular instructions: the suffix "B" designates a byte instruction, the suffix "L" designates a long word instruction, and no suffix designates a word instruction:

```
ADD     R0, R1          !Add word operands!

ADDB    RH0, RL0        !Add byte operands!

ADDL    RR0, RR2        !Add long operands!
```

2.2.3 Operand Field

Depending on the instruction specified, this field can have zero or more operands. If two or more operands are needed, each must be separated by a delimiter.

```
IRET                            !No operand!

COM     R10                     !One operand!

ADD     R6, #210                !Two operands!

LDM     R2, SAVEREG, #5         !Three operands!

CPD     R2, @R6, R1, Z          !Four operands!
```

Operands supply the information the instruction needs to carry out its action. An operand can be:

- Data to be processed (immediate data);
- The address of a location from which data is to be taken (source address);
- The address of a location where data is to be put (destination address);
- The address of a program location to which program control is to be passed;
- A condition code, used to direct the flow of program control.

Although there are a number of valid combinations of operands, there is one basic convention to remember: the destination operand always precedes the source operand. Refer to the specific instructions in Section 3 for valid operand combinations.

Immediate data can be in the form of a constant, an address, or an expression (constants and/or addresses combined by operators). Each of these forms is described in Section 2.3.

```
LD      R0, #K              !Load constant K into reg 0!

LD      R0, #COUNTER        !Load address of COUNTER into
                             reg 0!

ADD     R0, #CON/3 + 5      !Add value of expression
                             (CON/3 + 5) to contents of
                             reg 0!
```

Source, destination, and program addresses can also take several forms. PLZ/ASM addressing modes are described in Section 2.4. Some examples are:

```
LD    R0, @R5               !Load word value whose
                             address is in register 5
                             into register 0!

LDB   RH5, VAR1             !Load byte value located at
                             address labeled VAR1 into
                             register RH5!

LDL   RR10, VAR1 + 1        !Load long value at location
                             following that addressed by
                             VAR1 into register pair 10-11!

JP    Z, LOOP1              !Jump to program address
                             labeled LOOP1 if zero flag (Z)
                             is set!

JP    NZ, LOOP1 + 6         !Otherwise, jump to location
                             six bytes after LOOP1!
```

Condition codes are listed in Section 3.2.1.

2.2.4 Comments

Comments are used to document program code as a guide to program logic and also to simplify present or future program debugging. Comments can be inserted anywhere a program delimiter may appear. Comments are bounded by exclamation points (!) and can contain any characters except the exclamation point itself.

```
!Module 3, Changed 7-25-78!

RES     R15, #3     !?!
```

A single comment can cross line boundaries; that is, carriage returns can occur within a comment.

2.3 ARITHMETIC OPERANDS

2.3.1 Run-Time vs. Assembly-Time Arithmetic

Arithmetic is performed in two ways in an assembly language program. Run-time arithmetic is done while the program is actually executing.

```
SUB  R10, R12      !Subtract the contents of register
                    12 from the contents of register 10!
```

Assembly-time arithmetic is done by the assembler when the program is assembled and involves the evaluation of arithmetic expressions in operands, such as the following:

```
LD  R0, #(22/7 + X)

JP  Z, LOOP1 + 12

ADD R2, HOLDREG-1
```

Assembly-time arithmetic is more limited than run-time arithmetic in such areas as signed vs. unsigned arithmetic and the range of values permitted.

Only unsigned arithmetic is allowed in assembly-time expression evaluation. Run-time arithmetic uses both signed and unsigned modes, as determined from the assembly-language instruction specified and the meaning attached to operands by the programmer.

All assembly-time arithmetic is computed using 32-bit arithmetic, "modulo 4,294,967,296" (2 raised to the thirty-second power). Values greater than or equal to 4,294,967,296 are divided by 4,294,967,296 and the remainder of the division is used as the result. Depending on the number of bits required by the particular instruction, only the rightmost 4, 8, 16, or 32 bits of the resulting 32-bit value are used. If the result of assembly-time arithmetic is to be stored in four bits, the value is taken "modulo 16" to give a result in the range 0 to 15. If the result is to be stored in a single byte location, the value is taken "modulo 256" to give a result in the range 0 to 255 (or -128 to 127 if signed representation is intended). If the result is to be stored in a word, the value is taken "modulo 65536" to give a result in the range 0 to 65535 (or -32768 to 32767 if signed representation is intended).

```
LDB    RL4, #X+22          !Result of (X+22) must be in
                            range -128 to 255!

JP     X+22                !Modulo 65536.  Result is the
                            address 22 bytes beyond X!

ADDL   RR12, #32000*MAX    !Result of (32000*MAX) is taken
                            modulo 4,294,967,296!
```

2.3.2 Constants

A constant value is one that doesn't change throughout the program. Constants may be expressed as numbers, as character sequences, or as a symbolic name representing a constant value.

Numbers can be written in decimal, hexadecimal, binary, or octal notation. The latter three are preceded by a percent sign (%) and, in the case of binary and octal, by a base specifier enclosed in parentheses. If a number has no prefix, decimal is assumed.

```
 10              decimal
%10              hexadecimal
%AF0F            hexadecimal
%(2)10110010     binary
%(8)70           octal
```

A character sequence is a sequence of one or more characters enclosed in single quote marks. Any ASCII character (except a percent sign or single quote) can be included in the character sequence.

```
'A'
'This is a character sequence'
```

A character can also be represented in a character sequence in the form "%hh," where "hh" is the hexadecimal equivalent of the ASCII code for the character. (See Appendix E for the ASCII character set and its hexadecimal equivalents.)

```
'Here is an ESC character:  %1B'
```

For convenience, certain ASCII characters have been assigned shorter, more mnemonic codes as follows:

```
%L  or  %l      Linefeed
%T  or  %t      Tab
%R  or  %r      Carriage Return
%P  or  %p      Page (Form Feed)
%%              Percent Sign
%Q  or  %q      Single Quote
```

Example:

```
'First line%rSecond line%r'
'Quote%Qinside a quote%Q'
```

A constant can be assigned a symbolic name by a constant definition (CONSTANT) statement. A symbolic identifier, once associated with a constant value, retains that value through the entire program module.

Constant symbols are defined by the CONSTANT statement in the form shown below. Identifiers follow the rules outlined in Section 2.2.1. The special character pair ":=" can be read "is defined as".

```
CONSTANT
   REC_LENGTH              := 64
   BUFFER_LENGTH           := 4*RECLENGTH
   SEMICOLON               := ';'
   BIGNUMBER               := 1000000
   smallnumber             := -1
```

It is also possible to create a new symbol which will be treated the same as any reserved keyword (see Appendix D for a list of reserved keywords). If the symbol on the right side of the ':=' in a CONSTANT statement is a keyword, then the symbol on the left side can be used thereafter any place the keyword would be valid. One important use of this ability is to "rename" a register such as R5 with a more meaningful name such as SUBTOTAL (of course, the symbol R5 can still be used). The programmer is cautioned, however, that renaming keywords in general can lead to confusing and difficult to maintain programs.

2.3.3 Data Variables

A data variable can be thought of as a "container" that can hold different values from time to time. Just as a physical 8-ounce container can hold 0-8 ounces of liquid, an 8-bit (BYTE or SHORT_INTEGER) variable can hold values in the range 0 to 255 if unsigned, or -128 to 127 if signed two's complement representation is intended. A 16-bit (WORD or INTEGER) variable can hold values in the range 0 to 65535 if unsigned, or -32768 to 32767 if signed. Similarly, a 32-bit (LONG or LONG_INTEGER) variable can hold values in the range 0 to 4,296,967,296 (or -2,148,483,648 to 2,148,483,647 if signed two's complement representation is intended).

NOTE

While it is suggested that BYTE, WORD, and LONG variables be used for unsigned values, and SHORT_INTEGER, INTEGER, and LONG_INTEGER variables be used for signed values, there are no restrictions on whether a particular variable is signed or unsigned. In other words, BYTE and SHORT_INTEGER are treated as equivalent, as are WORD and INTEGER, as are LONG and LONG_INTEGER, with the appropriate interpretation left entirely to the programmer.

A data variable name can be associated with a data memory location; the value of the variable is the contents of that location at the time the variable is referenced. A data variable name is a symbolic identifier and follows the rules for identifiers in Section 2.2.1.

```
LD    R5, MPLIER          !Load the value contained in the
                           location symbolized by MPLIER!

ADD   R5, 4 + SUBTOTAL    !Add the value contained in the
                           location 4 bytes after the loca-
                           tion addressed by SUBTOTAL to
                           the contents of register 5!
```

If a data variable operand is preceded by "#," it is treated as immediate data and the value used is the data address associated with the variable, not the contents of the location. For example, suppose location 50 has the symbolic name COUNTER and contains the bit pattern 11111111 (decimal 255).

```
LDB   RL0,  COUNTER          !255 is loaded into RL0!

LDB   RL0, #COUNTER          !50 is loaded into RL0!

LDB   RL0,  COUNTER - 5      !Contents of location 45 are
                              loaded into RL0!

LDB   RL0, #COUNTER - 5      !45 is loaded into RL0!
```

Every data variable name has a type and scope associated with it, as well as a value. The type and scope (and, optionally, the initial value) are defined in a variable declaration statement like the following:

```
INTERNAL
    SWITCH1  BYTE
```

In this example, "INTERNAL" is the scope of the variable SWITCH1, and "BYTE" is its type.

Variables can have GLOBAL, EXTERNAL, INTERNAL, or LOCAL scope. They can be one of the "simple" types BYTE or SHORT_INTEGER (for 8-bit values), WORD or INTEGER (for 16-bit values), or LONG or LONG_INTEGER (for 32-bit values). They can also be one of the "structured" types ARRAY or RECORD. (Section 5.3.4 discusses variable declaration in more detail.)

2.3.4 Expressions and Operators

Expressions are formed using arithmetic, logical, shift, and relational operators in combination with constants and variables. These operators allow both unary (single-operand) and binary (two-operand) expressions, as shown below.

Arithmetic Operators. The arithmetic operators are as follows:

Operator	Operation
+	Unary plus, binary addition
-	Unary minus, binary subtraction
*	Unsigned multiplication
/	Unsigned division
MOD	Unsigned modulus

The division operator (/) truncates any remainder. The MOD operator returns the remainder from dividing its operands.

```
17/4 = 4

17 MOD 4 = 1
```

If zero is specified as the right operand for either of these division operators, the result is undefined.

Examples:

```
ADD  R5, #-3            !A minus 3 is added to
                         register 5!

ADD  R5, #K + (5*3)     !Value of constant K + 15 is
                         added to register 5!
```

Note that expressions containing these operators are evaluated at assembly time and, consequently, the arithmetic performed is unsigned. Signed arithmetic can still be done at run time.

Logical Operators. The logical operators are as follows:

Operator	Operation
LNOT	(Unary) Logical complement
LAND	Logical AND
LOR	Logical OR
LXOR	Logical EXCLUSIVE OR

LNOT simply complements the bit pattern of its (single) operand. All one bits are changed to zero and vice-versa.

```
LD    R2, #LNOT MASK    !Reverse the bits in a
                         mask and load into reg 2!
```

The effect of LAND, LOR, and LXOR can be seen from the following examples. Although 32-bit arithmetic would actually be done by the assembler, 8-bit arithmetic is shown for clarity. Assume two constants A and B have the bit patterns 11110000 and 01010101, respectively. The expressions:

```
A LAND B
A LOR B
A LXOR B
```

will result in the following evaluations of the operands:

```
LAND  11110000     LOR  11110000     LXOR  11110000
      01010101          01010101           01010101

      01010000          11110101           10100101
```

LAND sets a one bit whenever both ANDed bits are one; LOR sets a one bit whenever either ORed bit is one; LXOR sets a one bit when the two EXCLUSIVE-ORed bits are different.

The assembly-time logical operations performed by LNOT, LAND, LOR, and LXOR can also be done at run time by the Z8000 instructions COM, AND, OR, and XOR. The assembly-time operations require less code and register manipulation. The run-time operations allow greater flexibility, however. For example, they can operate on registers (variables) whose contents are not known at assembly time, as well as on known constant values.

Shift Operators. The shift operators are as follows:

SHR	Logical shift right
SHL	Logical shift left

When used in expressions, the shift operators have the form

```
d operator n
```

where "d" is the data to be shifted and "n" specifies the number of bits to be shifted. Vacated bits are replaced with zeros. For example, if the constant PRODUCT is equal to 10110011, the statement

```
LDB    RL0, #(PRODUCT SHL 2)
```

would load the value 11001100 into register RL0.

If the second operand supplied is negative (that is, if the sign bit is set), it has the effect of reversing the direction of the shift.

```
ADD    PRODUCT, #(MPLIER SHR -1)    !MPLIER is shifted left
                                     one bit position!
```

Relational Operators. The relational operators are as follows:

<	Less than
<=	Less than or equal
=	Equal
<>	Not equal
>=	Greater than or equal
>	Greater than

These six relational operators return a logical TRUE value (all ones) if the comparison of the two operands is true, and return a logical FALSE value (all zeros) otherwise. The operators assume both operands are unsigned.

```
LD   R0,#(1=2)          !Reg 0 is loaded with zeros!
LD   R0,#(2+2) < 5      !Reg 0 is loaded with ones!
```

Precedence of Operators. Expressions are generally evaluated left to right with operators having the highest precedence evaluated first. If two operators have equal precedence, the leftmost is evaluated first.

The following lists the PLZ/ASM operators in order of precedence:

- Unary operators: +, -, LNOT
- Multiplication/Division/Shift/AND: *, / , MOD, SHR, SHL, LAND
- Addition/Subtraction/OR/XOR: +, -, LOR, LXOR
- Relational operators: <, <=, =, <>, >=, >

Parentheses can be used to change the normal order of precedence. Items enclosed in parentheses are evaluated first. If parentheses are nested, the innermost are evaluated first.

20/5 - 12/3 = 0

20/(5 - 12/3) = 20

Modes of Arithmetic Expressions. All arithmetic expressions have a mode associated with them: absolute, relocatable, or external. These modes are defined in detail in Section 4, following the explanation of the concepts of "scope" and "program relocatability".

2.3.5 Segmented Address Operators

Two special operators are provided to ease the manipulation of segmented addresses. While addresses can be treated as a single value with a symbolic name assigned by the programmer, occasionally it is useful to determine the segment number or offset associated with a symbolic name.

The "SEG" unary operator is applied to an address expression which contains a symbolic name associated with an address, and returns a 16-bit value. This value is the 7-bit segment number associated with the expression and a one bit in the most significant bit of the high-order byte, and all zero bits in the low-order byte (Section 1.4.1).

The "OFFSET" unary operator is applied to an address expression and returns a 16-bit value which is the offset value associated with the expression.

```
Example:  LD   R2,#SEG PTR
          LD   R3,#OFFSET PTR
                                  !Load segmented address of PTR
                                  into register pair RR2, which
                                  is functionally equivalent to
                                  the following statement!
          LDL  RR2,#PTR
```

Because of the special properties of these address operators, no other operators may be applied to a subexpression containing a SEG or OFFSET operator, although other operators can be used within the subexpression SEG or OFFSET are applied to:

```
 SEG (PTR+4)          {Valid}
 (SEG PTR)+4          {Invalid}
-(OFFSET PTR)         {Invalid}
```

2.4 Z8000 ADDRESSING MODES

With the exception of immediate data and condition codes, all assembly-language operands are expressed as addresses: register, memory, and I/O addresses. The various address modes recognized by the Z8000 assembler are as follows:

- Immediate Data
- Register
- Indirect Register
- Direct Address
- Indexed Address
- Relative Address
- Based Address
- Based Indexed Address

Special characters are used in operands to identify certain of these address modes. The characters are:

- "R" preceding a word register number;
- "RH" or "RL" preceding a byte register number;
- "RR" preceding a register pair number;
- "RQ" preceding a register quadruple number;
- "@" preceding an indirect-register reference;
- "#" preceding immediate data;
- "()" used to enclose the displacement part of an indexed, based, or based indexed address;
- "$" signifying the current program counter location, usually used in relative addressing.

The use of these characters is shown in the following sections.

Not every address mode can be used by every instruction. The individual instruction descriptions in Section 3 tell which address modes can be used for each instruction.

2.4.1 Immediate Data

The operand value used by the instruction in Immediate Data addressing mode is the value supplied in the operand field itself.

Immediate data is preceded by the special character "#" and may be a constant (including character constants and symbols representing constants) or an expression as described in Section 2.3.4. Immediate data expressions are evaluated using 32-bit arithmetic. Depending on the instruction being used, the value represented by the rightmost 4, 8, 16, or 32 bits is actually used. An error message is generated for values that overflow the valid range for the instruction.

```
        LDB    RH0, #100          !Load 100 into byte register RH0!

        LDL    RR0, #%8000 * REP_COUNT
                                  !Load the value resulting from
                                   the multiplication of %8000 and
                                   the value of constant REP_COUNT
                                   into register pair RR0!
```

If a variable name or address expression is prefixed by "#", the value used is the address represented by the variable or the result of the expression evaluation, not the contents of the corresponding data location. In Nonsegmented mode, all address expressions result in a 16-bit value.

For segmented addresses, the assembler automatically creates the proper format for a long offset address which includes the segment number and the offset in a 32-bit value (Section 1.4.1). It is recommended that symbolic names be used wherever possible, since the corresponding segment number and offset for the symbolic name will be automatically managed by the assembler and can be assigned values later when the module is linked or loaded for execution.

For those cases where a specific segment is desired, the following notation may be used (the segment designator is enclosed in double angle brackets):

```
        <<segment>>offset
```

where "segment" is a constant expression which evaluates to a 7-bit value, and "offset" is a constant expression which evaluates to a 16-bit value. This notation is expanded into a long offset address by the assembler.

```
LD    RO, #DATATABLE + 12     !Add 12 to the address of
                               DATATABLE and load the result
                               into word register R0 (non-
                               segmented mode)!

LDL   RR2, #ADDR              !Load the address of ADDR into
                               register pair RR2 (segmented
                               mode)!

LDL   RR2, #<<3>>%1234        !Load the segmented address
                               with segment 3, offset %1234
                               into register pair RR2 (seg-
                               mented mode)!
```

2.4.2 Register Address

In Register addressing mode, the operand value is the contents of the specified general-purpose register. There are four different sizes of registers on the Z8000.

- Word register (16 bits)
- Byte register (8 bits)
- Register pair (32 bits)
- Register quadruple (64 bits)

A word register is indicated by an "R" followed by a number from 0 to 15 (decimal). These correspond to the 16 registers of the machine. Either the high or low byte of the first eight registers can be accessed by using the byte register constructs "RH" or "RL" followed by a number from 0 to 7. Any pair of word registers can be accessed as a register pair by using "RR" followed by an even number between 0 and 14. Register quadruples are equivalent to four consecutive word registers and are accessed by the notation "RQ" followed by one of the numbers 0, 4, 8, or 12.

If an odd register number is given with a register pair designator, or a number other than 0, 4, 8, or 12 is given for a register quadruple, an assembly error will result.

In general, the size of a register used in an operation depends on the particular instruction. Byte registers are used with byte instructions, which end with the suffix "B". Word registers are used with word instructions, which have no special suffix. Register pairs are used with long word instructions, which end with the suffix "L". Register quadruples are used only with three instructions (DIVL, EXTSL and MULTL) which use a 64-bit value. An assembly error will occur if the size of a register does not correspond correctly with the particular instruction.

```
LD     R5, #%3FFF        !Load register 5 with the
                          hexadecimal value 3FFF!

LDB    RH3, #%F3         !Load the high order byte of
                          word register 3 with the
                          hexadecimal value F3!

ADDL   RR2, RR4          !Add the register pairs 2-3 and
                          4-5 and store the result in 2-3!

MULTL  RQ8, RR12         !Multiply the value in register
                          pair 10-11 by the value in
                          register pair 12-13 and store the
                          result in register quadruple
                          8-9-10-11!
```

2.4.3 Indirect Register Address

In Indirect Register addressing mode, the operand value is the contents of the location whose address is contained in the specified register. A word register is used to hold the address in nonsegmented mode, while a register pair must be used in segmented mode. Any general-purpose word register (or register pair in segmented mode) can be used except R0 or RR0.

Indirect Register addressing mode is also used with the I/O instructions and always indicates a 16-bit I/O address. Any general-purpose word register can be used except R0.

An Indirect Register address is specified by a "commercial at" symbol (@) followed by either a word register or a register pair designator. For Indirect Register addressing mode, a word register is specified by an "R" followed by a number from 1 to 15, and a register pair is specified by a "RR" followed by an even number from 2 to 14 (Section 2.4.2).

```
JP     @R2               !Pass control (jump) to the
                          program memory location
                          addressed by register 2
                          (nonsegmented mode)!

LD     @R3, R2           !Load contents of register
                          2 into location addressed by
                          register 3 (nonsegmented mode)!

LD     @RR2, #30         !Load immediate value 30 into
                          location addressed by register
                          pair 2-3 (segmented mode)!
```

2.4.4 Direct Address

The operand value used by the instruction in Direct addressing mode is the contents of the location specified by the address in the instruction. A direct address may be specified as a symbolic name of a memory or I/O location, or an expression which evaluates to an address. In nonsegmented mode or for all I/O instructions, the address is a 16-bit value. In segmented mode, the memory address is either a 16-bit value (short offset) or a 32-bit value (long offset). All assembly-time address expressions are evaluated using 32-bit arithmetic, with only the rightmost 16 bits of the result used for nonsegmented addresses.

```
LD   R10, DATUM          !Load the contents of the
                          location addressed by DATUM
                          into register 10!

LD   STRUCT+8, R10       !Load the contents of register
                          10 into the location addressed
                          by adding 8 to STRUCT!

JP   C, %2000            !Jump to location %2000 if the
                          carry flag is set (nonsegmented
                          mode)!

INB  RH0, 77             !Input the contents of the I/O
                          location addressed by 77 into
                          RH0!
```

For segmented addresses, the assembler automatically creates the proper format which includes the segment number and the offset. It is recommended that symbolic names be used wherever possible, since the corresponding segment number and offset for the symbolic name will be automatically managed by the assembler and can be assigned values later when the module is linked or loaded for execution.

For those cases where a specific segment is desired, the following notation may be used (the segment designator is enclosed in double angle brackets):

```
<<segment>>offset
```

where "segment" is a constant expression which evaluates to a 7-bit value, and "offset" is a constant expression which evaluates to a 16-bit value. This notation is expanded into a long offset address by the assembler.

To force a short offset address, the segmented address may be enclosed in vertical bars ("|"). In this case, the offset must be in the range 0 to 255, and the final address includes the segment number and short offset in a 16-bit value.

```
LD   R10, |DATUM|              !Load the contents of the
                                location addressed by DATUM
                                (short offset format) into
                                register 10!

LD   <<STORAGE>>HEAD, R10      !Load the contents of reg-
                                ister 10 into the location
                                addressed by the segment
                                named STORAGE offset by
                                HEAD (long offset format)!

JP   |<<3>>%12|               !Jump to location addressed
                                by segment 3, offset %12
                                (short offset format)!
```

2.4.5 Indexed Address

An indexed address consists of a memory address displaced by the contents of a designated word register (the index). This displacement is added to the memory address and the resulting address points to the location whose contents are used by the instruction. In nonsegmented mode, the memory address is specified as an expression which evaluates to a 16-bit value. In segmented mode, the memory address is specified as an expression which evaluates to either a 16-bit value (short offset) or a 32-bit value (long offset). All assembly-time address expressions are evaluated using 32-bit arithmetic, with only the rightmost 16 bits of the result used for nonsegmented addresses. This address is followed by the index, a word register designator enclosed in parentheses. For Indexed Addressing, a word register is specified by an "R" followed by a number from 1 to 15. Any general-purpose word register can be used except R0.

```
LD   R10, TABLE(R3)            !Load the contents of the
                                location addressed by TABLE
                                plus the contents of reg-
                                ister 3 into register 10!

LD   240+38(R3), R10           !Load the contents of reg-
                                ister 10 into the location
                                addressed by 278 plus the
                                contents of register 3
                                (nonsegmented mode)!
```

For segmented addresses, the assembler automatically creates the proper format for the memory address which includes the segment number and the offset. As with Direct Addressing, symbolic names should be used wherever possible so that values can be assigned later when the module is linked or loaded for execution.

For those cases where a specific segment is desired, the following notation may be used (the segment designator is enclosed in double angle brackets):

```
<<segment>>offset(R)
```

where "segment" is a constant expression which evaluates to a 7-bit value, "offset" is a constant expression which evaluates to a 16-bit value, and "R" is a word register designator. This notation is expanded into a long offset address by the assembler.

To force a short offset address, the segmented address may be enclosed in vertical bars ("|"). In this case, the offset must be in the range 0 to 255, and the final address includes the segment number and short offset in a 16-bit value.

```
LD   R10, |TABLE|(R3)          !Load the contents of the
                                location addressed by
                                TABLE (short offset format)
                                plus the contents of reg-
                                ister 3 into register 10!

LD   <<STACK>>8(R13), R10      !Load the contents of reg-
                                ister 10 into the location
                                addressed by the segment
                                named STACK offset by 8
                                (long offset format) plus
                                the contents of register 13!
```

2.4.6 Relative Address

Relative address mode is implied by its instruction. It is used by the Call Relative (CALR), Decrement and Jump If Not Zero (DJNZ), Jump Relative (JR), Load Address Relative (LDAR), and Load Relative (LDR) instructions and is the only mode available to these instructions. The operand, in this case, represents a displacement that is added to the contents of the program counter to form the destination address that is relative to the current instruction. The original contents of the program counter is taken to be the address of the instruction byte <u>following</u> the instruction. The size and range of the displacement depends on the particular instruction, and is described with each instruction in Section 3.

The displacement value can be expressed in two ways. In the first case, the programmer provides a specific displacement in the form "$+n" where n is a constant expression in the range appropriate for the particular instruction and $ represents the contents of the program counter at the <u>start</u> of the instruction. The assembler automatically subtracts the value of the address of the <u>following</u> instruction to derive the actual displacement.

```
JR  OV, $+K      !Add value of constant K to program
                  counter and jump to new location if
                  overflow has occurred!
```

In the second method, the assembler calculates the displacement automatically. The programmer simply specifies an expression which evaluates to a number or a program label as in Direct Addressing. The address specified by the operand must be in the valid range for the instruction, and the assembler automatically subtracts the value of the address of the following instruction to derive the actual displacement.

```
DJNZ R5, LOOP        !Decrement register 5 and jump to
                      LOOP if the result is not zero!

LDR  R10, DATA       !Load the contents of the location
                      addressed by DATA into register 10!
```

2.4.7 Based Address

A based address consists of a register that contains the base and a 16-bit displacement. The displacement is added to the base and the resulting address indicates the location whose contents are used by the instruction.

In nonsegmented mode, the base address is held in a word register that is specified by an "R" followed by a number from 1 to 15. Any general-purpose word register can be used except R0. The displacement is specified as an expression which evaluates to a 16-bit value, preceded by a "#" symbol and enclosed in parentheses.

In segmented mode, the segmented base address is held in a register pair that is specified by an "RR" followed by an even number from 2 to 14. Any general-purpose register pair can be used except RR0. The displacement is specified as an expression which evaluates to a 16-bit value, preceded by a "#" symbol and enclosed in parentheses.

```
LDL   RR2, R1(#255)      !Load into register pair 2-3 the
                          long word value found in the
                          location resulting from adding
                          255 to the address in register
                          1 (nonsegmented mode)!

LD    RR4(#%4000), R2    !Load register 2 into the loca-
                          tion addressed by adding %4000
                          to the segmented address found
                          in register pair 4-5 (segmented
                          mode)!
```

2.4.8 Based Indexed Address

Based Indexed addressing is similar to Based addressing (2.4.7) except that the displacement ("index") as well as the base is held in a register. The contents of the registers are added together to determine the address used in the instruction.

In nonsegmented mode, the base address is held in a word register that is specified by an "R" followed by a number from 1 to 15. The index is held in a word register specified in a similar manner and enclosed in parentheses. Any general-purpose word registers can be used for either the base or index except R0.

In segmented mode, the segmented base address is held in a register pair that is specified by an "RR" followed by an even number from 2 to 14. Any general-purpose register pair can be used except RR0. The index is held in a word register that is specified by an "R" followed by a number from 1 to 15. Any general-purpose word register can be used except R0.

```
LD      R3, R8(R15)        !Load the value at the location
                            addressed by adding the address
                            in R8 to the displacement in
                            R15 into register 3 (nonseg-
                            mented mode)!

LDB     RR14(R4), RH2      !Load register RH2 into the
                            location addressed by the seg-
                            mented address in RR14 indexed by
                            the value in R4 (segmented mode)!
```

Section 3
Assembly Language Instruction Set

3.1 FUNCTIONAL SUMMARY

Z8000 PLZ/ASM instructions can be divided functionally into nine groups:

- Load and Exchange
- Arithmetic
- Logical
- Program Control
- Bit Manipulation
- Rotate and Shift
- Block Transfer and String Manipulation
- Input/Output
- CPU Control

The instruction summary shows the instructions belonging to each functional group and the number of operands required for each. The following notation is used:

- Operations with word, byte and long word data operands are listed with their instruction mnemonics grouped together. The suffix "B" designates a byte instruction, the suffix "L" designates a long word instruction, and no suffix designates a word instruction. For some instructions, a data size is either not applicable or depends on the segmentation mode, and thus the instruction mnemonic does not have a suffix to indicate data size.

- "src" is the source operand ("src1" and "src2" are used to distinguish between two source operands in the same instruction)

- "dst" is the destination operand

- "r" is a register operand, typically used as a counter

- "num" is a number; that is, an immediate value

- "cc" is a condition code (Section 3.2.1)

- "flag" is any combination of the C, Z, S, P, and V status flags

- "int" is any combination of the VI and NVI interrupt control bits

LOAD AND EXCHANGE INSTRUCTIONS

Instruction	Operand(s)	Name of Instruction
CLR CLRB	dst	Clear
EX EXB	dst,src	Exchange
LD LDB LDL	dst,src	Load
LDA	dst,src	Load Address
LDAR	dst,src	Load Address Relative
LDK	dst,src	Load Constant
LDM	dst,src,num	Load Multiple
LDR LDRB LDRL	dst,src	Load Relative
POP POPL	dst,src	Pop
PUSH PUSHL	dst,src	Push

ARITHMETIC INSTRUCTIONS

Instruction	Operand(s)	Name of Instruction
ADC ADCB	dst,src	Add with Carry
ADD ADDB ADDL	dst,src	Add
CP CPB CPL	dst,src	Compare

ARITHMETIC INSTRUCTIONS (continued)

Instruction	Operands(s)	Name of Instruction
DAB	dst	Decimal Adjust
DEC DECB	dst,src	Decrement
DIV DIVL	dst,src	Divide
EXTS EXTSB EXTSL	dst	Extend Sign
INC INCB	dst,src	Increment
MULT MULTL	dst,src	Multiply
NEG NEGB	dst	Negate
SBC SBCB	dst,src	Subtract with Carry
SUB SUBB SUBL	dst,src	Subtract

LOGICAL INSTRUCTIONS

Instruction	Operand(s)	Name of Instruction
AND ANDB	dst,src	And
COM COMB	dst	Complement
OR ORB	dst,src	Or

LOGICAL INSTRUCTIONS (continued)

Instruction	Operand(s)	Name of Instruction
TEST TESTB TESTL	dst	Test
TCC TCCB	cc,dst	Test Condition Code
XOR XORB	dst,src	Exclusive Or

PROGRAM CONTROL INSTRUCTIONS

Instruction	Operand(s)	Name of Instruction
CALL	dst	Call Procedure
CALR	dst	Call Procedure Relative
DJNZ DBJNZ	r,dst	Decrement and Jump if Not Zero
IRET		Interrupt Return
JP	cc,dst	Jump
JR	cc,dst	Jump Relative
RET	cc	Return from Procedure
SC	src	System Call

BIT MANIPULATION INSTRUCTIONS

Instruction	Operand(s)	Name of Instruction
BIT BITB	dst,src	Bit Test

BIT MANIPULATION INSTRUCTIONS (continued)

Instruction	Operand(s)	Name of Instruction
RES RESB	dst,src	Reset Bit
SET SETB	dst,src	Set Bit
TSET TSETB	dst	Test and Set

ROTATE AND SHIFT INSTRUCTIONS

Instruction	Operand(s)	Name of Instruction
RL RLB	dst,src	Rotate Left
RLC RLCB	dst,src	Rotate Left through Carry
RLDB	dst,src	Rotate Left Digit
RR RRB	dst,src	Rotate Right
RRC RRCB	dst,src	Rotate Right through Carry
RRDB	dst,src	Rotate Right Digit
SDA SDAB SDAL	dst,src	Shift Dynamic Arithmetic
SDL SDLB SDLL	dst,src	Shift Dynamic Logical
SLA SLAB SLAL	dst,src	Shift Left Arithmetic

ROTATE AND SHIFT INSTRUCTIONS (continued)

Instruction	Operand(s)	Name of Instruction
SLL SLLB SLLL	dst,src	Shift Left Logical
SRA SRAB SRAL	dst,src	Shift Right Arithmetic
SRL SRLB SRLL	dst,src	Shift Right Logical

BLOCK TRANSFER AND STRING MANIPULATION INSTRUCTIONS

Instruction	Operand(s)	Name of Instruction
CPD CPDB	dst,src,r,cc	Compare and Decrement
CPDR CPDRB	dst,src,r,cc	Compare, Decrement and Repeat
CPI CPIB	dst,src,r,cc	Compare and Increment
CPIR CPIRB	dst,src,r,cc	Compare, Increment and Repeat
CPSD CPSDB	dst,src,r,cc	Compare String and Decrement
CPSDR CPSDRB	dst,src,r,cc	Compare String, Decrement and Repeat
CPSI CPSIB	dst,src,r,cc	Compare String and Increment
CPSIR CPSIRB	dst,src,r,cc	Compare String, Increment and Repeat
LDD LDDB	dst,src,r	Load and Decrement

BLOCK TRANSFER AND STRING MANIPULATION INSTRUCTIONS (continued)

Instruction	Operand(s)	Name of Instruction
LDDR LDDRB	dst,src,r	Load, Decrement and Repeat
LDI LDIB	dst,src,r	Load and Increment
LDIR LDIRB	dst,src,r	Load, Increment and Repeat
TRDB	dst,src,r	Translate and Decrement
TRDRB	dst,src,r	Translate, Decrement and Repeat
TRIB	dst,src,r	Translate and Increment
TRIRB	dst,src,r	Translate, Increment and Repeat
TRTDB	src1,src2,r	Translate, Test and Decrement
TRTDRB	src1,src2,r	Translate, Test, Decrement and Repeat
TRTIB	src1,src2,r	Translate, Test and Increment
TRTIRB	src1,src2,r	Translate, Test, Increment and Repeat

INPUT/OUTPUT INSTRUCTIONS

Instruction	Operand(s)	Name of Instruction
IN INB	dst,src	Input
IND INDB	dst,src,r	Input and Decrement

INPUT/OUTPUT INSTRUCTIONS (continued)

Instructions	Operand(s)	Name of Instruction
INDR INDRB	dst,src,r	Input, Decrement and Repeat
INI INIB	dst,src,r	Input and Increment
INIR INIRB	dst,src,r	Input, Increment and Repeat
OTDR OTDRB	dst,src,r	Output, Decrement and Repeat
OTIR OTIRB	dst,src,r	Output, Increment and Repeat
OUT OUTB	dst,src	Output
OUTD OUTDB	dst,src,r	Output and Decrement
OUTI OUTIB	dst,src,r	Output and Increment
SIN SINB	dst,src	Special Input
SIND SINDB	dst,src,r	Special Input and Decrement
SINDR SINDRB	dst,src,r	Special Input, Decrement and Repeat
SINI SINIB	dst,src,r	Special Input and Increment
SINIR SINIRB	dst,src,r	Special Input, Increment and Repeat
SOTDR SOTDRB	dst,src,r	Special Output, Decrement and Repeat
SOTIR SOTIRB	dst,src,r	Special Output, Increment and Repeat

INPUT/OUTPUT INSTRUCTIONS (continued)

Instruction	Operand(s)	Name of Instruction
SOUT SOUTB	dst,src	Special Output
SOUTD SOUTDB	dst,src,r	Special Output and Decrement
SOUTI SOUTIB	dst,src,r	Special Output and Increment

CPU CONTROL INSTRUCTIONS

Instruction	Operand(s)	Name of Instruction
COMFLG	flag	Complement Flag
DI	int	Disable Interrupt
EI	int	Enable Interrupt
HALT		Halt
LDCTL LDCTLB	dst,src	Load Control Register
LDPS	src	Load Program Status
MBIT		Multi-Micro Bit Test
MREQ	dst	Multi-Micro Request
MRES		Multi-Micro Reset
MSET		Multi-Micro Set
NOP		No Operation
RESFLG	flag	Reset Flag
SETFLG	flag	Set Flag

The following set of instructions are privileged; that is, they can only be executed in System mode:

DI	INIB	OTDR	SIN	SOTDR
EI	INIR	OTDRB	SINB	SOTDRB
HALT	INIRB	OTIR	SIND	SOTIR
IN	IRET	OTIRB	SINDB	SOTIRB
INB	LDCTL	OUT	SINDR	SOUT
IND	LDPS	OUTB	SINDRB	SOUTB
INDB	MBIT	OUTD	SINI	SOUTD
INDR	MREQ	OUTDB	SINIB	SOUTDB
INDRB	MRES	OUTI	SINIR	SOUTI
INI	MSET	OUTIB	SINIRB	SOUTIB

3.2 NOTATION AND BINARY ENCODING

3.2.1 Operand Notation

Operands are represented by a notational shorthand in the detailed instruction descriptions that make up the rest of this section. The notation for operands and their actual assembly language syntax are as follows:

Notation	Address Mode	Actual Operand
IM	Immediate Data	#expression
R	Register	Rn, where n=0-15; RHn, or RLn, where n=0-7; RRn, where n=0,2,4,...14; RQn, where n=0,4,8,12
IR	Indirect Register	@Rn, where n=1-15 for nonsegmented mode; @RRn, where n=2,4,6...14 for segmented mode
DA	Direct Address	expression
X	Indexed Address	expression(Rn), where n=1-15
RA	Relative Address	expression
BA	Based Address	Rn(#expression), where n=1-15 for nonsegmented mode; RRn(#expression), where n=2,4,6...14 for segmented mode
BX	Based Indexed Address	Rn(Rn), where n=1-15 for nonsegmented mode; RRn(Rm), where n=2,4,6...14 and m=1-15 for segmented mode

The control registers (CTLR) are:

FLAGS	Status Flags
FCW	Flags and Control Word
REFRESH	Refresh
PSAP	Program Status Area Pointer*
PSAPSEG	Program Status Area Pointer Segment
PSAPOFF	Program Status Area Pointer Offset
NSP	Normal Stack Pointer*
NSPSEG	Normal Stack Pointer Segment
NSPOFF	Normal Stack Pointer Offset

* For the nonsegmented Z8002, PSAP is equivalent to PSAPOFF and NSP is equivalent to NSPOFF.

The status flags are:

C	Carry flag
Z	Zero flag
S	Sign flag
P	Parity flag
V	Overflow flag
D	Decimal-adjust flag*
H	Half-carry flag*

* These flags may not be specified in assembly language statements.

The interrupt control bits are:

VI	Vectored Interrupt
NVI	Non-Vectored Interrupt

The binary values used in the instruction format encoding for the general-purpose registers are:

Register				Binary
RQ0	RR0	R0	RH0	0000
		R1	RH1	0001
	RR2	R2	RH2	0010
		R3	RH3	0011
RQ4	RR4	R4	RH4	0100
		R5	RH5	0101

Register				Binary
	RR6	R6	RH6	0110
		R7	RH7	0111
RQ8	RR8	R8	RL0	1000
		R9	RL1	1001
	RR10	R10	RL2	1010
		R11	RL3	1011
RQ12	RR12	R12	RL4	1100
		R13	RL5	1101
	RR14	R14	RL6	1110
		R15	RL7	1111

The condition codes and the flag settings they represent are:

Code	Meaning	Flag Settings	Binary
	Always false	-	0000
(blank)	Always true	-	1000
Z	Zero	Z=1	0110
NZ	Not zero	Z=0	1110
C	Carry	C=1	0111
NC	No carry	C=0	1111
PL	Plus	S=0	1101
MI	Minus	S=1	0101
NE	Not equal	Z=0	1110
EQ	Equal	Z=1	0110
OV	Overflow	V=1	0100
NOV	No overflow	V=0	1100
PE	Parity even	P=1	0100
PO	Parity odd	P=0	1100
GE	Greater than or equal	(S XOR V) = 0	1001
LT	Less than	(S XOR V) = 1	0001
GT	Greater than	(Z OR (S XOR V)) = 0	1010
LE	Less than or equal	(Z OR (S XOR V)) = 1	0010
UGE	Unsigned greater than or equal	C=0	1111
ULT	Unsigned less than	C=1	0111
UGT	Unsigned greater than	((C=0) AND (Z=0)) = 1	1011
ULE	Unsigned less than or equal	(C OR Z) = 1	0011

Note that some of the condition codes correspond to identical flag settings: i.e., Z-EQ, NZ-NE, C-ULT, NC-UGE, PE-OV, PO-NOV.

3.2.2 Instruction Format Encoding

The binary encoding of each Z8000 instruction follows one of nine basic formats. Each format determines the grouping of bits into various instruction "fields", the meaning of which depends on the instruction. The following symbols represent these fields:

Symbol	Field Size	Value
mode	2 bits	Addressing mode specifier
src	Varies	Source operand address or immediate value
dst	Varies	Destination operand address
r	4 bits	Register counter address
num	4 bits	Number (immediate value)
cc	4 bits	Condition code
W/B	1 bit	Word or byte specifier
x	4 bits	Index register address for Based Indexed Addressing mode
displacement	Varies	Displacement for Relative or Based Addressing modes

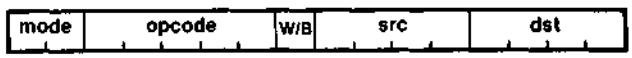
All other bit fields are referred to as operation code (opcode) fields because they determine the operation of the instruction.

In addition to the various fields within an instruction, address information for the data operands may be appended to the basic instruction encoding. The size and format of this additional address information depends on the particular address mode, as well as the data operand size in the case of Immediate Data, or the segmentation mode in the case of Direct Address or Indexed Address.

The standard formats for address information are illustrated by the following examples. Nonstandard formats are described in the detailed description for each instruction (Section 3.3). The following examples use Addition (ADD), Load (LD), and Load Relative (LDR) instructions for illustration purposes, although the address information formats apply to all instructions which require additional address information.

Register (R)

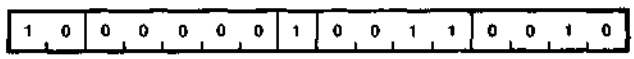
```
    ADD  dst  src
         R,   R
```



{mode = 10}

Example:

```
    ADD  R2,R3
```



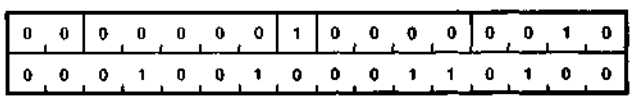
Immediate Data -- Word (IM)

```
    ADD  dst  src
         R,   IM
```

{mode = 00, src field = 0}

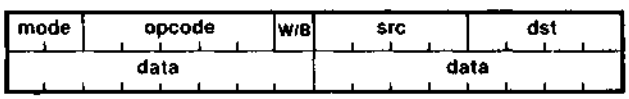
Example:

```
    ADD  R2,#%1234
```



Immediate Data -- Byte (IM)

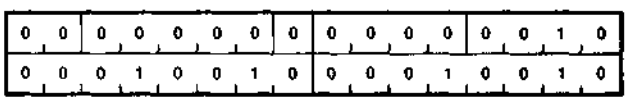
```
    ADDB dst  src
         R,   IM
```



{mode = 00, src field = 0, Byte Immediate Data is duplicated in each half of the second instruction word}

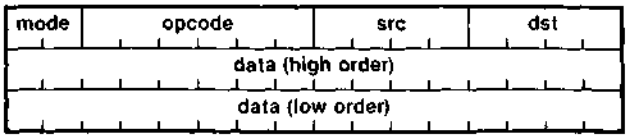
Example:

```
    ADDB RH2,#%12
```



Immediate Data -- Long (IM)

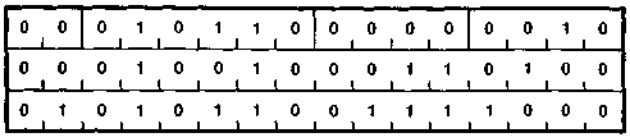
```
      dst  src
ADDL   R,   IM
```



{mode = 00, src field = 0}

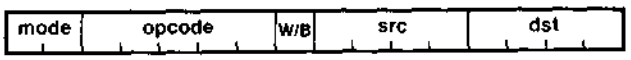
Example:

```
ADDL RR2,#%12345678
```



Indirect Register (IR)

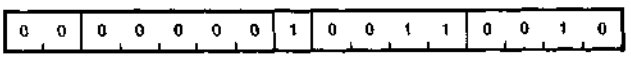
```
      dst  src
ADD    R,   IR
```



{mode = 00, src field <> 0}

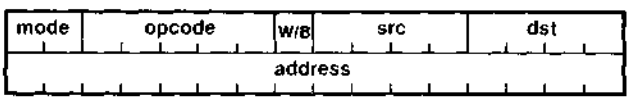
Example:

```
ADD  R2,@R3
```



Direct Address -- Nonsegmented (DA)

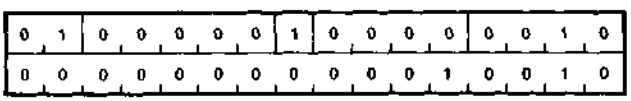
```
      dst  src
ADD    R,   DA
```



{mode = 01, src field = 0}

Example:

```
ADD  R2,%0012
```



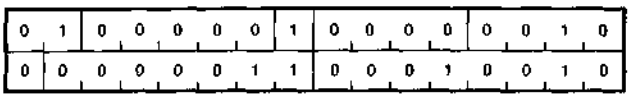
Direct Address -- Segmented Short Offset (DA)

```
        dst  src
ADD     R,   DA
```

{mode = 01, src field = 0}

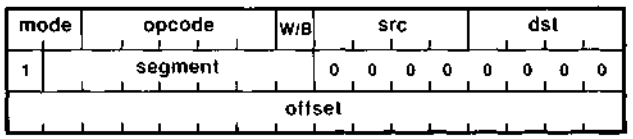
Example:

```
ADD   R2,|<<3>>%0012|
```



Direct Address -- Segmented Long Offset (DA)

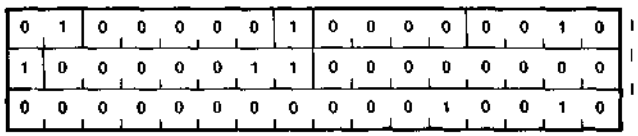
```
        dst  src
ADD     R,   DA
```



{mode = 01, src field = 0}

Example:

```
ADD   R2,<<3>>%0012
```



Indexed Address -- Nonsegmented (X)

```
        dst  src
ADD     R,   X
```

{mode = 01, src field <> 0}

Example:

```
ADD   R2,%0012(R4)
```

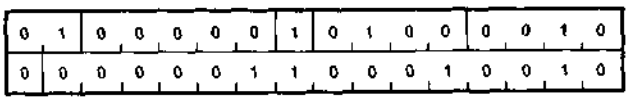
Indexed Address -- Segmented Short Offset (X)

```
        dst   src
ADD     R,    X
```

{mode = 01, src field <> 0}

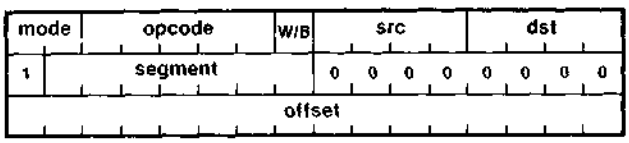
Example:

```
ADD R2,|<<3>>%0012|(R4)
```



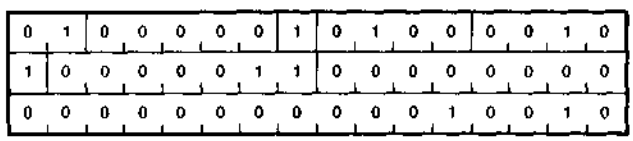
Indexed Address -- Segmented Long Offset (X)

```
        dst   src
ADD     R,    X
```



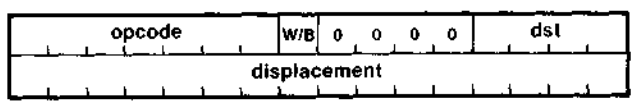
Example:

```
ADD   R2,<<3>>%0012(R4)
```



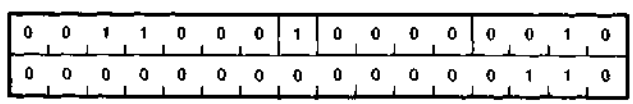
Relative Address (RA)

```
        dst   src
LDR     R,    RA
```



Example:

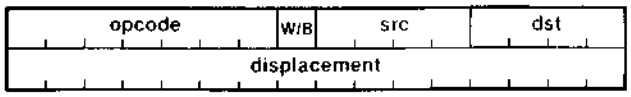
```
LDR   R2,$+10
```



{The assembler automatically calculates the relative address displacement (=6) as the given value (10 bytes after the start of the LDR instruction) minus the start of the following instruction (4 bytes after the start of the LDR instruction)}

Based Address (BA)

```
        dst   src
LD       R,   BA
```



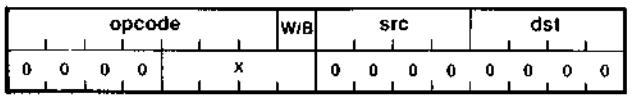
{src field <> 0}

Example:

```
LD  R2,R4(#%0012)
```

Based Indexed Address (BX)

```
        dst   src
LD       R,   BX
```



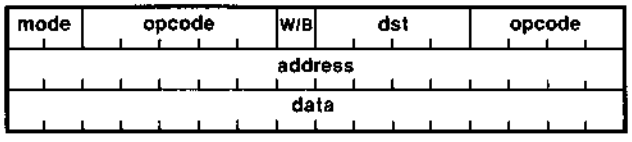
{src field <> 0}

Example:

```
LD  R2,R4(R5)
```

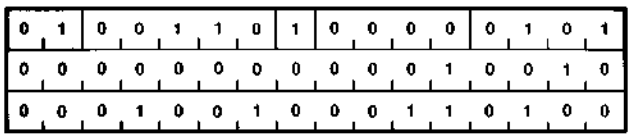
There are several instructions whose source operand is Immediate Data and destination operand is either Direct Address or Indexed Address. In this case, the Immediate Data follows the address information (DA or X) in the instruction encoding. The following example uses the nonsegmented DA format

```
        dst   src
LD      DA,   IM
```



Example:

```
LD  %0012,#%1234
```



The nine basic formats, F1 through F9, determine the size of the instruction, which also depends on the addressing mode. The following table can be used to determine the number of bytes in a particular instruction. The addressing mode (including the data operand size and whether segmented or nonsegmented mode is used) specifies the row in the table, while the format number specifies the column.

TABLE 3-1. NUMBER OF BYTES IN INSTRUCTIONS

MODE	FORMAT								
dst or src	F1	F2	F3	F4	F5	F6	F7	F8	F9
R, IR	2	2	--	--	4	4	2	2	--
IM (Word/Byte)	4	4	2	--	--	--	--	--	2
IM (Long)	6	6	--	--	--	--	--	--	--
DA,X (NS/SS)	4	4	--	--	6	6	4	--	--
DA,X (SL)	6	6	--	--	8	8	--	--	--
RA	--	--	2	4	--	--	--	--	--
BA	--	--	--	4	--	--	--	--	--
BX	--	--	--	4	--	--	--	--	--

NS = Nonsegmented, SS = Segmented Short Offset, SL = Segmented Long Offset

3.2.3 Operation Notation

The description of each instruction's operation includes a shorthand summary. The following symbols are used in the operation summary.

Symbol	Meaning
src	Source operand
dst	Destination operand
r	Register operand (typically used as a counter)
num	Number, that is, an immediate value
cc	Condition code
instruction	Binary value of the instruction itself
Rn	Register number "n"
PC	Program Counter register
PS	Program Status register
SP	Processor stack pointer register (R15 if nonsegmented, RR14 if segmented)
<-	Becomes (assignment of a value)
@n	Indirect; that is, the value found at the address contained in "n"
n1,n2	Concatenation; that is, both "n1" and "n2" are operated on together
f∨n	Operand addressed by the Boolean OR of the instruction field "f" and the value "n" (e.g., R0∨1 is R1)

Symbol	Meaning
operand(n)	Bit "n" of operand
operand(msb)	Most significant bit of operand
operand(n1:n2)	Bits "n1" through "n2" of operand
src[dst]	Indexed access, that is, the operand whose address is contained in the source operand offset by the destination operand.
AUTOINCREMENT	The address of the operand is automatically incremented by the size of the operand in bytes (the address is always contained in a register)
AUTODECREMENT	The address of the operand is automatically decremented by the size of the operand in bytes (the address is always contained in a register)

3.3 ASSEMBLY LANGUAGE INSTRUCTIONS

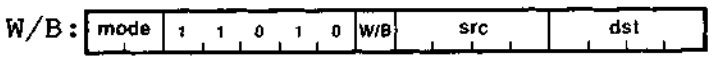
In the remainder of this chapter, Z8000 assembly language instructions are described in detail in alphabetic order. Each description includes:

- The name of the instruction.
- The assembly language statement format(s) including the required operands and valid addressing modes (see section 3.2.1).
- The binary instruction format(s) including the instruction fields and the format class (see section 3.2.2). If the instruction has a "mode" field, then the valid addressing modes for that format are listed to the right of the binary encoding. If the instruction has a "W/B" field, then a "1" bit indicates a word operation and a "0" bit indicates a byte operation. The format class appears in parentheses above the binary encoding and consists of two numbers (Fn.m). "Fn" specifies the basic format class F1 through F9 and can be used with Table 3-1 to determine the number of bytes for a particular instruction. The "m" portion of the format number specifies the sub-class (see Appendix A).
- The operation performed by the instruction including a summary description (see section 3.2.3) and a detailed description.
- The status flags affected by the instruction.
- The number of machine cycles used to execute the instruction.
- Special notes for some instructions concerning nonstandard instruction field encodings, hardware related functions, operation restrictions, and other architectural or implementation details.
- A short assembly language example showing the use of the instruction. All examples assume the nonsegmented mode unless otherwise specified.

ADC
Add with Carry

```
ADC   dst,src                    dst: R
ADCB                             src: R
```

INSTRUCTION FORMAT: (F2.1)



mode	dst	src
10	R	R

OPERATION: dst <- dst + src + C

The source operand, along with the setting of the carry flag, is added to the destination operand and the sum is stored in the destination. The contents of the source are not affected. Two's complement addition is performed. In multiple precision arithmetic, this instruction permits the carry from the addition of low-order operands to be carried into the addition of high-order operands.

FLAGS:
- C: Set if there is a carry from the most significant bit of the result; cleared otherwise
- Z: Set if the result is zero; cleared otherwise
- S: Set if the result is negative; cleared otherwise
- V: Set if arithmetic overflow occurs, that is, if both operands were of the same sign and the result is of the opposite sign; cleared otherwise
- D: ADC - unaffected; ADCB - cleared
- H: ADC - unaffected; ADCB - set if there is a carry from the most significant bit of the low-order four bits of the result; cleared otherwise

CYCLES:

Word/Byte
5

EXAMPLE: Long addition may be done with the following instruction sequence, assuming R0,R1 contain one operand and R2,R3 contain the other operand:

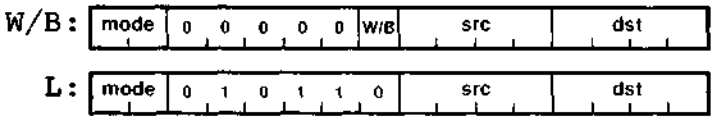
```
ADD  R1,R3  !add low-order words!
ADC  R0,R2  !add carry and high-order words!
```

If R0 contains %0000, R1 contains %FFFF, R2 contains %4320 and R3 contains %0001, then the above two instructions leave the value %4321 in R0 and %0000 in R1.

ADD
Add

```
ADD   dst,src                dst: R
ADDB                         src: R, IM, IR, DA, X
ADDL
```

INSTRUCTION FORMAT: (F2.1)



mode	dst	src
10	R	R
00	R	IM (src field=0)
00	R	IR (src field<>0)
01	R	DA (src field=0)
01	R	X (src field<>0)

OPERATION: dst <- dst + src

The source operand is added to the destination operand and the sum is stored in the destination. The contents of the source are not affected. Two's complement addition is performed.

FLAGS:
- C: Set if there is a carry from the most significant bit of the result; cleared otherwise
- Z: Set if the result is zero; cleared otherwise
- S: Set if the result is negative; cleared otherwise
- V: Set if arithmetic overflow occurs, that is, if both operands were of the same sign and the result is of the opposite sign; cleared otherwise
- D: ADD,ADDL - unaffected; ADDB - cleared
- H: ADD,ADDL - unaffected; ADDB - set if there is a carry from the most significant bit of the low-order four bits of the result; cleared otherwise

CYCLES:

src	Word/byte			Long		
	NS	SS	SL	NS	SS	SL
R	4	--	--	8	--	--
IM	7	--	--	14	--	--
IR	7	--	--	14	--	--
DA	9	10	12	15	16	18
X	10	10	13	16	16	19

EXAMPLE: If register R2 contains %0344 and the word at the location named AUGEND contains %0011, the statement

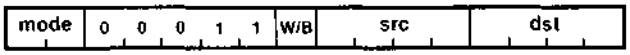
```
ADD  R2,AUGEND
```

will leave the value %0355 in R2.

```
AND   dst,src                dst: R
ANDB                         src: R, IM, IR, DA, X
```

INSTRUCTION FORMAT: (F2.1)

W/B:



mode	dst	src
10	R	R
00	R	IM (src field=0)
00	R	IR (src field<>0)
01	R	DA (src field=0)
01	R	X (src field<>0)

OPERATION: dst <- dst AND src

The source operand is logically ANDed with the destination operand and the result is stored in the destination. The contents of the source are not affected. The AND operation results in a one bit being stored whenever the corresponding bits in the two operands are both ones; otherwise, a zero bit is stored.

FLAGS:
- C: Unaffected
- Z: Set if the result is zero; cleared otherwise
- S: Set if the most significant bit of the result is set; is even; cleared otherwise
- P: AND - unaffected; ANDB - set if parity of the result is even; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

src	Word/Byte		
	NS	SS	SL
R	4	--	--
IM	7	--	--
IR	7	--	--
DA	9	10	12
X	10	10	13

EXAMPLE: If register RL3 contains %C3 (11000011) and the source operand is the immediate value %7B (01111011), the statement

```
ANDB  RL3,#%7B
```

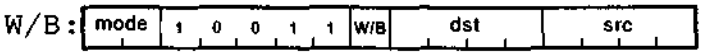
will leave the value %43 (01000011) in RL3.

BIT
Bit Test

```
BIT   dst,src
BITB
```

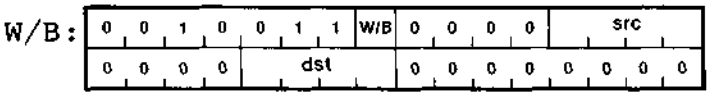
dst: R, IR, DA, X
src: R, IM

INSTRUCTION FORMAT: (F1.2)



mode	dst	src
10	R	IM
00	IR	IM (dst field<>0)
01	DA	IM (dst field=0)
01	X	IM (dst field<>0)

(F6.3)



dst	src
R	R

OPERATION: Z <- NOT dst(src)

Tests the specified bit within the destination operand, and sets the Z flag if the specified bit is 0; otherwise it clears the Z flag. The contents of the destination are not affected. The source (the bit number) can be specified as either an immediate value, or as a word register which contains the value. In the second case, the destination operand must be a register, and the source operand must be R0 through R7 for BITB, or R0 through R15 for BIT. The bit number is a value from 0 to 7 for BITB, or 0 to 15 for BIT, with 0 indicating the least significant bit.

FLAGS:
C: Unaffected
Z: Set if specified bit is zero; cleared otherwise
S: Unaffected
V: Unaffected
D: Unaffected
H: Unaffected

CYCLES:

dst	src	Word/Byte NS	SS	SL
R	IM	4	--	--
IR	IM	8	--	--
DA	IM	10	11	13
X	IM	11	11	14
R	R	10	--	--

NOTE: Only the lower four bits of the source operand are used to specify the bit number for BIT, while only the lower three bits of the source operand are used with BITB. When the source operand is an immediate value, the "src field" in the instruction format encoding contains the bit number in the lowest four bits for BIT, or the lowest three bits for BITB.

EXAMPLE: If register RH2 contains %B2 (10110010), the statement

```
BITB  RH2,#0
```

will leave the Z flag set.

CALL
Call Procedure

CALL dst dst: IR, DA, X

INSTRUCTION FORMAT: (F1.1)

mode	dst
00	IR
01	DA (dst field=0)
01	X (dst field<>0)

OPERATION:

Nonsegmented	Segmented
SP <- SP - 2	SP <- SP - 4
@SP <- PC	@SP <- PC
PC <- dst	PC <- dst

The current contents of the program counter (PC) are pushed onto the top of the processor stack. The stack pointer used is R15 if nonsegmented, or RR14 if segmented. (The program counter value used is the address of the first instruction byte following the CALL instruction.) The specified destination address is then loaded into the PC and points to the first instruction of a procedure.

At the end of the procedure a RET instruction can be used to return to original program flow. RET pops the top of the processor stack back into the PC.

FLAGS: No flags affected

CYCLES:

dst	Address		
	NS	SS	SL
IR	10	--	15
DA	12	18	20
X	13	18	21

EXAMPLE: In nonsegmented mode, if the contents of the program counter are %1000 and the contents of the stack pointer (R15) are %3002, the statement

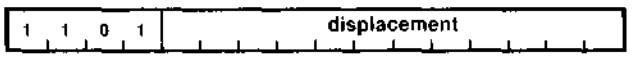
```
CALL  %2520
```

causes the stack pointer to be decremented to %3000, the value %1004 (the address following the CALL instruction with direct address mode specified) to be loaded into the word at location %3000, and the program counter to be loaded with the value %2520. The program counter now points to the address of the first instruction in the procedure to be executed.

CALR
Call Procedure Relative

CALR dst dst: RA

INSTRUCTION FORMAT: (F3.4)



dst
RA

OPERATION:

Nonsegmented	Segmented
SP <- SP - 2	SP <- SP - 4
@SP <- PC	@SP <- PC
PC <- PC - (2*disp)	PC <- PC - (2*disp)

The current contents of the program counter (PC) are pushed onto the top of the processor stack. The stack pointer used is R15 if nonsegmented, or RR14 if segmented. (The program counter value used is the address of the first instruction byte following the CALR instruction.) The destination address is calculated and then loaded into the PC and points to the first instruction of a procedure.

At the end of the procedure a RET instruction can be used to return to the original program flow. RET pops the top of the processor stack back into the PC.

FLAGS: No flags affected

CYCLES:

Address		
NS	SS	SL
10	--	15

NOTE: The relative addressing mode is calculated by doubling the displacement in the instruction, then subtracting this value from the updated value of the PC to derive the destination address. The updated PC value is taken to be the address of the instruction byte following the CALR instruction, while the displacement is a 12-bit signed value in the range -2048 to +2047. Thus, the destination address must be in the range -4092 to +4098 bytes from the start of the CALR instruction. In the segmented mode, the PC segment number is not affected. The assembler automatically calculates the displacement by subtracting the PC value of the following instruction from the address given by the programmer.

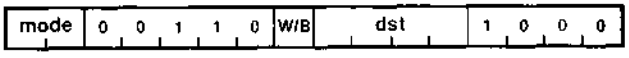
```
EXAMPLE:        CALR  PROC      !Procedure PROC is called!
NEXT:                           !Control eventually returns to here!
```

CLR
Clear

```
CLR   dst                          dst: R, IR, DA, X
CLRB
```

INSTRUCTION FORMAT: (Fl.1)

W/B:



mode	dst
10	R
00	IR
01	DA (dst field=0)
01	X (dst field<>0)

OPERATION: dst <- 0

The destination is cleared to zero.

FLAGS: No flags affected

CYCLES:

dst	Word/Byte		
	NS	SS	SL
R	7	--	--
IR	8	--	--
DA	11	12	14
X	12	12	15

EXAMPLE: If the word at location %ABBA contains 13, the statement

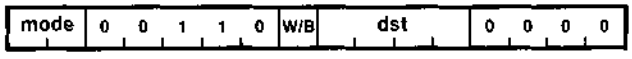
```
CLR  %ABBA
```

will leave the value 0 in the word at location %ABBA.

COM
Complement

```
COM   dst                          dst: R, IR, DA, X
COMB
```

INSTRUCTION FORMAT: (F1.1)



mode	dst
10	R
00	IR
01	DA (dst field=0)
01	X (dst field<>0)

OPERATION: dst <- NOT dst

The contents of the destination are complemented (one's complement); all one bits are changed to zero, and vice-versa.

FLAGS:
- C: Unaffected
- Z: Set if the result is zero; cleared otherwise
- S: Set if the most significant bit of the result is set; cleared otherwise
- P: COM - unaffected; COMB - set if parity of the result is even; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

dst	Word/Byte NS	SS	SL
R	7	--	--
IR	12	--	--
DA	15	16	18
X	16	16	19

EXAMPLE: If register R1 contains %2552 (0010010101010010), the statement

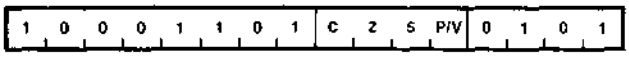
```
COM  R1
```

will leave the value %DAAD (1101101010101101) in R1.

COMFLG
Complement Flag

COMFLG flag flag: C, Z, S, P, V

INSTRUCTION FORMAT: (F9.1)



OPERATION: FLAGS(4:7) <- FLAGS(4:7) XOR instruction(4:7)

Any combination of the C, Z, S, P or V flags are complemented (each one bit is changed to zero, and vice-versa) if the corresponding bit in the instruction is one. If the corresponding bit in the instruction is zero, the flag will not be affected. All other bits in the FLAGS register are unaffected. Note that the P and V flags are represented by the same bit. There may be one, two, three or four operands in the assembly language statement, in any order.

FLAGS:
C: Complemented if specified; unaffected otherwise
Z: Complemented if specified; unaffected otherwise
S: Complemented if specified; unaffected otherwise
P/V: Complemented if specified; unaffected otherwise
D: Unaffected
H: Undefined

CYCLES: 7

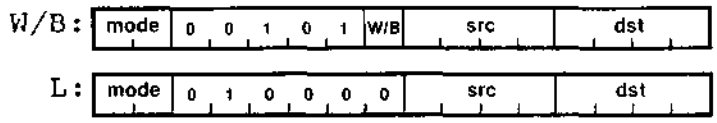
EXAMPLE: If the C, Z, and S flags are all clear (=0), and the P flag is set (=1), the statement

```
COMFLG  P, S, Z, C
```

will leave the C, Z, and S flags set (=1), and the P flag cleared (=0).

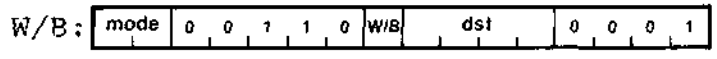
```
CP    dst,src                dst: R, IR, DA, X
CPB                          src: R, IM, IR, DA, X
CPL
```

INSTRUCTION FORMAT: (F2.1)



mode	dst	src
10	R	R
00	R	IM (src field=0)
00	R	IR (src field<>0)
01	R	DA (src field=0)
01	R	X (src field<>0)

(F5.1)



mode	dst	src
00	IR	IM
01	DA	IM (dst field=0)
01	X	IM (dst field<>0)

OPERATION: dst - src

The source operand is compared to (subtracted from) the destination operand, and the appropriate flags set accordingly, which may then be used for arithmetic and logical conditional jumps. Both operands are unaffected, with the only action being the setting of the flags. Subtraction is performed by adding the two's complement of the source operand to the destination operand.

FLAGS:
- C: Cleared if there is a carry from the most significant bit of the result; set otherwise, indicating a "borrow"
- Z: Set if the result is zero; cleared otherwise
- S: Set if the result is negative; cleared otherwise
- V: Set if arithmetic overflow occurs, that is, if both operands were of opposite signs and the sign of the result is the same as the sign of the source; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

dst	src	Word/Byte			Long		
		NS	SS	SL	NS	SS	SL
R	R	4	--	--	8	--	--
R	IM	7	--	--	14	--	--
R	IR	7	--	--	14	--	--
R	DA	9	10	12	15	16	18
R	X	10	10	13	16	16	19
IR	IM	11	--	--	--	--	--
DA	IM	14	15	17	--	--	--
X	IM	15	15	18	--	--	--

EXAMPLE: If register R5 contains %0400, the byte at location %0400 contains 2, and the source operand is the immediate value 3, the statement

```
CPB  @R5,#3
```

will leave the C flag set, indicating a borrow, the S flag set, and the Z and V flags cleared.

```
CPD     dst,src,r,cc        dst: R
CPDB                        src: IR
```

INSTRUCTION FORMAT: (F6.5)

dst	src	r
R	IR	R

OPERATION:
```
dst - src
AUTODECREMENT src {-1 if byte, -2 if word}
r <- r - 1
```

This instruction is used to search a string of data for an element meeting the specified condition. The contents of the location addressed by the source register is compared to (subtracted from) the destination operand, and the Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See section 3.2.1 for a list of condition codes. Both operands are unaffected.

The source register is then decremented by one if CPDB, or by two if CPD, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS:
- C: Undefined
- Z: Set if the condition code generated by the comparison matches cc; cleared otherwise
- S: Undefined
- V: Set if the result of decrementing r is zero; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte
20

EXAMPLE: If register RH0 contains %FF, register R1 contains %4001, the byte at location %4001 contains %00, and register R3 contains 5, the statement

```
CPDB  RH0,@R1,R3,EQ
```

will leave the Z flag cleared since the condition code would not have been "equal". Register R1 will contain the value %4000 and R3 will contain 4.

CPDR
Compare, Decrement and Repeat

```
CPDR  dst,src,r,cc          dst: R
CPDRB                       src: IR
```

INSTRUCTION FORMAT: (F6.5)

	dst	src	r
	R	IR	R

W/B:

OPERATION:
```
dst - src
AUTODECREMENT src {-1 if byte, -2 if word}
r <- r - 1
repeat until cc is true or r = 0
```

This instruction is used to search a string of data for an element meeting the specified condition. The contents of the location addressed by the source register is compared to (subtracted from) the destination operand, and the Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See section 3.2.1 for a list of condition codes. Both operands are unaffected.

The source register is then decremented by one if CPDRB, or by two if CPDR, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until either the condition is met or the result of decrementing r is zero. This instruction can search a string from 1 to 65536 bytes or 32768 words long (the value of r must not be greater than 32768 for CPDR).

FLAGS:
- C: Undefined
- Z: Set if the condition code generated by the comparison matches cc; cleared otherwise
- S: Undefined
- V: Set if the result of decrementing r is zero; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte	
11+9*n	(n=number of data elements compared)

NOTE: This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

EXAMPLE: If the string of words starting at location %2000 contain the values 0,2,4,6 and 8, register R2 contains %2008, R3 contains 3, and R8 contains 8, the statement

```
CPDR  R3,@R2,R8,GT
```

will leave the Z flag set indicating the condition was met. Register R2 will contain the value %2000, R3 will still contain 3, and R8 will contain 4.

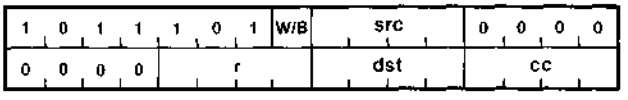
CPI
Compare and Increment

CPI dst,src,r,cc
CPIB

dst: R
src: IR

INSTRUCTION FORMAT: (F6.5)

W/B:



dst	src	r
R	IR	R

OPERATION:
dst - src
AUTOINCREMENT src {+1 if byte, +2 if word}
r <- r - 1

This instruction is used to search a string of data for an element meeting the specified condition. The contents of the location addressed by the source register is compared to (subtracted from) the destination operand, and the Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See section 3.2.1 for a list of condition codes. Both operands are unaffected.

The source register is then incremented by one if CPIB, or by two if CPI, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS:
C: Undefined
Z: Set if the condition code generated by the comparison matches cc; cleared otherwise
S: Undefined
V: Set if the result of decrementing r is zero; cleared otherwise
D: Unaffected
H: Unaffected

CYCLES:

Word/Byte
20

EXAMPLE: This instruction is used in a "loop" of instructions which searches a string of data for an element meeting the specified condition, but an intermediate operation on each data element is required. The following sequence "scans while numeric", that is, a string is searched until either an ASCII character not in the

range "0" to "9" (see Appendix) is found, or the end of the string is reached. This involves a range check on each character (byte) in the string.

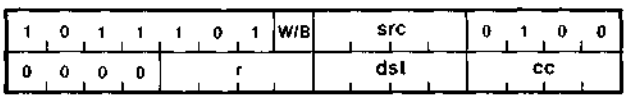
```
        LD      R3,#STRLEN          !Initialize counter!
        LD      R1,#STRSTART        !Load start address!
        LD      RL0,#'9'            !Largest numeric char!
LOOP:
        CPB     @R1,#'0'            !Test char < '0'!
        JR      ULT,NONNUMERIC
        CPIB    RL0,@R1,R3,ULT      !Test char > '9'!
        JR      Z,NONNUMERIC
        JR      NOV,LOOP            !Repeat until counter=0!
DONE:    .
         .
         .
NONNUMERIC:                         !Handle non-numeric char!
```

CPIR
Compare, Increment and Repeat

```
CPIR  dst,src,r,cc          dst: R
CPIRB                       src: IR
```

INSTRUCTION FORMAT: (F6.5)

W/B:



dst	src	r
R	IR	R

OPERATION:
```
dst - src
AUTOINCREMENT src {+1 if byte, +2 if word}
r <- r - 1
repeat until cc is true or r = 0
```

This instruction is used to search a string of data for an element meeting the specified condition. The contents of the location addressed by the source register is compared to (subtracted from) the destination operand, and the Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See section 3.2.1 for a list of condition codes. Both operands are unaffected.

The source register is then incremented by one if CPIRB, or by two if CPIR, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until either the condition is met or the result of decrementing r is zero. This instruction can search a string from 1 to 65536 bytes or 32768 words long (the value of r must not be greater than 32768 for CPIR).

FLAGS:
- C: Undefined
- Z: Set if the condition code generated by the comparison matches cc; cleared otherwise
- S: Undefined
- V: Set if the result of decrementing r is zero; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte	
11+9*n	(n=number of data elements compared)

NOTE: This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved

before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

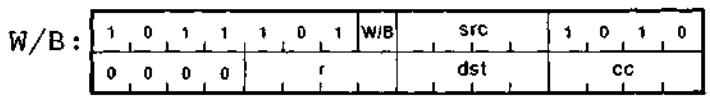
EXAMPLE: The following sequence of instructions can be used to search a string for an ASCII return character. The pointer to the start of the string is set, the string length is set, the character (byte) to be searched for is set, and then the search is accomplished. Testing the Z flag determines whether the character was found.

```
LD      R1,#STRSTART
LD      R3,#STRLEN
LD      RL0,#'%R'
CPIRB   RL0,@R1,R3,EQ
JR      Z,FOUND
```

CPSD
Compare String and Decrement

```
CPSD  dst,src,r,cc          dst: IR
CPSDB                       src: IR
```

INSTRUCTION FORMAT: (F6.5)



dst	src	r
IR	IR	R

OPERATION:
```
dst - src
AUTODECREMENT dst and src  {-1 if byte, -2 if word}
r <- r - 1
```

This instruction is used to compare two strings of data until the specified condition is true. The contents of the location addressed by the source register is compared to (subtracted from) the contents of the location addressed by the destination register. The Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See section 3.2.1 for a list of condition codes. Both operands are unaffected.

The source and destination registers are then decremented by one if CPSDB, or by two if CPSD, thus moving the pointers to the previous elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS:
- C: Undefined
- Z: Set if the condition code generated by the comparison matches cc; cleared otherwise
- S: Undefined
- V: Set if the result of decrementing r is zero; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte
25

EXAMPLE: If register R2 contains %2000, the byte at location %2000 contains %FF, register R3 contains %3000, the byte at location %3000 contains %00, and register R4 contains 1, the statement

```
CPSDB  @R2,@R3,R4,UGE
```

will leave the Z flag set since the condition code would have been "unsigned greater than or equal", and the V flag will be set to indicate that the counter R4 now contains 0. R2 will contain %1FFF, and R3 will contain %2FFF.

CPSDR
Compare String, Decrement and Repeat

CPSDR dst,src,r,cc
CPSDRB

dst: IR
src: IR

INSTRUCTION FORMAT: (F6.5)

dst	src	r
IR	IR	R

OPERATION:
```
dst - src
AUTODECREMENT dst and src  {-1 if byte, -2 if word}
r <- r - 1
repeat until cc is true or r = 0
```

This instruction is used to compare two strings of data until the specified condition is true. The contents of the location addressed by the source register is compared to (subtracted from) the contents of the location addressed by the destination register. The Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See section 3.2.1 for a list of condition codes. Both operands are unaffected.

The source and destination registers are then decremented by one if CPSDRB, or by two if CPSDR, thus moving the pointers to the previous elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until either the condition is met or the result of decrementing r is zero. This instruction can compare strings from 1 to 65536 bytes or 32768 words long (the value of r must not be greater than 32768 for CPSDR).

FLAGS:
- C: Undefined
- Z: Set if the condition code generated by the comparison matches cc; cleared otherwise
- S: Undefined
- V: Set if the result of decrementing r is zero; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte
11+14*n

(n=number of data elements compared)

NOTE: This instruction can be interrupted after each execution of the basic operation. The program counter of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

EXAMPLE: If the words from location %1000 to %1006 contain the values 0,2,4, and 6, the words from location %2000 to %2006 contain the values 0,1,1,0, register R13 contains %1006, register R14 contains %2006, and register R0 contains 4, the statement

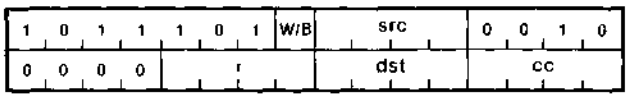
```
CPSDR  @R13,@R14,R0,EQ
```

leaves the Z flag set since the condition code would have been "equal" (locations %1000 and %2000 both contain the value 0). R13 will contain %0FFE, R14 will contain %1FFE, and R0 will contain 0.

CPSI
Compare String and Increment

```
CPSI  dst,src,r,cc          dst: IR
CPSIB                       src: IR
```

INSTRUCTION FORMAT: (F6.5)



dst	src	r
IR	IR	R

OPERATION:
```
dst - src
AUTOINCREMENT dst and src  {+1 if byte, +2 if word}
r <- r - 1
```

This instruction is used to compare two strings of data until the specified condition is true. The contents of the location addressed by the source register is compared to (subtracted from) the contents of the location addressed by the destination register. The Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See section 3.2.1 for a list of condition codes. Both operands are unaffected.

The source and destination registers are then incremented by one if CPSIB, or by two if CPSI, thus moving the pointers to the next elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS:
- C: Undefined
- Z: Set if the condition code generated by the comparison matches cc; cleared otherwise
- S: Undefined
- V: Set if the result of decrementing r is zero; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte
25

EXAMPLE: This instruction is used in a "loop" of instructions which compares two strings until the specified condition is true, but where an intermediate operation on each data element is required. The following sequence attempts to match a given source string to the destination string which is known to contain all upper-case characters. The match should succeed even if the source

string contains some lower-case characters. This involves a forced conversion of the source string to upper-case (only ASCII alphabetic letters are assumed, see Appendix) by resetting bit 5 of each character (byte) before comparison.

```
        LD      R1,#SRCSTART    !Load start addresses!
        LD      R2,#DSTSTART
        LD      R3,#STRLEN      !Initialize counter!
LOOP:
        RESB    @R1,#5          !Force upper-case!
        CPSIB   @R1,@R2,R3,NE   !Compare until not equal!
        JR      Z,NOTEQUAL      !Exit loop if match fails!
        JR      NOV,LOOP        !Repeat until counter=0!
DONE:    .                      !Match succeeds!
         .
         .
NOTEQUAL:                       !Match fails!
```

CPSIR
Compare String, Increment and Repeat

```
CPSIR  dst,src,r,cc          dst: IR
CPSIRB                       src: IR
```

INSTRUCTION FORMAT: (F6.5)

W/B:

dst	src	r
IR	IR	R

OPERATION:
```
dst - src
AUTOINCREMENT dst and src  {+1 if byte, +2 if word}
r <- r - 1
repeat until cc is true or r = 0
```

This instruction is used to compare two strings of data until the specified condition is true. The contents of the location addressed by the source register is compared to (subtracted from) the contents of the location addressed by the destination register. The Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See section 3.2.1 for a list of condition codes. Both operands are unaffected.

The source and destination registers are then incremented by one if CPSIRB, or by two if CPSIR, thus moving the pointers to the next elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until either the condition is met or the result of decrementing r is zero. This instruction can compare strings from 1 to 65536 bytes or 32768 words long (the value of r must not be greater than 32768 for CPSIR).

FLAGS:
- C: Undefined
- Z: Set if the condition code generated by the comparison matches cc; cleared otherwise
- S: Undefined
- V: Set if the result of decrementing r is zero; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte
11+14*n (n=number of data elements compared)

NOTE: This instruction can be interrupted after each execution of the basic operation. The program counter of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

EXAMPLE: The following sequence of instructions can be used to compare two strings of possibly different lengths, but at least one character (byte) long. It determines if the destination string is lexicographically greater than or equal to the source string as determined by the ordering of the ASCII character set (see Appendix). Notice that the string "ABC" is greater than "AB", and "AC" is greater than "ABC". The destination string is initially pointed to by R1, with its length in bytes in R3. The source string is initially pointed to by R2, with its length in bytes in R4.

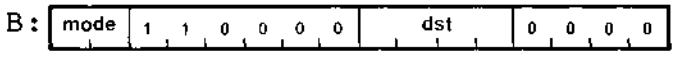
```
        CP      R3,R4               !Find shortest length!
        JR      ULE,COMPARE
        LD      R3,R4               !If source is shorter,!
        EX      R1,R2               !then swap pointers!
COMPARE:
        CPSIRB  @R1,@R2,R3,ULT      !Compare while >=!
        JR      Z,FAIL
SUCCEED:        .                   !Destination >= source!
                .
                .
FAIL:                               !Destination < source!
```

DAB
Decimal Adjust

Decimal Adjust

DAB dst dst: R

INSTRUCTION FORMAT: (Fl.1)



mode	dst
10	R

OPERATION: dst <- DA dst

The destination byte is adjusted to form two 4-bit BCD digits following an addition or subtraction operation. For addition (ADDB,ADCB) or subtraction (SUBB,SBCB), the following table indicates the operation performed:

Instruction	Carry Before DAB	Bits 4-7 Value (Hex)	H Flag Before DAB	Bits 0-3 Value (Hex)	Number Added To Byte	Carry After DAB
ADDB ADCB	0	0-9	0	0-9	00	0
	0	0-8	0	A-F	06	0
	0	0-9	1	0-3	06	0
	0	A-F	0	0-9	60	1
	0	9-F	0	A-F	66	1
	0	A-F	1	0-3	66	1
	1	0-2	0	0-9	60	1
	1	0-2	0	A-F	66	1
SUBB SBCB	0	0-9	0	0-9	00	0
	0	0-8	1	6-F	FA	0
	1	7-F	0	0-9	A0	1
	1	6-F	1	6-F	9A	1

The operation is undefined if the destination byte was not the result of a valid addition or subtraction of BCD digits.

FLAGS:
- C: Set or cleared according to the table above
- Z: Set if the result is zero; cleared otherwise
- S: Set if the most significant bit of the result is set; cleared otherwise
- V: Unaffected
- D: Unaffected
- H: Unaffected

CYCLES:

Byte
5

EXAMPLE: If addition is performed using the BCD values 15 and 27, the result should be 42. The sum is incorrect, however, when the binary representations are added in the destination location using standard binary arithmetic.

```
  0001  0101
+ 0010  0111
  ----  ----
  0011  1100 = %3C
```

The DAB instruction adjusts this result so that the correct BCD representation is obtained.

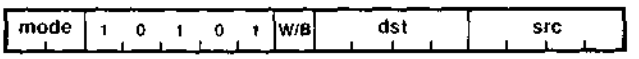
```
  0011  1100
+ 0000  0110
  ----  ----
  0100  0010 = 42
```

DEC
Decrement

```
DEC   dst,src                    dst: R, IR, DA, X
DECB                             src: IM
```

INSTRUCTION FORMAT: (F1.2)

W/B:



mode	dst	src	
10	R	IM	
00	IR	IM	
01	DA	IM	(dst field=0)
01	X	IM	(dst field<>0)

OPERATION: dst <- dst - src {src is 1 to 16}

The source operand (a value from 1 to 16) is subtracted from the destination operand and the result is stored in the destination. Subtraction is performed by adding the two's complement of the source operand to the destination operand. The source operand may be omitted from the assembly language statement and thus defaults to the value 1.

FLAGS:
- C: Unaffected
- Z: Set if the result is zero; cleared otherwise
- S: Set if the result is negative; cleared otherwise
- V: Set if arithmetic overflow occurs, that is, the sign of the destination was opposite the sign of the result; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

dst	Word/Byte		
	NS	SS	SL
R	4	--	--
IR	11	--	--
DA	13	14	16
X	14	14	17

NOTE: The "src field" in the instruction format encoding contains the source operand. The "src field" values range from 0 to 15 corresponding to the source values 1 to 16.

EXAMPLE: If register R10 contains %002A, the statement

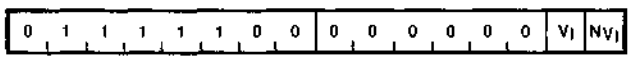
```
DEC  R10
```

will leave the value %0029 in R10.

DI
Disable Interrupt

DI int int: VI, NVI

INSTRUCTION FORMAT: (F9.2)



OPERATION: if instruction(0) = 0 then NVI <- 0
if instruction(1) = 0 then VI <- 0

Any combination of the Vectored Interrupt (VI) or Non-Vectored Interrupt (NVI) control bits in the Flags and Control Word (FCW) are cleared to zero if the corresponding bit in the instruction is zero, thus disabling the appropriate type of interrupt. If the corresponding bit in the instruction is one, the control bit will not be affected. All other bits in the FCW are not affected. There may be one or two operands in the assembly language statement, in either order.

FLAGS: No flags affected

CYCLES: 7

NOTE: This is a privileged instruction.

EXAMPLE: If both the NVI and VI control bits are set (=1), the statement

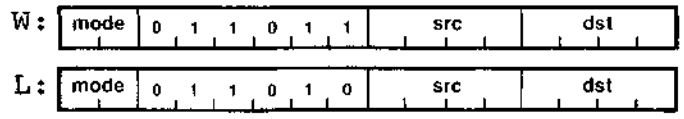
```
DI  VI
```

will leave the NVI control bit set (=1), and the VI control bit cleared (=0).

DIV
Divide

```
DIV   dst,src                dst: R
DIVL                         src: R, IM, IR, DA, X
```

INSTRUCTION FORMAT: (F2.1)



mode	dst	src
10	R	R
00	R	IM (src field=0)
00	R	IR (src field<>0)
01	R	DA (src field=0)
01	R	X (src field<>0)

OPERATION:

Word
dst,dst∨1 <- dst,dst∨1 / src
{dst <- remainder
dst∨1 <- quotient}

Long
dst,dst∨1,dst∨2,dst∨3 <- dst,dst∨1,dst∨2,dst∨3 / src,src∨1
{dst,dst∨1 <- remainder
dst∨2,dst∨3 <- quotient}

The destination operand (dividend) is divided by the source operand (divisor), the quotient is stored in the low-order half of the destination and the remainder is stored in the high-order half of the destination. The contents of the source are not affected. Both operands are treated as signed, two's complement integers and division is performed so that the remainder is of the same sign as the dividend. For DIV, the destination is a register pair and the source is a word value; for DIVL, the destination is a register quadruple and the source is a long word value.

FLAGS:
- C: DIV - set if quotient is less than -2^{15} or not less than 2^{15}; cleared otherwise; DIVL - set if quotient is less than -2^{31} or not less than 2^{31}; cleared otherwise
- Z: Set if the quotient or divisor is zero; cleared otherwise
- S: Set if the quotient is negative; cleared otherwise (contains the sign of the divisor if the instruction is aborted)
- V: Set if the divisor is zero or if the quotient would be too large to fit in the low-order half of the destination operand (the instruction is aborted if the absolute value of the high-order half of the dividend is larger than the absolute value of the divisor); cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

src	Word			Long		
	NS	SS	SL	NS	SS	SL
R	95	--	--	723	--	--
IM	95	--	--	723	--	--
IR	95	--	--	723	--	--
DA	96	97	99	724	725	727
X	97	97	100	725	725	728

(Divisor is zero)

R	13	--	--	30	--	--
IM	13	--	--	30	--	--
IR	13	--	--	30	--	--
DA	14	15	17	31	32	34
X	15	15	18	32	32	35

(Absolute value of the high-order half of the dividend is larger than the absolute value of the divisor)

R	25	--	--	51	--	--
IM	25	--	--	51	--	--
IR	25	--	--	51	--	--
DA	26	27	28	52	53	55
X	27	27	29	53	53	56

NOTE: For proper instruction execution, the "dst field" in the instruction format encoding must be even for DIV, and must be a multiple of 4 (0,4,8,12) for DIVL. If the source operand in DIVL is a register, the "src field" must be even.

EXAMPLE: If register RR0 (composed of word registers R0 and R1) contains %00000022 and register R3 contains 6, the statement

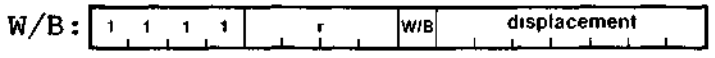
```
DIV  RR0,R3
```

will leave the value %00040005 in RR0 (R1 contains the quotient 5 and R0 contains the remainder 4).

DJNZ
Decrement and Jump if Not Zero

DJNZ r,dst dst: RA
DBJNZ

INSTRUCTION FORMAT: (F3.1)



r	dst
R	RA

OPERATION:
```
r <- r - 1
if r <> 0 then PC <- PC - (2*disp)
```

The register being used as a counter is decremented. If the contents of the register are not zero after decrementing, the destination address is calculated and then loaded into the program counter (PC). Control will then pass to the instruction whose address is pointed to by the PC. When the register counter reaches zero, control falls through to the instruction following DJNZ or DBJNZ. This instruction provides a simple, efficient method of loop control.

FLAGS: No flags affected

CYCLES:

Word/Byte
11

NOTE: The relative addressing mode is calculated by doubling the displacement in the instruction, then subtracting this value from the updated value of the PC to derive the destination address. The updated PC value is taken to be the address of the instruction byte following the DJNZ or DBJNZ instruction, while the displacement is a 7-bit positive value in the range 0 to 127. Thus, the destination address must be in the range -252 to 2 bytes from the start of the DJNZ or DBJNZ instruction. In the segmented mode, the PC segment number is not affected. The assembler automatically calculates the displacement by subtracting the PC value of the following instruction from the address given by the programmer. Note that DJNZ or DBJNZ can not be used to transfer control in the forward direction.

EXAMPLE: DJNZ and DBJNZ are typically used to control a "loop" of instructions. In this example, 100 bytes are moved from one buffer area to another, and the sign bit of

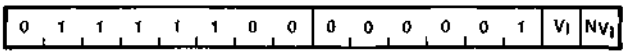
each byte is cleared to zero. Register RH0 is used as the counter.

```
     LDB     RH0,#100      !Initialize counter!
     LD      R1,#SRCBUF    !Load start addresses!
     LD      R2,#DSTBUF
LOOP:
     LDB     RL0,@R1       !Load source byte!
     RESB    RL0,#7        !Mask off sign bit!
     LDB     @R2,RL0       !Store into destination!
     INC     R1            !Advance pointers!
     INC     R2
     DBJNZ   RH0,LOOP      !Repeat until counter=0!
NEXT:
```

EI
Enable Interrupt

EI int int: VI, NVI

INSTRUCTION FORMAT: (F9.2)



OPERATION: if instruction(0) = 0 then NVI <- 1
if instruction(1) = 0 then VI <- 1

Any combination of the Vectored Interrupt (VI) or Non-Vectored Interrupt (NVI) control bits in the Flags and Control Word (FCW) are set to one if the corresponding bit in the instruction is zero, thus enabling the appropriate type of interrupt. If the corresponding bit in the instruction is one, the control bit will not be affected. All other bits in the FCW are not affected. There may be one or two operands in the assembly language statement, in either order.

FLAGS: No flags affected

CYCLES: 7

NOTE: This is a privileged instruction.

EXAMPLE: If the NVI control bit is set (=1), and the VI control bit is clear (=0), the statement

```
EI  NVI,VI
```

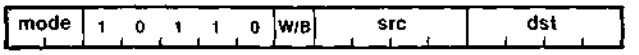
will leave both the NVI and VI control bits set (=1).

EX dst,src
EXB

dst: R
src: R, IR, DA, X

INSTRUCTION FORMAT: (F2.1)

W/B:



mode	dst	src
10	R	R
00	R	IR
01	R	DA (src field=0)
01	R	X (src field<>0)

OPERATION:
```
tmp <- src  {tmp is an internal register}
src <- dst
dst <- tmp
```

The contents of the source operand are exchanged with the contents of the destination operand.

FLAGS: No flags affected

CYCLES:

src	Word/Byte NS	SS	SL
R	6	--	--
IR	12	--	--
DA	15	16	18
X	16	16	19

EXAMPLE: If register R0 contains 8 and register R5 contains 9, the statement

```
EX  R0,R5
```

will leave the values 9 in R0, and 8 in R5.

EXTS
Extend Sign

```
EXTS   dst                          dst: R
EXTSB
EXTSL
```

INSTRUCTION FORMAT: (F1.1)

	mode							dst				
B:	mode	1	1	0	0	0	1	dst	0	0	0	0
W:	mode	1	1	0	0	0	1	dst	1	0	1	0
L:	mode	1	1	0	0	0	1	dst	0	1	1	1

mode	dst
10	R

OPERATION: Byte

```
if dst(7)=0 then dst(8:15) <- 000...000
            else dst(8:15) <- 111...111
```

Word

```
if dstV1(15)=0 then dst(16:31) <- 000...000
               else dst(16:31) <- 111...111
```

Long

```
if dstV2,dstV3(31)=0 then dst,dstV1(32:63) <- 000...000
                     else dst,dstV1(32:63) <- 111...111
```

The sign bit of the low-order half of the destination operand is copied into all bit positions of the high-order half of the destination. For EXTSB, the destination is a word register; for EXTS, the destination is a register pair; for EXTSL, the destination is a register quadruple. This instruction is useful in multiple precision arithmetic or for conversion of small signed operands to larger signed operands.

FLAGS: No flags affected

CYCLES:

Word/Byte	Long
11	11

NOTE: For proper instruction execution, the "dst field" in the instruction format encoding must be even for EXTS, and must be a multiple of 4 (0,4,8,12) for EXTSL.

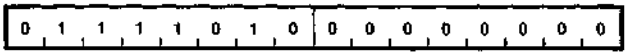
EXAMPLE: If register pair RR2 (composed of word registers R2 and R3) contains %12345678, the statement

```
    EXTS  RR2
```

will leave the value %00005678 in RR2 (because the sign bit of R3 was 0).

HALT

INSTRUCTION FORMAT: (F9.3)



OPERATION:

The CPU operation is suspended until an interrupt or reset request is received. This instruction is used to synchronize the Z8000 with external events, preserving its state until an interrupt or reset request is honored. After an interrupt is serviced, control falls through to the instruction following the HALT.

FLAGS: No flags affected

CYCLES: 8+3*n (Interrupts are recognized at the end of each 3-cycle period; thus n=number of periods without interruption)

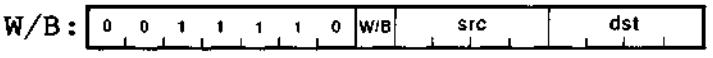
NOTES: This is a privileged instruction.

While in the halt state, memory refresh cycles will still occur, and $\overline{\text{BUSRQ}}$ will be honored.

IN
Input

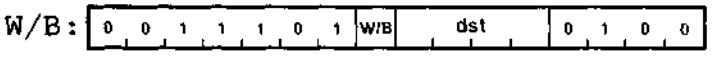
```
IN     dst,src                    dst: R
INB                               src: IR, DA
```

INSTRUCTION FORMAT: (F7.3)



dst	src
R	IR

(F7.1)



dst	src
R	DA

OPERATION: dst <- src

The contents of the source operand are loaded into the destination. I/O addresses are always 16 bits.

FLAGS: No flags affected

CYCLES:

src	Word/Byte
IR	10
DA	12

NOTE: This is a privileged instruction.

EXAMPLE: If register R6 contains the I/O address %0123, and the "port" %0123 contains %FF, the statement

```
INB  RH2,@R6
```

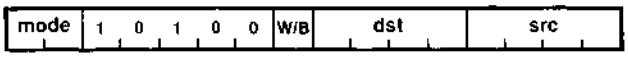
will leave the value %FF in register RH2.

INC
Increment

```
INC   dst,src                dst: R, IR, DA, X
INCB                         src: IM
```

INSTRUCTION FORMAT: (Fl.2)

W/B:



mode	dst	src
10	R	IM
00	IR	IM
01	DA	IM (dst field=0)
01	X	IM (dst field<>0)

OPERATION: dst <- dst + src {src is 1 to 16}

The source operand (a value from 1 to 16) is added to the destination operand and the sum is stored in the destination. Two's complement addition is performed. The source operand may be omitted from the assembly language statement and thus defaults to the value 1.

FLAGS:
- C: Unaffected
- Z: Set if the result is zero; cleared otherwise
- S: Set if the result is negative; cleared otherwise
- V: Set if arithmetic overflow occurs, that is, the sign of the destination was opposite the sign of the result; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

dst	Word/Byte		
	NS	SS	SL
R	4	--	--
IR	11	--	--
DA	13	14	16
X	14	14	17

NOTE: The "src field" in the instruction format encoding contains the source operand. The "src field" values range from 0 to 15 corresponding to the source values 1 to 16.

EXAMPLE: If register RH2 contains %21, the statement

```
INCB  RH2,#6
```

will leave the value %27 in RH2.

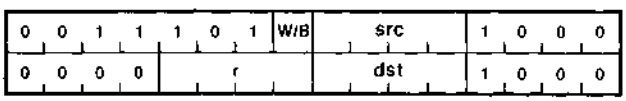
IND
Input and Decrement

```
IND  dst,src,r                    dst: IR
INDB                              src: IR
```

INSTRUCTION FORMAT: (F6.4)

	dst	src	r
	IR	IR	R

W/B:



OPERATION:
```
dst <- src
AUTODECREMENT dst {-1 if byte, -2 if word}
r <- r - 1
```

This instruction is used for block input of strings of data. The contents of the I/O location addressed by the source register are loaded into the memory location addressed by the destination register. I/O addresses are always 16 bits. The destination register is then decremented by one if INDB, or by two if IND, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS:
- C: Unaffected
- Z: Undefined
- S: Unaffected
- V: Set if the result of decrementing r is zero; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte
21

NOTE: This is a privileged instruction.

EXAMPLE: In segmented mode, if register RR4 contains %02004000 (segment 2, offset %4000), register R6 contains the I/O address %0228, the "port" %0228 contains %05B9, and register R0 contains %0016, the statement

```
IND  @RR4,@R6,R0
```

will leave the value %05B9 in location %02004000, the value %02003FFE in RR4, and the value %0015 in R0. Register R6 still contains the value %0228.

INDR
Input, Decrement and Repeat

```
INDR  dst,src,r                dst: IR
INDRB                          src: IR
```

INSTRUCTION FORMAT: (F6.4)

	dst	src	r
	IR	IR	R

W/B:

OPERATION:
```
dst <- src
AUTODECREMENT dst {-1 if byte, -2 if word}
r <- r - 1
repeat until r = 0
```

This instruction is used for block input of strings of data. The contents of the I/O location addressed by the source register are loaded into the memory location addressed by the destination register. I/O addresses are always 16 bits. The destination register is then decremented by one if INDRB, or by two if INDR, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can input from 1 to 65536 bytes or 32768 words (the value for r must not be greater than 32768 for INDR).

FLAGS:
C: Unaffected
Z: Undefined
S: Unaffected
V: Set
D: Unaffected
H: Unaffected

CYCLES:

Word/Byte
11+10*n

(n=number of data elements transferred)

NOTES: This is a privileged instruction.

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

INDR
Input, Decrement and Repeat

EXAMPLE: If register R1 contains %202A, register R2 contains the I/O address %0050, and register R3 contains 8, the statement

```
INDRB  @R1,@R2,R3
```

will input 8 bytes from "port" %0050 and leave them in descending order from %202A to %2023. Register R1 will contain %2022, and R3 will contain 0. R2 will not be affected.

```
INI     dst,src,r                dst: IR
INIB                             src: IR
```

INSTRUCTION FORMAT: (F6.4)

W/B:

dst	src	r
IR	IR	R

OPERATION:

```
dst <- src
AUTOINCREMENT dst {+1 if byte, +2 if word}
r <- r - 1
```

This instruction is used for block input of strings of data. The contents of the I/O location addressed by the source register are loaded into the memory location addressed by the destination register. I/O addresses are always 16 bits. The destination register is then incremented by one if INIB, or by two if INI, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS:
- C: Unaffected
- Z: Undefined
- S: Unaffected
- V: Set if the result of decrementing r is zero; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte
21

NOTE: This is a privileged instruction.

EXAMPLE: If register R4 contains %4000, register Ro contains the I/O address %0228, the "port" %0228 contains %B9, and register R0 contains %0016, the statement

```
INIB  @R4,@R6,R0
```

will leave the value %B9 in location %4000, the value %4001 in R4, and the value %0015 in R0. Register R6 still contains the value %0228.

INIR
Input, Increment and Repeat

```
INIR   dst,src,r          dst: IR
INIRB                     src: IR
```

INSTRUCTION FORMAT: (F6.4)

	dst	src	r
	IR	IR	R

OPERATION:
```
dst <- src
AUTOINCREMENT dst {+1 if byte, +2 if word}
r <- r - 1
repeat until r = 0
```

This instruction is used for block input of strings of data. The contents of the I/O location addressed by the source register are loaded into the memory location addressed by the destination register. I/O addresses are always 16 bits. The destination register is then incremented by one if INIRB, or by two if INIR, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can input from 1 to 65536 bytes or 32768 words (the value for r must not be greater than 32768 for INIR).

FLAGS:
- C: Unaffected
- Z: Undefined
- S: Unaffected
- V: Set
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte
11+10*n

(n=number of data elements transferred)

NOTES: This is a privileged instruction.

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

EXAMPLE: If register R1 contains %2023, register R2 contains the I/O address %0050, and register R3 contains 8, the statement

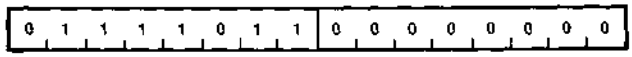
```
INIRB  @R1,@R2,R3
```

will input 8 bytes from "port" %0050 and leave them in ascending order from %2023 to %202A. Register R1 will contain %202B, and R3 will contain 0. R2 will not be affected.

IRET
Interrupt Return

IRET

INSTRUCTION FORMAT: (F9.3)



OPERATION:

Nonsegmented	Segmented
SP <- SP + 2	SP <- SP + 2 {Pop "identifier"}
PS <- @SP	PS <- @SP
SP <- SP + 4	SP <- SP + 6

This instruction is used to return to a previously executing procedure at the end of a procedure entered by an interrupt or trap (including a System Call instruction). First, the "identifier" word associated with the interrupt or trap is popped from the system processor stack and discarded. Then contents of the the location addressed by the system processor stack pointer are popped into the program status (PS), loading the Flags and Control Word (FCW) and the program counter (PC). The new value of the FCW is not effective until the next instruction, so that the status pins will not be affected by the new control bits until after the IRET instruction execution is completed. The next instruction executed is that addressed by the new contents of the PC. The system stack pointer (R15 if nonsegmented, or RR14 if segmented) is used.

FLAGS:
C: Loaded from processor stack
Z: Loaded from processor stack
S: Loaded from processor stack
P/V: Loaded from processor stack
D: Loaded from processor stack
H: Loaded from processor stack

CYCLES:

Address		
NS	SS	SL
13	--	16

NOTES: This is a privileged instruction.

The Z8001 version always executes the segmented mode of the IRET instruction.

EXAMPLE: In the nonsegmented Z8002 version, if the program counter contains %2550, the system stack pointer (R15) contains %3000, and locations %3000, %3002 and %3004 contain %7F03, a saved FCW value, and %1004, respectively, the statement

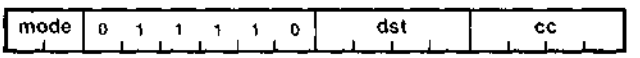
```
IRET
```

will leave the value %3006 in the system stack pointer and the program counter will contain %1004, the address of the next instruction to be executed. The program status is now determined by the saved FCW value.

JP
Jump

JP cc,dst dst: IR, DA, X

INSTRUCTION FORMAT: (F1.3)



mode	dst
00	IR
01	DA (dst field=0)
01	X (dst field<>0)

OPERATION: if cc is true then PC <- dst

A conditional jump transfers program control to the destination address if the condition specified by "cc" is true. See section 3.2.1 for a list of condition codes. If the condition code is true, the program counter (PC) is loaded with the designated address; otherwise, control falls through to the instruction following the JP instruction.

FLAGS: No flags affected

CYCLES:

dst	(jump is taken) Address			(jump not taken) Address		
	NS	SS	SL	NS	SS	SL
IR	10	--	15	7	--	7
DA	7	8	10	7	8	10
X	8	8	11	8	8	11

NOTE: The destination address must be even since instructions are word data.

EXAMPLE: If the carry flag is set, the statement

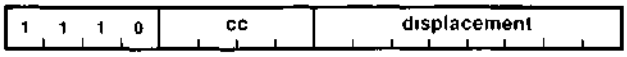
```
JP  C,%1520
```

replaces the contents of the program counter with %1520, thus transferring control to that location.

JR
Jump Relative

JR cc,dst dst: RA

INSTRUCTION FORMAT: (F3.3)



dst
RA

OPERATION: if cc is true then PC <- PC + (2*disp)

A conditional jump transfers program control to the destination address if the condition specified by "cc" is true. See section 3.2.1 for a list of condition codes. If the condition code is true, the program counter (PC) is loaded with the designated address; otherwise, control falls through to the instruction following the JR instruction.

FLAGS: No flags affected

CYCLES:

Address	
6	(jump is taken)
6	(jump not taken)

NOTE: The relative addressing mode is calculated by doubling the displacement in the instruction, then adding this value to the updated value of the PC to derive the destination address. The updated PC value is taken to be the address of the instruction byte following the JR instruction, while the displacement is a 8-bit signed value in the range -128 to +127. Thus, the destination address must be in the range -254 to +256 bytes from the start of the JR instruction. In the segmented mode, the PC segment number is not affected. The assembler automatically calculates the displacement by subtracting the PC value of the following instruction from the address given by the programmer.

EXAMPLES: If the result of the last arithmetic operation executed is negative, the following four statements (which occupy a total of twelve bytes) are to be skipped. This can be accomplished with the statement

```
JR  MI,$+14
```

If the S flag is not set, execution continues with the statement following the JR.

JR
Jump Relative

A byte-saving form of a jump to the label LAB is

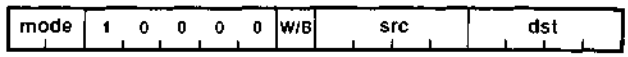
```
JR  LAB
```

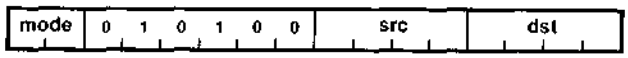
where LAB must be within the allowed range. The condition code is "blank" in this case, and indicates that the jump is always taken.

LD dst,src
LDB
LDL

dst: R, IR, DA, X, BA, BX
src: R, IM, IR, DA, X, BA, BX

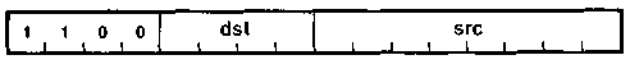
INSTRUCTION FORMAT: (F2.1)

W/B: 

L: 

mode	dst	src
10	R	R
00	R	IM*(src field=0)
00	R	IR (src field<>0)
01	R	DA (src field=0)
01	R	X (src field<>0)

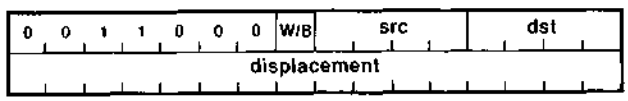
(F3.2)

B: 

dst	src
R	IM*

*See Note

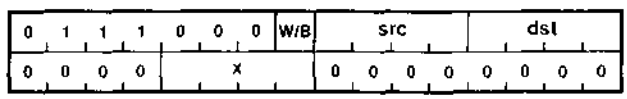
(F4.4)

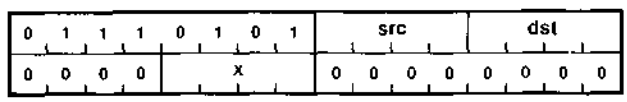
W/B: 

L: 

dst	src	
R	BA	(src field<>0)

(F4.3)

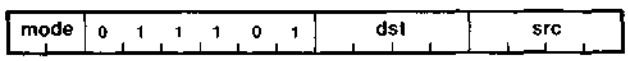
W/B: 

L: 

dst	src	
R	BX	(src field<>0)

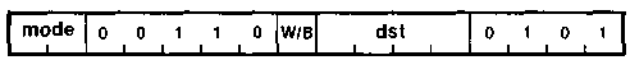
(F2.2)

W/B: 

L: 

mode	dst	src
00	IR	R
01	DA	R (dst field=0)
01	X	R (dst field<>0)

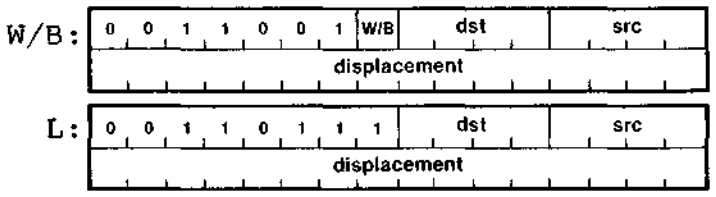
(F5.1)

W/B: 

mode	dst	src
00	IR	IM
01	DA	IM (dst field=0)
01	X	IM (dst field<>0)

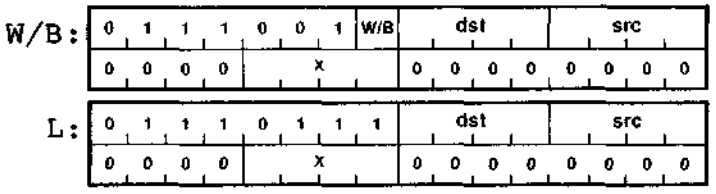
LD
Load

(F4.2)



dst	src	
BA	R	(dst field<>0)

(F4.1)



dst	src	
BX	R	(dst field<>0)

OPERATION: dst <- src

The contents of the source operand are loaded into the destination. The contents of the source are not affected.

FLAGS: No flags affected

CYCLES:

dst	src	Word/Byte NS	Word/Byte SS	Word/Byte SL	Long NS	Long SS	Long SL
R	R	3	--	--	5	--	--
R	IM	7	--	--	11	--	--
R	IM	5	(byte only - see Note)				
R	IR	7	--	--	11	--	--
R	DA	9	10	12	12	13	15
R	X	10	10	13	13	13	16
R	BA	14	--	--	17	--	--
R	BX	14	--	--	17	--	--
IR	R	8	--	--	11	--	--
DA	R	11	12	14	14	15	17
X	R	12	12	15	15	15	18
IR	IM	11	--	--	--	--	--
DA	IM	14	15	17	--	--	--
X	IM	15	15	18	--	--	--
BA	R	14	--	--	17	--	--
BX	R	14	--	--	17	--	--

NOTE: Although both formats F2.1 and F3.2 exist for "LDB R,IM", the assembler always uses the short format, F3.2. In this case, the "src field" in the instruction format encoding contains the source operand.

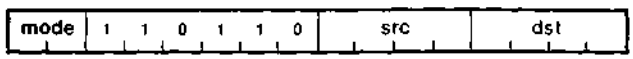
```
EXAMPLES:  LD   R0,R1          !Register R0 is loaded with the
                                contents of register R1!

           LD   R0,#20         !Register R0 is loaded with the value
                                20!

           LD   R0,20          !Register R0 is loaded with the con-
                                tents of the word at location 20!

           LD   R0,@R1         !If register R1 contains 20, then
                                register R0 is loaded with the con-
                                tents of the word at location 20!

           LDB  @R1,RH0        !If register R1 contains 20, then the
                                byte at location 20 is loaded with
                                the contents of register RH0!

           LD   R4,8(R1)       !If register R1 contains 20, then
                                register R4 is loaded with the con-
                                tents of the word at location 28
                                (addressed by the sum of the address 8
                                and the contents of the index register
                                R1)!

           LDL  RR4,R1(#8)     !If register R1 contains 20, then
                                register RR4 is loaded with the con-
                                tents of the long word at location 28
                                (addressed by the sum of the base
                                address contained in register R1 and
                                the displacement 8)!

           LD   R1(R2),R4      !If register R1 contains 20 and regis-
                                ter R2 contains 8, then the word at
                                location 28 (addressed by the sum of
                                the base address contained in R1 and
                                the contents of the index register R2)
                                is loaded with the contents of register
                                R4!
```

LDA
Load Address

LDA dst,src

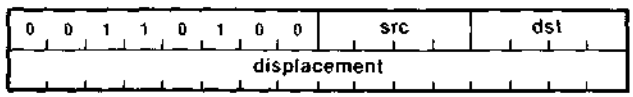
dst: R
src: DA, X, BA, BX

INSTRUCTION FORMAT: (F2.1)



mode	dst	src	
01	R	DA	(src field=0)
01	R	X	(src field<>0)

(F4.4)



dst	src	
R	BA	(src field<>0)

(F4.3)

dst	src	
R	BX	(src field<>0)

OPERATION: dst <- ADDRESS src

The address of the source operand is computed and loaded into the destination. The contents of the source are not affected. The address computation follows the rules for addressing mode arithmetic. The destination is a word register in nonsegmented mode, and a register pair in segmented mode (the segmented address loaded into the destination has an undefined value in all "reserved" bits).

FLAGS: No flags affected

CYCLES:

src	Address NS	SS	SL
DA	12	13	15
X	13	13	16
BA	15	--	--
BX	15	--	--

EXAMPLES:

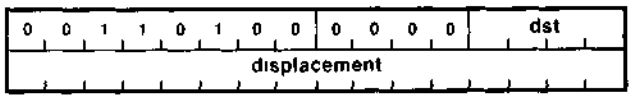
```
LDA   R4,STRUCT      !In nonsegmented mode, register R4 is
                      loaded with the nonsegmented address
                      of the location named STRUCT!

LDA   RR2,8(R4)      !In segmented mode, if index register
                      R4 contains %0020, then register RR2
                      is loaded with the segmented address
                      %00000028 (segment 0, offset %0028)!

LDA   RR2,RR4(#8)    !In segmented mode, if base register
                      RR4 contains %01000020, then reg-
                      ister RR2 is loaded with the seg-
                      mented address %01000028 (segment 1,
                      offset %0028)!
```

LDAR
Load Address Relative

LDAR dst,src

dst: R
src: RA

INSTRUCTION FORMAT: (F4.4)



dst	src
R	RA

OPERATION: dst <- ADDRESS src

The address of the source operand is computed and loaded into the destination. The contents of the source are not affected. The destination is a word register in nonsegmented mode, and a register pair in segmented mode (the segmented address loaded into the destination has all "reserved" bits cleared to zero).

FLAGS: No flags affected

CYCLES:

Address
15

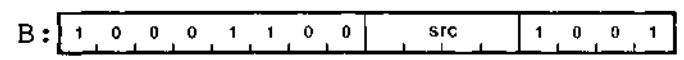
NOTE: The relative addressing mode is calculated by adding the displacement in the instruction to the updated value of the program counter (PC) to derive the address. The updated PC value is taken to be the address of the instruction byte following the LDAR instruction, while the displacement is a 16-bit signed value in the range -32768 to +32767. The assembler automatically calculates the displacement by subtracting the PC value of the following instruction from the address given by the programmer.

```
EXAMPLE:  LDAR  R2,DATA        !Register R2 is loaded with the
                                nonsegmented address of the
                                location named DATA!
```

LDCTL
Load Control Register

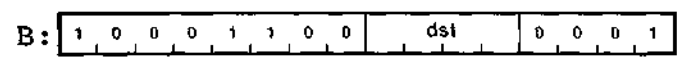
```
LDCTL  dst,src                dst: R, CTLR
LDCTLB                        src: R, CTLR
```

INSTRUCTION FORMAT: (F8.1)



dst: CTLR src: R

(F8.2)



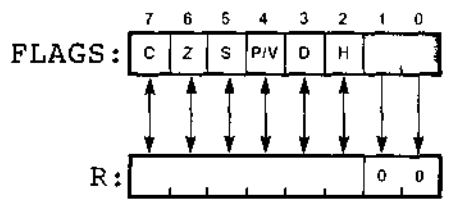
dst: R src: CTLR

OPERATION: LDCTLB FLAGS,R
FLAGS(2:7) <- src(2:7)

The contents of the source (a byte register) are loaded into the FLAGS register. The lower two bits of the FLAGS register and the entire source register are unaffected.

LDCTLB R,FLAGS
dst(2:7) <- FLAGS(2:7)
dst(0:1) <- 0

The contents of the upper six bits of the FLAGS register are loaded into the destination (a byte register). The lower two bits of the destination register are cleared to zero. The FLAGS register is unaffected.



Shaded areas are reserved

FLAGS: LDCTLB FLAGS,R
C: Loaded from source
Z: Loaded from source
S: Loaded from source
P/V: Loaded from source
D: Loaded from source
H: Loaded from source

LDCTLB R,FLAGS
No flags affected

CYCLES: 7

NOTE: This is the only Load Control Register instruction which is not privileged, thus it can be used in Normal or System mode.

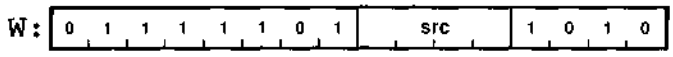
EXAMPLE: If the C, S and H flags are set (=1), all other flags are clear (=0), the statement

```
LDCTLB  RH0,FLAGS
```

will leave the value %A4 (10100100) in RH0.

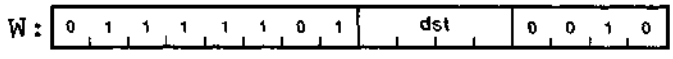
LDCTL
Load Control Register

INSTRUCTION FORMAT: (F8.1)



dst CTLR src R

(F8.2)



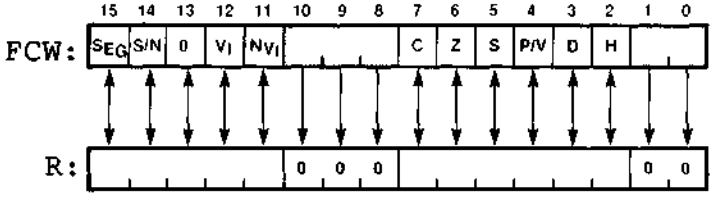
dst R src CTLR

OPERATION: LDCTL FCW,R
FCW(2:7) <- src(2:7)
FCW(11:15) <- src(11:15)

The contents of the source (a word register) are loaded into the Flags and Control Word (FCW) register. Bits 0, 1, 8, 9 and 10 of the FCW register and the entire source register are unaffected.

LDCTL R,FCW
dst(2:7) <- FCW(2:7)
dst(11:15) <- FCW(11:15)
dst(0:1) <- 0
dst(8:10) <- 0

The contents of the FCW register are loaded into the destination (a word register). Bits 0, 1, 8, 9 and 10 of the destination register are cleared to zero. The FCW register is unaffected.



Shaded areas are reserved

FLAGS: LDCTL FCW,R
C: Loaded from source
Z: Loaded from source
S: Loaded from source
P/V: Loaded from source
D: Loaded from source
H: Loaded from source

LDCTL R,FCW
No flags affected

CYCLES: 7

NOTE: This is a privileged instruction.

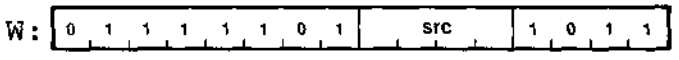
EXAMPLE: If register R7 contains %D8FC (1101100011111100), the statement

```
    LDCTL  FCW,R7
```

will leave the value %D8FC in the FCW register (Segmentation and System modes are set, interrupts are enabled, and all the flags are set).

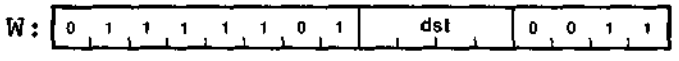
LDCTL
Load Control Register

INSTRUCTION FORMAT: (F8.1)



dst = CTLR src = R

(F8.2)



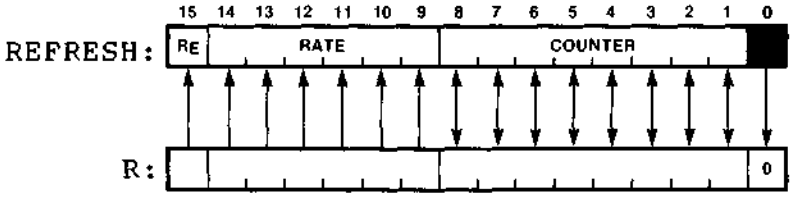
dst = R src = CTLR

OPERATION: LDCTL REFRESH,R
REFRESH(1:15) <- src(1:15)

The contents of the source (a word register) are loaded into the REFRESH register. Bit 0 of the REFRESH register and the entire source register are unaffected.

LDCTL R,REFRESH
dst(1:8) <- REFRESH(1:8)
dst(0) <- 0
dst(9:15) <- undefined

The contents of the counter field (bits 1 to 8) of the REFRESH register are loaded into the destination (a word register). Bit 0 of the destination register is cleared to zero, and the upper seven bits are undefined. The REFRESH register is unaffected.



Shaded areas are reserved

FLAGS: No flags affected

CYCLES: 7

NOTE: This is a privileged instruction.

EXAMPLE: If register R0 contains the value %8600 (1000011000000000), the statement

LDCTL REFRESH,R0

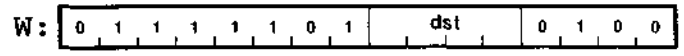
will leave the value %8600 in the REFRESH register (refresh is enabled, the rate is 3, and the counter starts at zero).

LDCTL
Load Control Register

INSTRUCTION FORMAT: (F8.1)

dst CTLR src R

(F8.2)



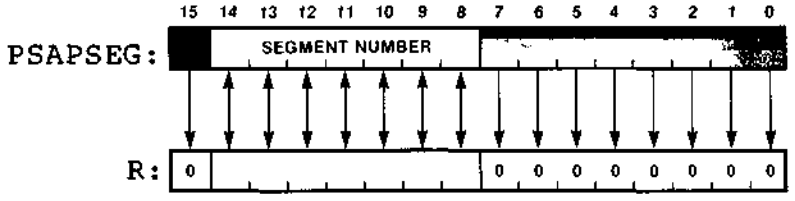
dst R src CTLR

OPERATION: LDCTL PSAPSEG,R
PSAPSEG(8:14) <- src(8:14)

The contents of the source (a word register) are loaded into the Program Status Area Pointer Segment (PSAPSEG) register. Bits 0, 1, 2,...,7, and 15 of the PSAPSEG register and the entire source register are unaffected.

LDCTL R,PSAPSEG
dst(8:14) <- PSAPSEG(8:14)
dst(0:7) <- 0
dst(15) <- 0

The contents of the PSAPSEG register are loaded into the destination (a word register). Bits 0, 1, 2,...,7, and 15 of the destination register are cleared to zero. The PSAPSEG register is unaffected.



Shaded areas are reserved

FLAGS: No flags affected

CYCLES: 7

NOTES: This is a privileged instruction.

The PSAPSEG register may not be used in the nonsegmented Z8002 version. In the segmented Z8001 version, care must be exercised when changing the two PSAP register values so that an interrupt occurring between the changing of PSAPSEG and PSAPOFF is handled correctly.

LDCTL
Load Control Register

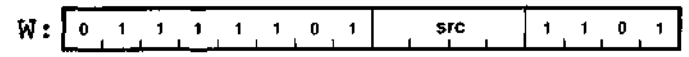
EXAMPLE: If register R12 contains %0200, the statement

```
LDCTL  PSAPSEG,R12
```

will leave the value %0200 (segment 2) in the PSAPSEG register.

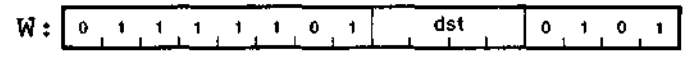
LDCTL
Load Control Register

INSTRUCTION FORMAT: (F8.1)



dst CTLR src R

(F8.2)



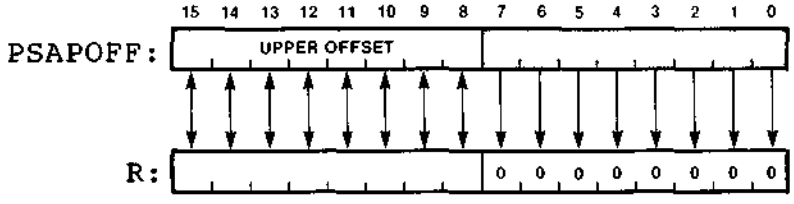
dst R src CTLR

OPERATION: LDCTL PSAPOFF,R
PSAPOFF(8:15) <- src(8:15)

The contents of the source (a word register) are loaded into the Program Status Area Pointer Offset (PSAPOFF) register. Bits 0, 1, 2,...,7 of the PSAPOFF register and the entire source register are unaffected.

LDCTL R,PSAPOFF
dst(8:15) <- PSAPOFF(8:15)
dst(0:7) <- 0

The contents of the PSAPOFF register are loaded into the destination (a word register). Bits 0, 1, 2,...,7 of the destination register are cleared to zero. The PSAPOFF register is unaffected.



Shaded areas are reserved

FLAGS: No flags affected

CYCLES: 7

NOTES: This is a privileged instruction.

In the nonsegmented Z8002 version, the mnemonic "PSAP" should be used in the assembly language statement, and indicates the same control register as the mnemonic "PSAPOFF".

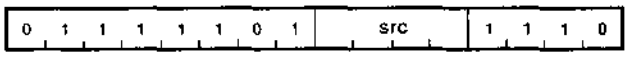
In the segmented Z8001 version, care must be exercised when changing the two PSAP register values so that an interrupt occurring between the changing of PSAPSEG and PSAPOFF is handled correctly.

EXAMPLE: If the PSAPOFF register contains %0100, the statement

```
LDCTL  R13,PSAPOFF
```

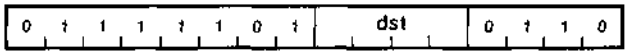
will leave the value %0100 in register R13.

INSTRUCTION FORMAT: (F8.1)



dst CTLR src R

(F8.2)



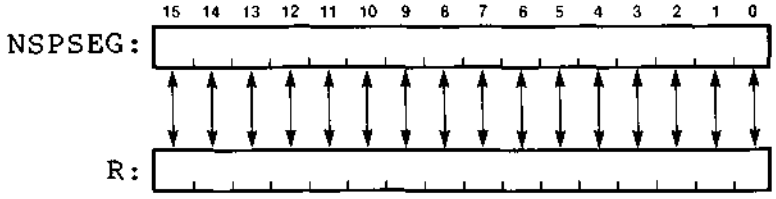
dst R src CTLR

OPERATION: LDCTL NSPSEG,R
NSPSEG <- src

The contents of the source (a word register) are loaded into the Normal Stack Pointer Segment (NSPSEG) register. In segmented mode, the NSPSEG register is R14 in Normal mode, and contains the segment number of the normal processor stack pointer. (In nonsegmented mode, R14 is not used as part of the normal processor stack pointer.)

LDCTL R,NSPSEG
dst <- NSPSEG

The contents of the NSPSEG register are loaded into the destination (a word register). The NSPSEG register is unaffected.



FLAGS: No flags affected

CYCLES: 7

NOTES: This is a privileged instruction.

The NSPSEG register may not be used in the nonsegmented Z8002 version.

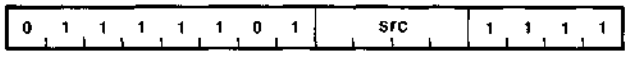
EXAMPLE: If register R12 contains %7F00, the statement

```
LDCTL  NSPSEG,R12
```

will leave the value %7F00 (segment %7F) in the NSPSEG register.

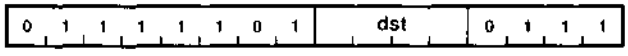
LDCTL
Load Control Register

INSTRUCTION FORMAT: (F8.1)



dst (CTLR) src (R)

(F8.2)



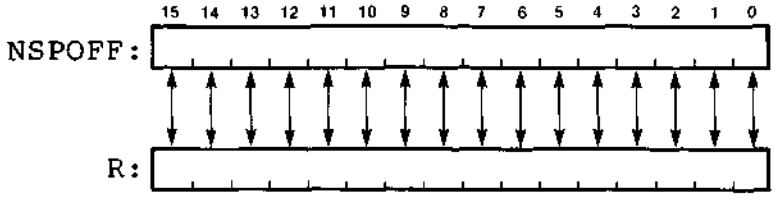
dst (R) src (CTLR)

OPERATION: LDCTL NSPOFF,R
NSPOFF <- src

The contents of the source (a word register) are loaded into the Normal Stack Pointer Offset (NSPOFF) register. In segmented mode, the NSPOFF register is R15 in Normal mode, and contains the offset part of the normal processor stack pointer. In nonsegmented mode, R15 is the entire normal processor stack pointer. The source register is unaffected.

LDCTL R,NSPOFF
dst <- NSPOFF

The contents of the NSPOFF register are loaded into the destination (a word register). The NSPOFF register is unaffected.



FLAGS: No flags affected

CYCLES: 7

NOTES: This is a privileged instruction.

In the nonsegmented Z8002 version, the mnemonic "NSP" should be used in the assembly language statement, and indicates the same control register as the mnemonic "NSPOFF".

EXAMPLE: If the NSPOFF register contains %F002, the statement

LDCTL R13,NSPOFF

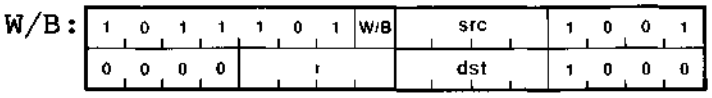
will leave the value %F002 in register R13.

LDD
Load and Decrement

```
LDD   dst,src,r          dst: IR
LDDB                     src: IR
```

INSTRUCTION FORMAT: (F6.4)

dst	src	r
IR	IR	R



OPERATION:
```
dst <- src
AUTODECREMENT dst and src {-1 if byte, -2 if word}
r <- r - 1
```

This instruction is used for block transfers of strings of data. The contents of the location addressed by the source register are loaded into the location addressed by the destination register. The source and destination registers are then decremented by one if LDDB, or by two if LDD, thus moving the pointers to the previous elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS:
- C: Unaffected
- Z: Undefined
- S: Unaffected
- V: Set if the result of decrementing r is zero; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte
20

EXAMPLE: If register R1 contains %202A, register R2 contains %404A, the word at location %404A contains %FFFF, and register R3 contains 5, the statement

```
LDD  @R1,@R2,R3
```

will leave the value %FFFF in location %202A, the value %2028 in R1, the value %4048 in R2, and the value 4 in R3.

LDDR
Load, Decrement and Repeat

LDDR dst,src,r
LDDRB

dst: IR
src: IR

INSTRUCTION FORMAT: (F6.4)

	dst	src	r
	IR	IR	R

W/B:

OPERATION:
dst <- src
AUTODECREMENT dst and src {-1 if byte, -2 if word}
r <- r - 1
repeat until r = 0

This instruction is used for block transfers of strings of data. The contents of the location addressed by the source register are loaded into the location addressed by the destination register. The source and destination registers are then decremented by one if LDDRB, or by two if LDDR, thus moving the pointers to the previous elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can transfer from 1 to 65536 bytes or 32768 words (the value for r must not be greater than 32768 for LDDR).

The effect of decrementing the pointers during the transfer is important if the source and destination strings overlap with the source string starting at a lower memory address. Placing the pointers at the highest address of the strings, and decrementing the pointers, ensures that the source string will be copied without destroying the overlapping area.

FLAGS:
C: Unaffected
Z: Undefined
S: Unaffected
V: Set
D: Unaffected
H: Unaffected

CYCLES:

Word/Byte	
11+9*n	(n=number of data elements transferred)

NOTE: This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

EXAMPLE: If register R1 contains %202A, register R2 contains %404A, the words at locations %4040 through %404A all contain %FFFF, and register R3 contains 5, the statement

```
LDDR  @R1,@R2,R3
```

will leave the value %FFFF in the words at locations %2020 through %202A, the value %201E in R1, the value %403E in R2, and 0 in R3.

LDI
Load and Increment

```
LDI   dst,src,r                dst: IR
LDIB                           src: IR
```

INSTRUCTION FORMAT: (F6.4)

W/B:

dst	src	r
IR	IR	R

OPERATION:

```
dst <- src
AUTOINCREMENT dst and src {+1 if byte, +2 if word}
r <- r - 1
```

This instruction is used for block transfers of strings of data. The contents of the location addressed by the source register are loaded into the location addressed by the destination register. The source and destination registers are then incremented by one if LDIB, or by two if LDI, thus moving the pointers to the next elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS:
- C: Unaffected
- Z: Undefined
- S: Unaffected
- V: Set if the result of decrementing r is zero; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte
20

EXAMPLE: This instruction is used in a "loop" of instructions which transfers a string of data from one location to another, but an intermediate operation on each data element is required. The following sequence transfers a string of 80 bytes, but tests for a special value (%0D, an ASCII return character) which terminates the loop if found:

```
      LD      R3,#80          !Initialize counter!
      LD      R1,#DSTBUF      !Load start addresses!
      LD      R2,#SRCBUF
LOOP:
      CPB     @R2,#%0D        !Check for return character!
      JR      EQ,DONE         !Exit loop if found!
      LDIB    @R1,@R2,R3      !Transfer next byte!
      JR      NOV,LOOP        !Repeat until counter=0!
DONE:
```

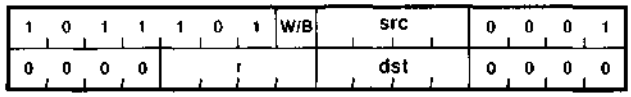
LDIR
Load, Increment and Repeat

```
LDIR  dst,src,r                dst: IR
LDIRB                          src: IR
```

INSTRUCTION FORMAT: (F6.4)

	dst	src	r
	IR	IR	R

W/B:



OPERATION:

```
dst <- src
AUTOINCREMENT dst and src {+1 if byte, +2 if word}
r <- r - 1
repeat until r = 0
```

This instruction is used for block transfers of strings of data. The contents of the location addressed by the source register are loaded into the location addressed by the destination register. The source and destination registers are then incremented by one if LDIRB, or by two if LDIR, thus moving the pointers to the next elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can transfer from 1 to 65536 bytes or 32768 words (the value for r must not be greater than 32768 for LDIR).

The effect of incrementing the pointers during the transfer is important if the source and destination strings overlap with the source string starting at a higher memory address. Placing the pointers at the lowest address of the strings, and incrementing the pointers, ensures that the source string will be copied without destroying the overlapping area.

FLAGS:
C: Unaffected
Z: Undefined
S: Unaffected
V: Set
D: Unaffected
H: Unaffected

CYCLES:

Word/Byte
11+9*n

(n=number of data elements transferred)

LDIR

Load, Increment and Repeat

NOTE: This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

EXAMPLE: The following sequence of instructions can be used to copy a buffer of 512 words (1024 bytes) from one area to another. The pointers to the start of the source and destination buffers are set, the number of words to transfer is set, and then the transfer is accomplished.

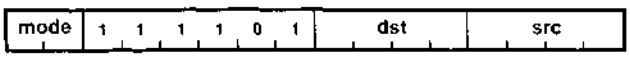
```
LD      R1,#DSTBUF
LD      R2,#SRCBUF
LD      R3,#512
LDIR    @R1,@R2,R3
```

LDK
Load Constant

LDK dst,src

dst: R
src: IM

INSTRUCTION FORMAT: (F1.2)



mode	dst	src
10	R	IM

OPERATION: dst <- src {src is 0 to 15}

The source operand (a constant value) is loaded into the destination. The source is a value from 0 to 15 which is loaded into the low-order four bits of the destination with the high-order twelve bits cleared to zeros.

FLAGS: No flags affected

CYCLES:

Word
5

NOTE: The "src field" in the instruction format encoding contains the source operand. The "src field" values range from 0 to 15 corresponding to the source values 0 to 15.

```
EXAMPLE:  LDK  R3,#9        !Register R3 is loaded with the
                             value 9!
```

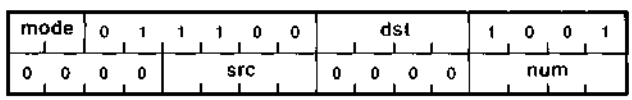
LDM
Load Multiple

LDM dst,src,num

dst: R, IR, DA, X
src: R, IR, DA, X
num: IM

INSTRUCTION FORMAT: (F6.1)

mode	dst	src
00	R	IR
01	R	DA (src field=0)
01	R	X (src field<>0)



mode	dst	src
00	IR	R
01	DA	R (dst field=0)
01	X	R (dst field<>0)

Registers from Memory
Rdst,Rdst+num-1 <- src

The contents of num source words are loaded into the destination register and the following (num-1) registers. The contents of the source(s) are not affected. The value of num can vary from 1 to 16, and the registers are loaded in increasing order, with R0 following R15.

Memory from Registers
dst <- Rsrc,Rsrc+num-1

The contents of num word registers including the source register and the following (num-1) registers are loaded into the destination words. The contents of the source(s) are not affected. The value of num can vary from 1 to 16, and the registers are loaded in increasing order, with R0 following R15.

FLAGS: No flags affected

CYCLES:

dst	src	Word NS	Word SS	Word SL	
R	IR	11+3*n	--	--	(n=number of registers)
R	DA	14+3*n	15+3*n	17+3*n	
R	X	15+3*n	15+3*n	18+3*n	
IR	R	11+3*n	--	--	
DA	R	14+3*n	15+3*n	17+3*n	
X	R	15+3*n	15+3*n	18+3*n	

NOTES: The "num field" in the instruction format encoding contains the num operand. The "num field" values range from 0 to 15 corresponding to the num values 1 to 16.

The starting address of the block of memory where the registers are loaded from or to is computed once at the start of execution, and incremented by two for each register loaded. If the original address computation involved a register, the register's value will not be affected by the address incrementation during execution.

EXAMPLE: If register R5 contains 5, R6 contains %0100, and R7 contains 7, the statement

```
LDM  @R6,R5,#3
```

will leave the values 5, %0100, and 7 at word locations %0100, %0102, and %0104, respectively, and none of the registers will be affected.

LDPS
Load Program Status

LDPS src

src: IR, DA, X

INSTRUCTION FORMAT: (F2.3)

mode	src
00	IR
01	DA (src field=0)
01	X (src field<>0)

OPERATION: PS <- src

The contents of the source operand are loaded into the Program Status (PS), loading the Flags and Control Word (FCW) and the program counter (PC). The new value of the FCW does not become effective until the next instruction, so that the status pins will not be affected by the new control bits until after the LDPS instruction execution is completed. The next instruction executed is that addressed by the new contents of the PC. The contents of the source are not affected.

This instruction is used to set the Program Status of a program, and is particularly useful for setting the System/Normal mode of a program to Normal mode, or for running a nonsegmented program in the segmented Z8001 version. The PC segment number is not affected by the LDPS instruction in nonsegmented mode.

The format of the source operand (Program Status block) depends on the current Segmentation mode (not on the version of the Z8000), and is illustrated in the following figure:

(shaded area is reserved--must be zero)

LDPS
Load Program Status

FLAGS:
C: Loaded from source
Z: Loaded from source
S: Loaded from source
P/V: Loaded from source
D: Loaded from source
H: Loaded from source

CYCLES:

src	Address NS	SS	SL
IR	12	--	16
DA	16	20	22
X	17	20	23

NOTE: This is a privileged instruction.

EXAMPLE: In the nonsegmented Z8002 version, if the program counter contains %2550, register R3 contains %5000, location %5000 contains %1800, and location %5002 contains %A000, the statement

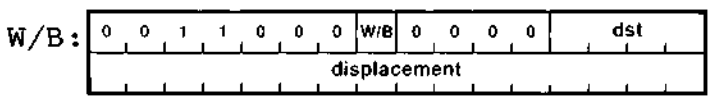
```
LDPS  @R3
```

will leave the value %A000 in the program counter, and the FCW value will be %1800 (indicating Normal mode, interrupts enabled, and all flags cleared).

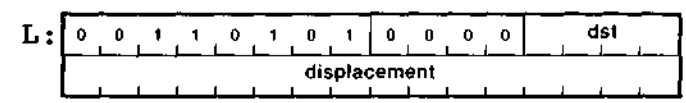
LDR
Load Relative

```
LDR   dst,src                    dst: R, RA
LDRB                             src: R, RA
LDRL
```

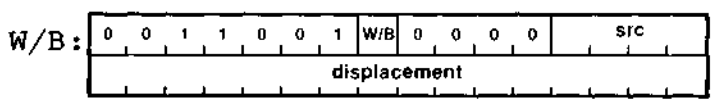
INSTRUCTION FORMAT: (F4.4)



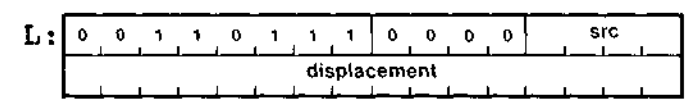
dst	src
R	RA



(F4.2)



dst	src
RA	R



OPERATION: dst <- src

The contents of the source operand are loaded into the destination. The contents of the source are not affected.

FLAGS: No flags affected

CYCLES:

Word/Byte	Long
14	17

NOTES: The relative addressing mode is calculated by adding the displacement in the instruction to the updated value of the program counter (PC) to derive the operand's address. The updated PC value is taken to be the address of the instruction byte following the LDR, LDRB, or LDRL instruction, while the displacement is a 16-bit signed value in the range -32768 to +32767. The assembler automatically calculates the displacement by subtracting the PC value of the following instruction from the address given by the programmer.

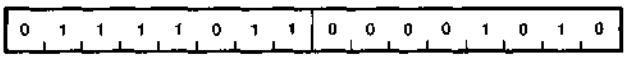
Status pin information during the access to memory for the data operand will be IFn instead of $\overline{\text{MREQ}}$ because the address is relative to PC.

```
EXAMPLE:  LDR  R2,DATA        !Register R2 is loaded with the
                               value in the location named DATA!
```

MBIT
Multi-Micro Bit Test

MBIT

INSTRUCTION FORMAT: (F9.3)



OPERATION: S <- NOT $\overline{\mu I}$

This instruction is used to synchronize multiple processors' exclusive access to shared hardware resources. The multi-micro input pin ($\overline{\mu I}$) is tested, and the S flag is cleared if the pin is one (active state = low voltage); otherwise, the S flag is set, indicating that the pin is zero (inactive state = high voltage).

After the MBIT instruction is executed, the S flag can be used to determine whether a requested resource is available or not. If the S flag is clear, then the resource is not available; if the S flag is set, then the resource is available for use by this CPU.

FLAGS:
C: Unaffected
Z: Undefined
S: Set if $\overline{\mu I}$ is zero; cleared otherwise
V: Unaffected
D: Unaffected
H: Unaffected

CYCLES: 7

NOTE: This is a privileged instruction.

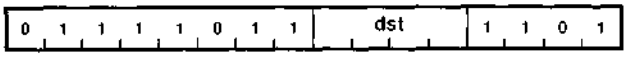
EXAMPLE: The following sequence of instructions can be used to wait for the availability of a resource.

```
LOOP:
     MBIT              !Test multi-micro input!
     JR  PL,LOOP       !Repeat until resource is available!
AVAILABLE:
```

MREQ
Multi-Micro Request

MREQ dst dst: R

INSTRUCTION FORMAT: (F8.2)




```
OPERATION:   Z <- 0
             if μI = 1 then  S <- 0
                             μO <- 0

                       else  μO <- 1
                             repeat dst <- dst - 1
                             μntil dst = 0
                             if μI = 1  then  S <- 1
                                        else  S <- 0
                                              μO <- 0
                             Z <- 1
```

This instruction is used to synchronize multiple processors' exclusive access to shared hardware resources. A request for a resource is signalled through the multi-micro input and output pins ($\overline{\mu I}$ and $\overline{\mu O}$), with the S and Z flags indicating the availability of the resource after the MREQ instruction has been executed.

First, the Z flag is cleared. Then the $\overline{\mu I}$ pin is tested. If the $\overline{\mu I}$ pin is one (active state = low voltage), the S flag is cleared and the $\overline{\mu O}$ pin is cleared to zero (inactive state = high voltage), thus indicating that the resource is not available. Instruction execution is finished in this case.

If the $\overline{\mu I}$ pin is zero, indicating that the resource may be available, a sequence of machine operations occurs. First, the $\overline{\mu O}$ pin is set to one (active state = low voltage), signalling a request by the CPU for the resource. Next, a finite delay to allow for propagation of the signal to other processors is accomplished by repeatedly decrementing the contents of the destination (a word register) until its value is zero. Then the $\overline{\mu I}$ pin is tested to determine whether the request for the resource was acknowledged. If the $\overline{\mu I}$ pin is one (active state = low voltage), the S flag is set to one indicating that the resource is available and access is granted. If the $\overline{\mu I}$ pin is zero (inactive state = high voltage), the S flag is

cleared to zero, and the $\overline{\mu O}$ pin is cleared to zero (inactive state = high voltage), indicating that the request was not granted. Finally, in either case, the Z flag is set to one, indicating that the original test of the $\overline{\mu I}$ pin caused a request to be made.

S flag	Z flag	$\overline{\mu O}$	Indicates
0	0	0	Request not signalled (resource not available)
0	1	0	Request not granted (resource not available)
1	1	1	Request granted (resource available)

FLAGS: C: Unaffected
Z: Set if request was signalled; cleared otherwise
S: Set if request was signalled and granted; cleared otherwise
V: Unaffected
D: Unaffected
H: Unaffected

CYCLES: 12 (request not signalled)
12+7*n (request signalled, n=number of times destination is decremented)

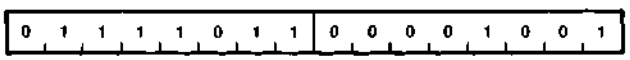
NOTE: This is a privileged instruction.

EXAMPLE:
```
TRY:
          LD     R0,#50     !Allow for propagation delay!
          MREQ   R0         !Multi-micro request with
                             delay in register R0!
          JR   MI,AVAILABLE
          JR    Z,NOT_GRANTED

NOT_AVAILABLE:  .           !Resource not available!
                .
                .
NOT_GRANTED:    .           !Request not granted!
                .
                .
          JR   TRY          !Try again after awhile!
AVAILABLE:      .           !Use resource!
                .
                .
          MRES              !Release resource!
```

MRES
Multi-Micro Reset

MRES

INSTRUCTION FORMAT: (F9.3)



OPERATION: $\overline{\mu O}$ <- 0

This instruction is used to synchronize multiple processors' exclusive access to shared hardware resources. The multi-micro output pin ($\overline{\mu O}$) is cleared to zero (inactive state = high voltage).

Resetting $\overline{\mu O}$ to zero is used to indicate that a resource controlled by the CPU is available for use by other processors.

FLAGS: No flags affected

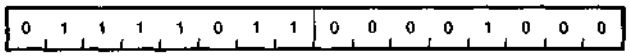
CYCLES: 5

NOTE: This is a privileged instruction.

```
EXAMPLE:  MRES      !Signal that resource controlled
                     by this CPU is available to
                     other processors!
```

MSET

INSTRUCTION FORMAT: (F9.3)



OPERATION: $\overline{\mu O}$ <- 1

This instruction is used to synchronize multiple processors' exclusive access to shared hardware resources. The multi-micro output pin ($\overline{\mu O}$) is set to one (active state = low voltage).

Setting $\overline{uO}$ to one is used either to indicate that a resource controlled by the CPU is not available to other processors, or to signal a request for a resource controlled by some other processor.

FLAGS: No flags affected

CYCLES: 5

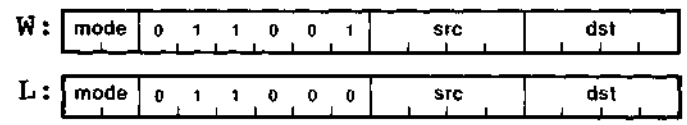
NOTE: This is a privileged instruction.

EXAMPLE:
```
MSET      !Signal that controlled resource
           is not available to other processors!
```

MULT
Multiply

```
MULT  dst,src                    dst: R
MULTL                            src: R, IM, IR, DA, X
```

INSTRUCTION FORMAT: (F2.1)



mode	dst	src
10	R	R
00	R	IM (src field=0)
00	R	IR (src field<>0)
01	R	DA (src field=0)
01	R	X (src field<>0)

OPERATION:

Word
dst,dst$_{V}$1 <- dst$_{V}$1 * src

Long
dst,dst$_{V}$1,dst$_{V}$2,dst$_{V}$3 <- dst$_{V}$2,dst$_{V}$3 * src,src$_{V}$1

The low-order half of the destination operand (multiplicand) is multiplied by the source operand (multiplier) and the product is stored in the destination. The contents of the source are not affected. Both operands are treated as signed, two's complement integers. For MULT, the destination is a register pair and the source is a word value; for MULTL, the destination is a register quadruple and the source is a long word value.

FLAGS:
- C: MULT - set if product is less than -2^{15} or greater than or equal to 2^{15}; cleared otherwise; MULTL - set if product is less than 2^{31} or greater than or equal to 2^{31} ; cleared otherwise
- Z: Set if the result is zero; cleared otherwise
- S: Set if the result is negative; cleared otherwise
- V: Cleared
- D: Unaffected
- H: Unaffected

CYCLES:

src	Word			Long		
	NS	SS	SL	NS	SS	SL
R	70	--	--	282+7*n	--	--
IM	70	--	--	282+7*n	--	--
IR	70	--	--	282+7*n	--	--
DA	71	72	74	283+7*n	284+7*n	286+7*n
X	72	72	75	284+7*n	284+7*n	287+7*n

(n=number of bits equal to one in the absolute value of the low-order 16 bits of the destination operand)

(Multiplier is zero)

R	18	--	--	30	--	--
IM	18	--	--	30	--	--
IR	18	--	--	30	--	--
DA	19	20	22	31	32	34
X	20	20	22	32	32	35

NOTE: For proper instruction execution, the "dst field" in the instruction format encoding must be even for MULT, and must be a multiple of 4 (0, 4, 8, 12) for MULTL. If the source operand in MULTL is a register, the "src field" must be even.

EXAMPLE: If register RQ0 (composed of register pairs RR0 and RR2) contains %2222222200000031 (RR2 contains decimal 49), the statement

```
MULTL  RQ0,#10
```

will leave the value %00000000000001EA (decimal 490) in RQ0.

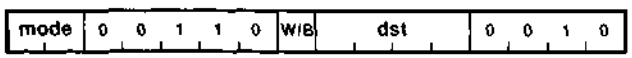
NEG
Negate

NEG dst
NEGB

dst: R, IR, DA, X

INSTRUCTION FORMAT: (F1.1)

W/B:



mode	dst
10	R
00	IR
01	DA (dst field=0)
01	X (dst field<>0)

OPERATION: dst <- - dst

The contents of the destination is negated, that is, replaced by its two's complement value. Note that %8000 for NEG and %80 for NEGB are replaced by themselves since in two's complement representation the most negative number has no positive counterpart.

FLAGS:
- C: Cleared if the result is zero; set otherwise, which indicates a "borrow"
- Z: Set if the result is zero; cleared otherwise
- S: Set if the result is negative; cleared otherwise
- V: Set if the result is %8000 for NEG, or %80 for NEGB; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

dst	Word/Byte		
	NS	SS	SL
R	7	--	--
IR	12	--	--
DA	15	16	18
X	16	16	19

EXAMPLE: If register R8 contains %051F, the statement

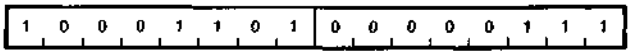
```
NEG  R8
```

will leave the value %FAE1 in R8.

NOP
No Operation

NOP

INSTRUCTION FORMAT: (F9.3)



OPERATION:

No operation is performed. This instruction may be used for timing delays.

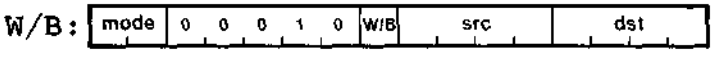
FLAGS: No flags affected

CYCLES: 7

OR
Or

```
        OR    dst,src                dst: R
        ORB                          src: R, IM, IR, DA, X
```

INSTRUCTION FORMAT: (F2.1)



mode	dst	src
10	R	R
00	R	IM (src field=0)
00	R	IR (src field<>0)
01	R	DA (src field=0)
01	R	X (src field<>0)

OPERATION: dst <- dst OR src

The source operand is logically ORed with the destination operand and the result is stored in the destination. The contents of the source are not affected. The OR operation results in a one bit being stored whenever either of the corresponding bits in the two operands is one; otherwise, a zero bit is stored.

FLAGS:
- C: Unaffected
- Z: Set if the result is zero; cleared otherwise
- S: Set if the most significant bit of the result is set; cleared otherwise
- P: OR - unaffected; ORB - set if parity of the result is even; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

src	Word/Byte		
	NS	SS	SL
R	4	--	--
IM	7	--	--
IR	7	--	--
DA	9	10	12
X	10	10	13

EXAMPLE: If register RL3 contains %C3 (11000011) and the source operand is the immediate value %7B (01111011), the statement

```
    ORB  RL3,#%7B
```

will leave the value %FB (11111011) in RL3.

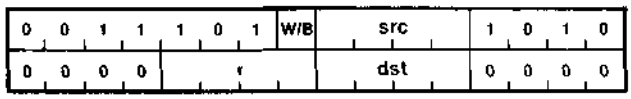
OTDR
Output, Decrement and Repeat

```
OTDR  dst,src,r                    dst: IR
OTDRB                              src: IR
```

INSTRUCTION FORMAT: (F6.4)

	dst	src	r
	IR	IR	R

W/B:



OPERATION:
```
dst <- src
AUTODECREMENT src {-1 if byte, -2 if word}
r <- r - 1
repeat until r = 0
```

This instruction is used for block output of strings of data. The contents of the memory location addressed by the source register are loaded into the I/O location addressed by the destination register. I/O addresses are always 16 bits. The source register is then decremented by one if OTDRB, or by two if OTDR, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can output from 1 to 65536 bytes or 32768 words (the value for r must not be greater than 32768 for OTDR).

FLAGS:
```
C: Unaffected
Z: Undefined
S: Unaffected
V: Set
D: Unaffected
H: Unaffected
```

CYCLES:

Word/Byte	
11+10*n	(n=number of data elements transferred)

NOTES: This is a privileged instruction.

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

OTDR
Output, Decrement and Repeat

EXAMPLE: If register R11 contains %0FFF, register R12 contains %B006, and R13 contains 6, the statement

```
OTDR @R11,@R12,R13
```

will output the string of words from locations %AFFC to %B006 in descending order to "port" %0FFF. R12 will contain %AFFA, and R13 will contain 0. R11 will not be affected.

OTIR
Output, Increment and Repeat

```
OTIR  dst,src,r                dst: IR
OTIRB                          src: IR
```

INSTRUCTION FORMAT: (F6.4)

W/B:

dst	src	r
IR	IR	R

OPERATION:
```
dst <- src
AUTOINCREMENT src {+1 if byte, +2 if word}
r <- r - 1
repeat until r = 0
```

This instruction is used for block output of strings of data. The contents of the memory location addressed by the source register are loaded into the I/O location addressed by the destination register. I/O addresses are always 16 bits. The source register is the incremented by one if OTIRB, or by two if OTIR, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can output from 1 to 65536 bytes or 32768 words (the value for r must not be greater than 32768 for OTIR).

FLAGS:
```
C: Unaffected
Z: Undefined
S: Unaffected
V: Set
D: Unaffected
H: Unaffected
```

CYCLES:

Word/Byte	
11+10*n	(n=number of data elements transferred)

NOTES: This is a privileged instruction.

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

OTIR

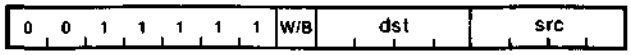
Output, Increment and Repeat

EXAMPLE: The following sequence of instructions can be used to output a string of bytes to the specified I/O "port". The pointers to the I/O port and the start of the source string are set, the number of bytes to output is set, and then the output is accomplished.

```
LD      R1,#PORT
LD      R2,#SRCBUF
LD      R3,#LENGTH
OTIRB   @R1,@R2,R3
```

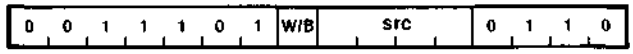
```
OUT   dst,src                    dst: IR,DA
OUTB                             src: R
```

INSTRUCTION FORMAT: (F7.4)



dst	src
IR	R

(F7.2)



dst	src
DA	R

OPERATION: dst <- src

The contents of the source operand are loaded into the destination. I/O addresses are always 16 bits.

FLAGS: No flags affected

CYCLES:

dst	Word/Byte
IR	10
DA	12

NOTE: This is a privileged instruction.

EXAMPLE: If register R6 contains %5252, the statement

```
OUT  %1120,R6
```

will output the value %5252 to the "port" %1120.

OUTD
Output and Decrement

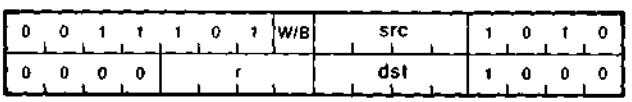
OUTD dst,src,r
OUTDB

dst: IR
src: IR

INSTRUCTION FORMAT: (F6.4)

	dst	src	r
	IR	IR	R

W/B:



OPERATION: dst <- src
AUTODECREMENT src {-1 if byte, -2 if word}
r <- r - 1

This instruction is used for block output of strings of data. The contents of the memory location addressed by the source register are loaded into the I/O location addressed by the destination register. I/O addresses are always 16 bits. The source register is then decremented by one if OUTDB, or by two if OUTD, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS:
C: Unaffected
Z: Undefined
S: Unaffected
V: Set if the result of decrementing r is zero; cleared otherwise
D: Unaffected
H: Unaffected

CYCLES:

Word/Byte
21

NOTE: This is a privileged instruction.

EXAMPLE: In segmented mode, if register R2 contains the I/O address %0030, register RR6 contains %12005552 (segment %12, offset %5552), the word at memory location %12005552 contains %1234, and register R8 contains %1001, the statement

```
OUTD @R2,@RR6,R8
```

will output the value %1234 to "port" %0030, and leave the value %12005550 in RR6, and %1000 in R8. Register R2 will not be affected.

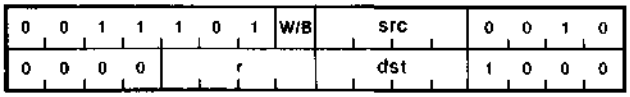
OUTI
Output and Increment

```
OUTI  dst,src,r                dst: IR
OUTIB                          src: IR
```

INSTRUCTION FORMAT: (F6.4)

	dst	src	r
	IR	IR	R

W/B:



OPERATION:
```
dst <- src
AUTOINCREMENT src {+1 if byte, +2 if word}
r <- r - 1
```

This instruction is used for block output of strings of data. The contents of the memory location addressed by the source register are loaded into the I/O location addressed by the destination register. I/O addresses are always 16 bits. The source register is then incremented by one if OUTIB, or by two if OUTI, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS:
- C: Unaffected
- Z: Undefined
- S: Unaffected
- V: Set if the result of decrementing r is zero; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte
21

NOTE: This is a privileged instruction.

EXAMPLE: This instruction is used in a "loop" of instructions which outputs a string of data, but an intermediate operation on each element is required. The following sequence outputs a string of 80 ASCII characters (bytes) with the most significant bit of each byte set or reset to provide even parity for the entire byte.

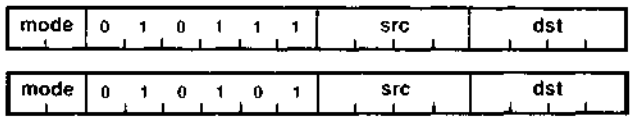
OUTI

Output and Increment

```
        LD      R1,#PORT        !Load I/O address!
        LD      R2,#SRCSTART    !Load start of string!
        LD      R3,#80          !Initialize counter!
LOOP:
        TESTB   @R2             !Test byte parity!
        JR      PE,EVEN
        SETB    @R2,#7          !Force even parity!
EVEN:
        OUTIB   @R1,@R2,R3      !Output next byte!
        JR      NOV,LOOP        !Repeat until counter=0!
DONE:
```

```
POP   dst,src                    dst: R, IR, DA, X
POPL                             src: IR
```

INSTRUCTION FORMAT: (F2.1)



mode	dst	src
10	R	IR
00	IR	IR
01	DA	IR (dst field=0)
01	X	IR (dst field<>0)

OPERATION: dst <- src
AUTOINCREMENT src {+2 if word, +4 if long}

The contents of the location addressed by the source register (a "stack pointer") are loaded into the destination. The source register is then incremented by a value which equals the size in bytes of the destination operand, thus removing the top element of the "stack" by changing the stack pointer. Any register except R0 (or RR0) can be used as a stack pointer.

FLAGS: No flags affected

CYCLES:

dst	Word			Long		
	NS	SS	SL	NS	SS	SL
R	8	--	--	12	--	--
IR	12	--	--	19	--	--
DA	16	16	19	23	23	26
X	16	16	19	23	23	26

NOTE: For POPL, the same register must not be used in both the source and destination addressing mode designators.

EXAMPLE: If register R12 (a "stack pointer") contains %1000, the word at location %1000 contains %0055, and register R3 contains %0022, the statement

```
POP  R3,@R12
```

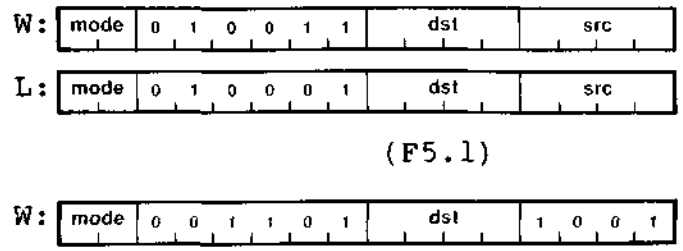
will leave the value %0055 in R3 and the value %1002 in R12.

PUSH
Push

PUSH dst,src
PUSHL

dst: IR
src: R, IM, IR, DA, X

INSTRUCTION FORMAT: (F2.2)



(F5.1)

mode	dst	src
10	IR	R
00	IR	IR
01	IR	DA (src field=0)
01	IR	X (src field<>0)

mode	dst	src
00	IR	IM

OPERATION: AUTODECREMENT dst {-2 if word, -4 if long}
dst <- src

The contents of the destination register (a "stack pointer") is decremented by a value which equals the size in bytes of the destination operand. Then the source operand is loaded into the location addressed by the updated destination register, thus adding a new element to the top of the "stack" by changing the stack pointer. Any register except R0 (or RR0) can be used as a stack pointer.

FLAGS: No flags affected

CYCLES:

src	Word			Long		
	NS	SS	SL	NS	SS	SL
R	9	--	--	12	--	--
IM	12	--	--	--	--	--
IR	13	--	--	20	--	--
DA	14	14	17	21	21	24
X	14	14	17	21	21	24

NOTE: For PUSHL, the same register must not be used in both the source and destination addressing mode designators.

EXAMPLE: If register R12 (a "stack pointer") contains %1002, the word at location %1000 contains %0055, and register R3 contains %0022, the statement

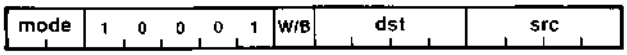
```
PUSH  @R12,R3
```

will leave the value %0022 in location %1000 and the value %1000 in R12.

```
RES   dst,src                dst: R, IR, DA, X
RESB                         src: R, IM
```

INSTRUCTION FORMAT: (F1.2)

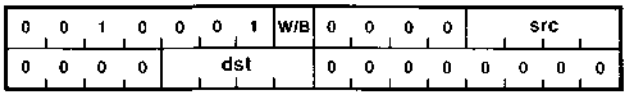
W/B:



mode	dst	src	
10	R	IM	
00	IR	IM	(dst field<>0)
01	DA	IM	(dst field=0)
01	X	IM	(dst field<>0)

(F6.3)

W/B:



dst	src
R	R

OPERATION: dst(src) <- 0

Clears the specified bit within the destination operand without affecting any other bits in the destination. The source (the bit number) can be specified as either an immediate value, or as a word register which contains the value. In the second case, the destination operand must be a register, and the source operand must be R0 through R7 for RESB, or R0 through R15 for RES. The bit number is a value from 0 to 7 for RESB, or 0 to 15 for RES, with 0 indicating the least significant bit.

FLAGS: No flags affected

CYCLES:

dst	src	Word/Byte NS	SS	SL
R	IM	4	--	--
IR	IM	11	--	--
DA	IM	13	14	16
X	IM	14	14	17
R	R	10	--	--

NOTE: Only the lower four bits of the source operand are used to specify the bit number for RES, while only the lower three bits of the source operand are used with RESB. When the source operand is an immediate value, the "src field" in the instruction format encoding contains the bit number in the lowest four bits for RES, or the lowest three bits for RESB.

EXAMPLE: If register RL3 contains %B2 (10110010), the statement

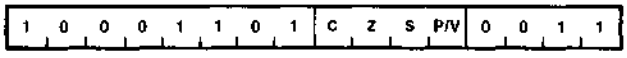
```
RESB  RL3,#1
```

will leave the value %B0 (10110000) in RL3.

RESFLG
Reset Flag

RESFLG flag flag: C, Z, S, P, V

INSTRUCTION FORMAT: (F9.1)



OPERATION: FLAGS(4:7) <- FLAGS(4:7) AND NOT instruction(4:7)

Any combination of the C, Z, S, P or V flags are cleared to zero if the corresponding bit in the instruction is one. If the corresponding bit in the instruction is zero, the flag will not be affected. All other bits in the FLAGS register are unaffected. Note that the P and V flags are represented by the same bit. There may be one, two, three or four operands in the assembly language statement, in any order.

FLAGS:
C: Cleared if specified; unaffected otherwise
Z: Cleared if specified; unaffected otherwise
S: Cleared if specified; unaffected otherwise
P/V: Cleared if specified; unaffected otherwise
D: Unaffected
H: Unaffected

CYCLES: 7

EXAMPLE: If the C, S, and V flags are all set (=1), and the Z flag is clear (=0), the statement

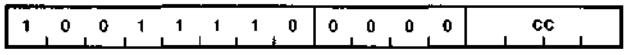
```
RESFLG  C, V
```

will leave the S flag set (=1), and the C, Z, and V flags cleared (=0).

RET
Return from Procedure

RET cc

INSTRUCTION FORMAT: (F9.4)



OPERATION:

Nonsegmented	Segmented
if cc is true then	if cc is true then
PC <- @SP	PC <- @SP
SP <- SP + 2	SP <- SP + 4

This instruction is used to return to a previously executing procedure at the end of a procedure entered by a CALL or CALR instruction. If the condition specified by "cc" is true, then the contents of the location addressed by the processor stack pointer are popped into the program counter (PC). The next instruction executed is that addressed by the new contents of the PC. See section 3.2.1 for a list of condition codes. The stack pointer used is R15 if nonsegmented, or RR14 if segmented. If the condition code is false, then control falls through to the instruction following the RET instruction.

FLAGS: No flags affected

CYCLES:

Address			
NS	SS	SL	
10	--	13	(return is taken)
7	--	7	(return not taken)

EXAMPLE: In nonsegmented mode, if the program counter contains %2550, the stack pointer (R15) contains %3000, location %3000 contains %1004, and the Z flag is clear, then the statement

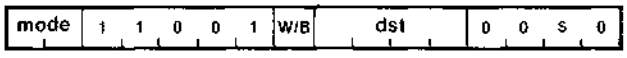
RET NZ

will leave the value %3002 in the stack pointer and the program counter will contain %1004 (the address of the next instruction to be executed).

RL
Rotate Left

```
RL    dst,src                dst: R
RLB                          src: IM
```

INSTRUCTION FORMAT: (F1.1)



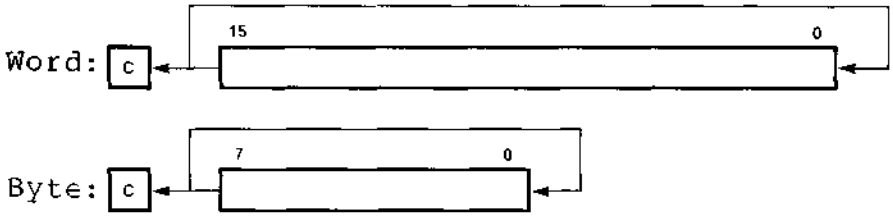
mode	dst	src
10	R	IM

(Perform operation once if s=0, twice if s=1)

OPERATION:

```
C <- dst(msb)
dst(0) <- dst(msb)
dst(n+1) <- dst(n)   {n is 0 to msb-1}
```

The contents of the destination operand are rotated left one bit position if the source operand is 1, or two bit positions if the source operand is 2. The source operand may be omitted from the assembly language statement and thus defaults to the value 1. The most significant bit of the destination operand is moved to the bit 0 position and also replaces the C flag.



FLAGS:
- C: Set if the last bit rotated from the most significant bit position was 1; cleared otherwise
- Z: Set if the result is zero; cleared otherwise
- S: Set if the most significant bit of the result is set; cleared otherwise
- V: Set if arithmetic overflow occurs, that is, if the sign of the destination changed during rotation; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte	
6	(once)
7	(twice)

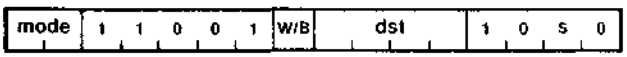
EXAMPLE: If register RH5 contains %88 (10001000), the statement

```
RLB  RH5
```

will leave the value %11 (00010001) in RH5 and the carry flag will be set to one.

```
RLC   dst,src                dst: R
RLCB                         src: IM
```

INSTRUCTION FORMAT: (F1.1)

W/B: 

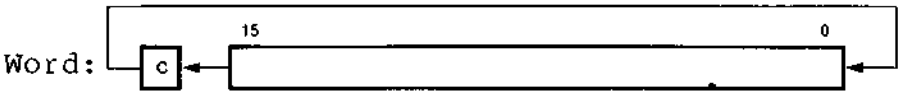
mode	dst	src
10	R	IM

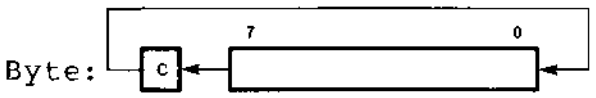
(Perform operation once if s=0, twice if s=1)

OPERATION:

```
dst(0) <- C
C <- dst(msb)
dst(n+1) <- dst(n) {n is 0 to msb-1}
```

The contents of the destination operand with the C flag are rotated left one bit position if the source operand is 1, or two bit positions if the source operand is 2. The source operand may be omitted from the assembly language statement and thus defaults to the value 1. The most significant bit of the destination operand replaces the C flag and the previous value of the C flag is moved to the bit 0 position of the destination.

Word: 

Byte: 

FLAGS:
- C: Set if the last bit rotated from the most significant bit position was 1; cleared otherwise
- Z: Set if the result is zero; cleared otherwise
- S: Set if the most significant bit of the result is set; cleared otherwise
- V: Set if arithmetic overflow occurs, that is, if the sign of the destination changed during rotation; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte	
6	(once)
7	(twice)

EXAMPLE: If the carry flag is clear (=0) and register R0 contains %800F (1000000000001111), the statement

```
RLC  R0,#2
```

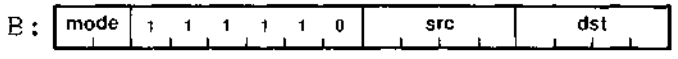
will leave the value %003D (0000000000111101) in R0 and clear the carry flag.

RLDB
Rotate Left Digit

RLDB dst,src

dst: R
src: R

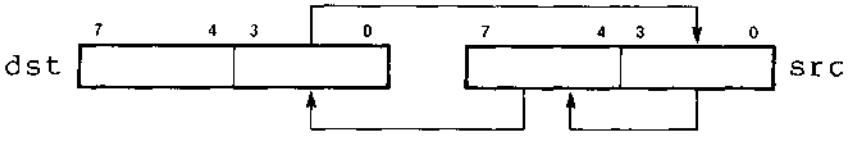
INSTRUCTION FORMAT: (F2.1)



mode	dst	src
10	R	R

OPERATION:
```
src(0:3) <- dst(0:3)
src(4:7) <- src(0:3)
dst(0:3) <- src(4:7)
```

Rotates to the left BCD digits between source and destination byte operands. Simultaneously, the lower digit of destination is moved to the lower digit of source, the lower digit of source is moved to the upper digit of source, and the upper digit of source is moved to the lower digit of destination. The result is the destination after the operation, whose upper digit is unaffected. In multiple-digit BCD arithmetic, this instruction can be used to shift to the left a string of BCD digits, thus multiplying it by a power of ten.



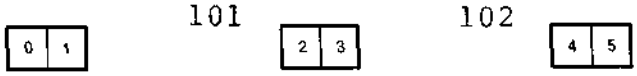
FLAGS:
- C: Unaffected
- Z: Set if the result is zero; cleared otherwise
- S: Set if the most significant bit of the result is set; cleared otherwise
- V: Unaffected
- D: Unaffected
- H: Unaffected

CYCLES:

Byte
9

NOTE: The same register must not be used in both the source and destination addressing mode designators.

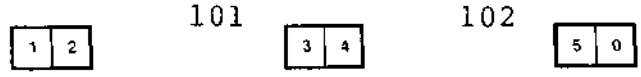
EXAMPLE: If location 100 contains the BCD digits 0,1 (00000001), location 101 contains 2,3 (00100011), and location 102 contains 4,5 (01000101)



the sequence of statements

```
     LD     R3,#3       !Set loop counter for 3 bytes (6
                         digits)!
     LD     R2,#102     !Set pointer to low-order digits!
     CLRB   RH1         !Zero-fill low-order digit!
LOOP:
     LDB    RL1,@R2     !Get next two digits!
     RLDB   RH1,RL1     !Shift digits left one position!
     LDB    @R2,RL1     !Replace shifted digits!
     DEC    R2          !Advance pointer!
     DJNZ   R3,LOOP     !Repeat until counter is zero!
```

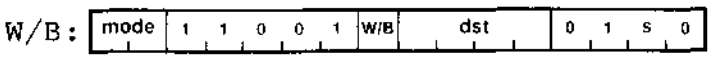
will leave the digits 1,2 (00010010) in location 100, the digits 3,4 (00110100) in location 101, and the digits 5,0 (01010000) in location 102.



RR
Rotate Right

```
RR    dst,src                    dst: R
RRB                              src: IM
```

INSTRUCTION FORMAT: (F1.1)



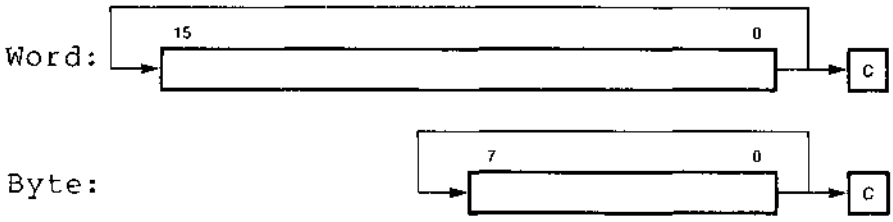
mode	dst	src
10	R	IM

(Perform operation once if s=0, twice if s=1)

OPERATION:

```
C <- dst(0)
dst(msb) <- dst(0)
dst(n) <- dst(n+1)    {n is 0 to msb-1}
```

The contents of the destination operand are rotated right one bit position if the source operand is 1, or two bit positions if the source operand is 2. The least significant bit of the destination operand is moved to the most significant bit and also replaces the C flag. The source operand may be omitted from the assembly language statement and thus defaults to the value 1.



FLAGS:
- C: Set if the last bit rotated from the least significant bit position was 1; cleared otherwise
- Z: Set if the result is zero; cleared otherwise
- S: Set if the most significant bit of the result is set; cleared otherwise
- V: Set if arithmetic overflow occurs, that is, if the sign of the destination changed during rotation; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte	
6	(once)
7	(twice)

EXAMPLE: If register RL6 contains %31 (00110001), the statement

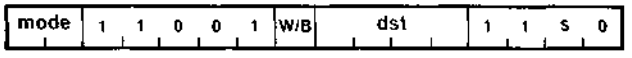
```
RRB  RL6
```

will leave the value %98 (10011000) in RL6 and the carry flag will be set to one.

RRC
Rotate Right through Carry

```
RRC   dst,src                    dst: R
RRCB                             src: IM
```

INSTRUCTION FORMAT: (F1.1)

W/B: 

mode	dst	src
10	R	IM

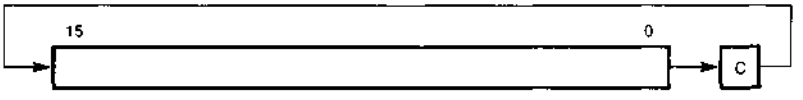
(Perform operation once if s=0, twice if s=1)

OPERATION:

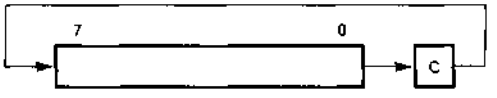
```
dst(msb) <- C
C <- dst(0)
dst(n) <- dst(n+1)   {n is 0 to msb-1}
```

The contents of the destination operand with the C flag are rotated right one bit position if the source operand is 1, or two bit positions if the source operand is 2. The least significant bit of the destination operand replaces the C flag and the previous value of the C flag is moved to the most significant bit position of the destination. The source operand may be omitted from the assembly language statement and thus defaults to the value 1.

Word:



Byte:



FLAGS:
- C: Set if the last bit rotated from the least significant bit position was 1; cleared otherwise
- Z: Set if the result is zero; cleared otherwise
- S: Set if the most significant bit of the result is set; cleared otherwise
- V: Set if arithmetic overflow occurs, that is, if the sign of the destination changed during rotation; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte	
6	(once)
7	(twice)

RRC
Rotate Right through Carry

EXAMPLE: If the carry flag is clear (=0) and the register R0 contains %00DD (0000000011011101), the statement

```
RRC  R0,#2
```

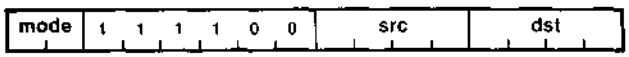
will leave the value %8037 (1000000000110111) in R0 and clear the carry flag.

RRDB
Rotate Right Digit

RRDB dst,src

dst: R
src: R

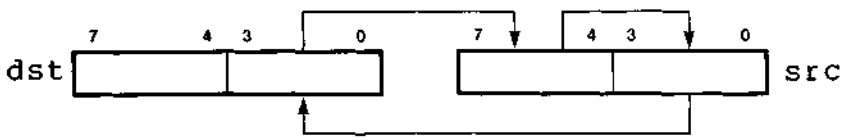
INSTRUCTION FORMAT: (F2.1)



mode	dst	src
10	R	R

OPERATION:
```
src(4:7) <- dst(0:3)
src(0:3) <- src(4:7)
dst(0:3) <- src(0:3)
```

Rotates to the right BCD digits between source and destination byte operands. Simultaneously, the lower digit of destination is moved to the upper digit of source, the upper digit of source is moved to the lower digit of source, and the lower digit of source is moved to the lower digit of destination. The result is the destination after the operation, whose upper digit is unaffected. In multiple-digit BCD arithmetic, this instruction can be used to shift to the right a string of BDC digits, thus dividing it by a power of ten.



FLAGS:
- C: Unaffected
- Z: Set if the result is zero; cleared otherwise
- S: Set if the most significant bit of the result is set; cleared otherwise
- V: Unaffected
- D: Unaffected
- H: Unaffected

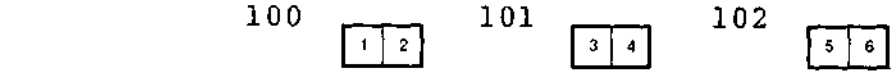
CYCLES:

Byte
9

NOTE: The same register must not be used in both the source and destination addressing mode designators.

RRDB
Rotate Right Digit

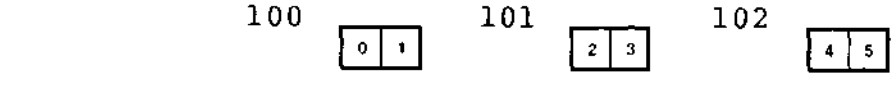
EXAMPLE: If location 100 contains the BCD digits 1,2 (00010010), location 101 contains 3,4 (00110100), and location 102 contains 5,6 (01010110)



the sequence of statements

```
     LD     R3,#3     !Set loop counter for 3 bytes (6
                       digits)!
     LD     R2,#100   !Set pointer to high-order digits!
     CLRB   RH1       !Zero-fill high-order digit!
LOOP:
     LDB    RL1,@R2   !Get next two digits!
     RRDB   RH1,RL1   !Shift digits right one position!
     LDB    @R2,RL1   !Replace shifted digits!
     INC    R2        !Advance pointer!
     DJNZ   R3,LOOP   !Repeat until counter is zero!
```

will leave the digits 0,1 (00000001) in location 100, the digits 2,3 (00100011) in location 101, and the digits 4,5 (01000101) in location 102.

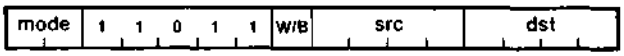


```
SBC   dst,src                    dst: R
SBCB                             src: R
```

INSTRUCTION FORMAT: (F2.1)

													mode	dst	src
W/B:	mode	1	1	0	1	1		src			dst		10	R	R

OPERATION:



The source operand, along with the setting of the carry flag, is subtracted from the destination operand and the result is stored in the destination. The contents of the source are not affected. Subtraction is performed by adding the two's complement of the source operand to the destination operand. In multiple precision arithmetic, this instruction permits the carry ("borrow") from the subtraction of low-order operands to be subtracted from the subtraction of high-order operands.

FLAGS:
- C: Cleared if there is a carry from the most significant bit of the result; set otherwise, indicating a "borrow"
- Z: Set if the result is zero; cleared otherwise
- S: Set if the result is negative; cleared otherwise
- V: Set if arithmetic overflow occurs, that is, if the operands were of opposite signs and the sign of the result is the same as the sign of the source; cleared otherwise
- D: SBC - unaffected; SBCB - set
- H: SBC - unaffected; SBCB - cleared if there is a carry from the most significant bit of the low-order four bits of the result; set otherwise, indicating a "borrow"

CYCLES:

Word/Byte
5

EXAMPLE: Long subtraction may be done with the following instruction sequence, assuming R0,R1 contain one operand and R2,R3 contain the other operand:

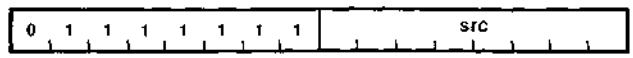
```
SUB   R1,R3   !subtract low-order words!
SBC   R0,R2   !subtract carry and high-order words!
```

If R0 contains %0038, R1 contains %4000, R2 contains %000A and R3 contains %F000, then the above two instructions leave the value %002D in R0 and %5000 in R1.

SC
System Call

SC src src: IM

INSTRUCTION FORMAT: (F9.5)



src
IM

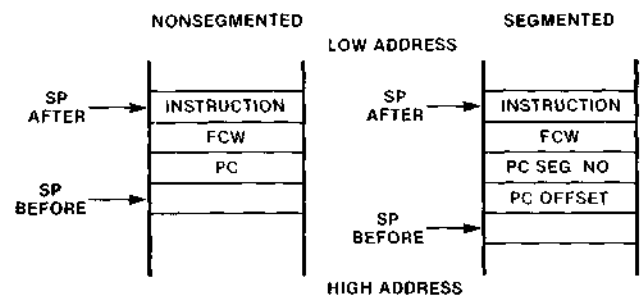
OPERATION:

Nonsegmented	Segmented
SP <- SP - 4	SP <- SP - 6
@SP <- PS	@SP <- PS
SP <- SP - 2	SP <- SP - 2
@SP <- instruction	@SP <- instruction
PS <- System Call PS	PS <- System Call PS

This instruction is used for controlled access to operating system software in a manner similar to a trap or interrupt. The current program status (PS) is pushed on the system processor stack, and then the instruction itself is pushed which includes the source operand (an 8-bit value). The PS includes the Flags and Control Word (FCW), and the updated program counter (PC). (The updated program counter value used is the address of the first instruction byte following the SC instruction.)

The system stack pointer is always used (R15 if nonsegmented, or RR14 if segmented), regardless of whether system or normal mode is in effect. The new PS is then loaded from the Program Status block associated with the System Call trap (see section 1.6.4), and control is passed to the procedure whose address is the program counter value contained in the new PS. This procedure may inspect the source operand on the top of the stack to determine the particular software service desired.

The following figure illustrates the format of the saved program status in the system stack:



FLAGS: No flags affected

CYCLES:

Address		
NS	SS	SL
33	--	39

NOTES: The Z8001 version always executes the segmented mode of the System Call instruction, regardless of the current mode, and sets the Segmentation Mode bit (SEG) to segmented mode (=1) at the start of the SC instruction execution. Both the Z8001 and Z8002 versions set the System/Normal Mode bit (S/N) to System mode (=1) at the start of the SC instruction execution. The status pins reflect the setting of these control bits during the execution of the SC instruction. However, the setting of SEG and S/N does not affect the value of these bits in the old FCW pushed on the stack. The new value of the FCW is not effective until the next instruction, so that the status pins will not be affected by the new control bits until after the SC instruction execution is completed.

The "src field" in the instruction format encoding contains the source operand. The "src field" values range from 0 to 255 corresponding to the source values 0 to 255.

EXAMPLE: In the nonsegmented Z8002 version, if the contents of the program counter are %1000, the contents of the system stack pointer (R15) are %3006, and the program counter value associated with the System Call trap in the Program Status Area is %2000, the statement

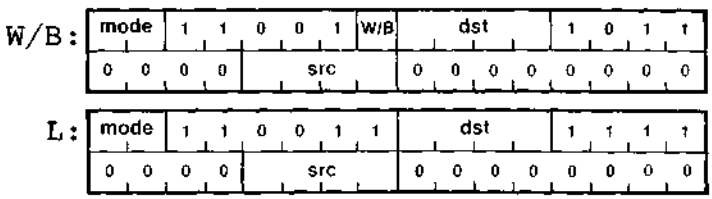
```
SC  #3  !System call,request code=3!
```

causes the system stack pointer to be decremented to %3000. Location %3000 contains %7F03 (0111111100000011, the SC instruction). Location %3002 contains the old FCW, and location %3004 contains %1002 (the address of the instruction following the SC instruction). System mode is in effect, and the program counter contains the value %2000, which is the start of a System Call trap handler.

SDA
Shift Dynamic Arithmetic

```
SDA   dst,src                    dst: R
SDAB                             src: R
SDAL
```

INSTRUCTION FORMAT: (F6.6)

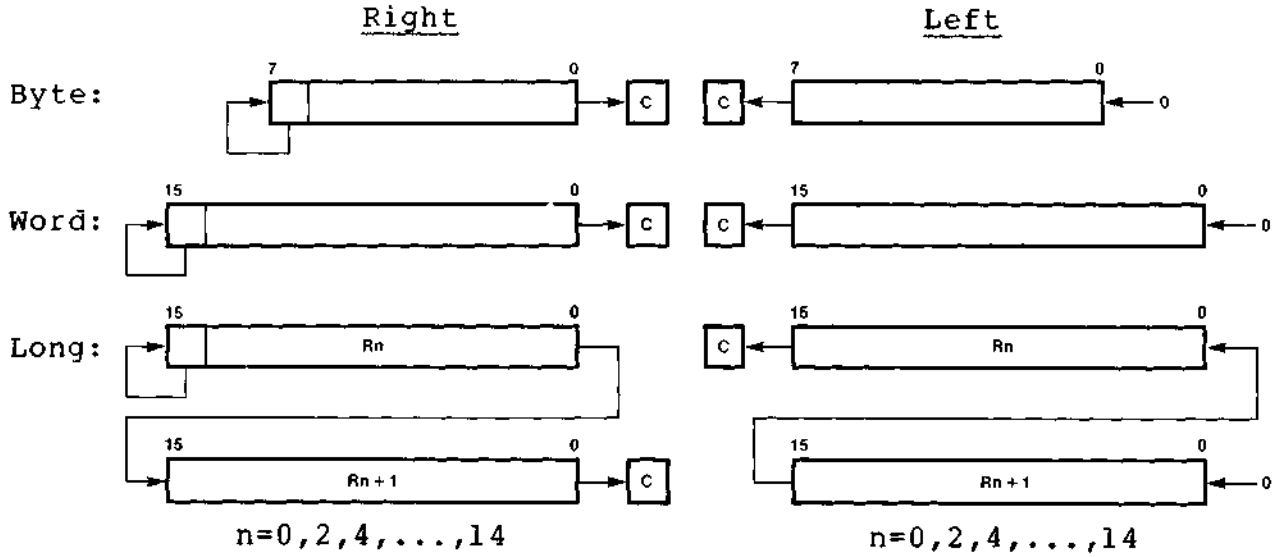


mode	dst	src
10	R	R

OPERATION:

```
Right  repeat src times:
dst(msb) <- dst(msb)
dst(n) <- dst(n+1) {n is 0 to msb-1}
C <- dst(0)

Left   repeat src times:
dst(0) <- 0
dst(n+1) <- dst(n) {n is 0 to msb-1}
C <- dst(msb)
```

The destination operand is shifted arithmetically right or left the number of bit positions specified by the source operand (a word register). The shift count varies from -8 to +8 for SDAB, from -16 to +16 for SDA, and from -32 to +32 for SDAL, where a negative value is a right shift and a positive value is a left shift. A shift of zero positions does not affect the destination; however, the flags are set according to the destination value. The sign bit is replicated in shifts to the right, and the C flag is loaded from bit 0 of the destination. The least significant bit is filled with 0 in shifts to the left, and the C flag is loaded from the sign bit of the destination.



FLAGS:
- C: Set if the last bit shifted from the destination was 1, cleared otherwise
- Z: Set if the result is zero; cleared otherwise
- S: Set if the result is negative; cleared otherwise
- V: Set if arithmetic overflow occurs, that is, if the sign of the destination changed during shifting; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte	Long	
15+3*n	15+3*n	(n=number of bit positions, where 0 is equivalent to 1)

NOTE: The source operand is represented as a 16-bit two's complement value. For each operand size, the operation is undefined if the value is not in the range specified above.

EXAMPLE: If register R5 contains %C705 (1100011100000101) and register R1 contains -2 (%FFFE or 1111111111111110), the statement

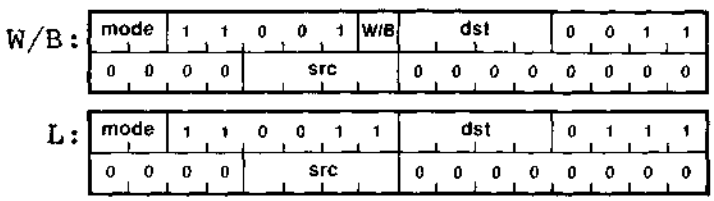
```
SDA  R5,R1
```

performs an arithmetic right shift of two bit positions, leaves the value %F1C1 (1111000111000001) in R5, and clears the carry flag.

SDL
Shift Dynamic Logical

```
SDL   dst,src                    dst: R
SDLB                             src: R
SDLL
```

INSTRUCTION FORMAT: (F6.6)

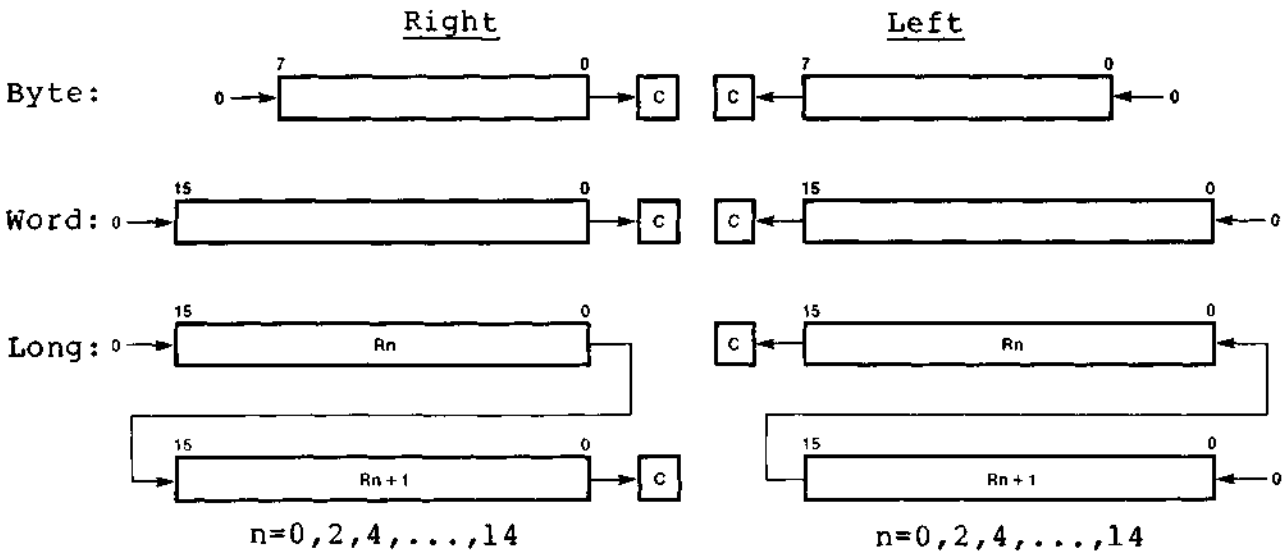


mode	dst	src
10	R	R

OPERATION:

```
Right  repeat src times:
dst(msb) <- 0
dst(n) <- dst(n+1) {n is 0 to msb-1}
C <- dst(0)

Left   repeat src times:
dst(0) <- 0
dst(n+1) <- dst(n) {n is 0 to msb-1}
C <- dst(msb)
```

The destination operand is shifted logically right or left the number of bit positions specified by the source operand (a word register). The shift count varies from -8 to +8 for SDLB, from -16 to +16 for SDL, and from -32 to +32 for SDLL, where a negative value is a right shift and a positive value is a left shift. A shift of zero positions does not affect the destination; however, the flags are set according to the destination value. The most significant bit is filled with 0 in shifts to the right, and the C flag is loaded from bit 0 of the destination. The least significant bit is filled with 0 in shifts to the left, and the C flag is loaded from the most significant bit of the destination.



FLAGS:
- C: Set if the last bit shifted from the destination was 1; cleared otherwise
- Z: Set if the result is zero; cleared otherwise
- S: Set if the most significant bit of the result is set; cleared otherwise
- V: Undefined
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte	Long	
15+3*n	15+3*n	(n=number of bit positions, where 0 is equivalent to 1)

NOTE: The source operand is represented as a 16-bit two's complement value. For each operand size, the operation is undefined if the value is not in the range specified above.

EXAMPLE: If register RL5 contains %B3 (10110011) and register R1 contains 4 (0000000000000100), the statement

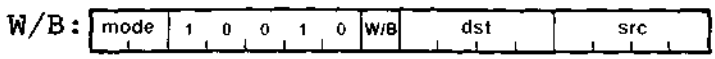
```
SDLB  RL5,R1
```

performs a logical left shift of four bit positions, leaves the value %30 (00110000) in RL5, and sets the carry flag.

SET
Set Bit

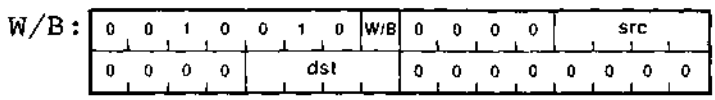
```
SET   dst,src                    dst: R, IR, DA, X
SETB                             src: R, IM
```

INSTRUCTION FORMAT: (F1.2)



mode	dst	src
10	R	IM
00	IR	IM (dst field<>0)
01	DA	IM (dst field=0)
01	X	IM (dst field<>0)

(F6.3)



dst	src
R	R

OPERATION: dst(src) <- 1

Sets the specified bit within the destination operand without affecting any other bits in the destination. The source (the bit number) can be specified as either an immediate value, or as a word register which contains the value. In the second case, the destination operand must be a register, and the source operand must be R0 through R7 for SETB, or R0 through R15 for SET. The bit number is a value from 0 to 7 for SETB or 0 to 15 for SET, with 0 indicating the least significant bit.

FLAGS: No flags affected

CYCLES:

dst	src	Word/Byte NS	SS	SL
R	IM	4	--	--
IR	IM	11	--	--
DA	IM	13	14	16
X	IM	14	14	17
R	R	10	--	--

NOTE: Only the lower four bits of the source operand are used to specify the bit number for SET, while only the lower three bits of the source operand are used with SETB. When the source operand is an immediate value, the "src field" in the instruction format encoding contains the bit number in the lowest four bits for SET, or the lowest three bits for SETB.

EXAMPLE: If register RL3 contains %B2 (10110010) and register R2 contains the value 6, the statement

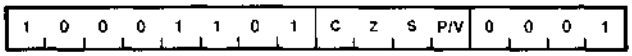
```
SETB  RL3,R2
```

will leave the value %F2 (11110010) in RL3.

SETFLG
Set Flag

SETFLG flag flag: C, Z, S, P, V

INSTRUCTION FORMAT: (F9.1)



OPERATION: FLAGS(4:7) <- FLAGS(4:7) OR instruction(4:7)

Any combination of the C, Z, S, P or V flags are set to one if the corresponding bit in the instruction is one. If the corresponding bit in the instruction is zero, the flag will not be affected. All other bits in the FLAGS register are unaffected. Note that the P and V flags are represented by the same bit. There may be one, two, three or four operands in the assembly language statement, in any order.

FLAGS:
C: Set if specified; unaffected otherwise
Z: Set if specified; unaffected otherwise
S: Set if specified; unaffected otherwise
P/V: Set if specified; unaffected otherwise
D: Unaffected
H: Unaffected

CYCLES: 7

EXAMPLE: If the C, Z, and S flags are all clear (=0), and the P flag is set (=1), the statement

```
SETFLG  C
```

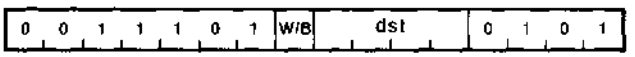
will leave the C and P flags set (=1), and the Z and S flags cleared (=0).

SIN
Special Input

SIN dst,src
SINB

dst: R
src: DA

INSTRUCTION FORMAT: (F7.1)



dst	src
R	DA

OPERATION: dst <- src

The contents of the source operand are loaded into the destination. I/O addresses are always 16 bits. This instruction is used to load information from the Memory Management Unit.

FLAGS: No flags affected

CYCLES:

Word/Byte
12

NOTES: This is a privileged instruction.

The status pins indicate a special I/O reference.

SIND dst,src,r
SINDB

dst: IR
src: IR

INSTRUCTION FORMAT: (F6.4)

W/B:

dst	src	r
IR	IR	R

OPERATION:
dst <- src
AUTODECREMENT dst {-1 if byte, -2 if word}
r <- r - 1

This instruction is used for block input of strings of data, typically status information from the Memory Management Unit. The contents of the I/O location addressed by the source register are loaded into the memory location addressed by the destination register. I/O addresses are always 16 bits. The destination register is then decremented by one if SINDB, or by two if SIND, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS:
C: Unaffected
Z: Undefined
S: Unaffected
V: Set if the result of decrementing r is zero; cleared otherwise
D: Unaffected
H: Unaffected

CYCLES:

Word/Byte
21

NOTES: This is a privileged instruction.

The status pins indicate a special I/O reference.

SINDR
Special Input, Decrement and Repeat

```
SINDR  dst,src,r                dst: IR
SINDRB                          src: IR
```

INSTRUCTION FORMAT: (F6.4)

W/B:

dst	src	r
IR	IR	R

```
OPERATION:      dst <- src
                AUTODECREMENT dst {-1 if byte, -2 if word}
                r <- r - 1
                repeat until r = 0
```

This instruction is used for block input of strings of data, typically status information from the Memory Management Unit. The contents of the I/O location addressed by the source register are loaded into the memory location addressed by the destination register. I/O addresses are always 16 bits. The destination register is then decremented by one if SINDRB, or by two if SINDR, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can input from 1 to 65536 bytes or 32768 words (the value for r must not be greater than 32768 for SINDR).

```
FLAGS:  C: Unaffected
        Z: Undefined
        S: Unaffected
        V: Set
        D: Unaffected
        H: Unaffected
```

CYCLES:

Word/Byte
11+10*n (n=number of data elements transferred)

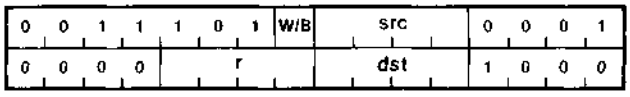
NOTES: This is a privileged instruction.

The status pins indicate a special I/O reference.

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

```
SINI  dst,src,r                dst: IR
SINIB                          src: IR
```

INSTRUCTION FORMAT: (F6.4)



dst	src	r
IR	IR	R

OPERATION:

```
dst <- src
AUTOINCREMENT dst {+1 if byte, +2 if word}
r <- r - 1
```

This instruction is used for block input of strings of data, typically status information from the Memory Management Unit. The contents of the I/O location addressed by the source register are loaded into the memory location addressed by the destination register. I/O addresses are always 16 bits. The destination is then incremented by one if SINIB, or by two if SINI, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS:
- C: Unaffected
- Z: Undefined
- S: Unaffected
- V: Set if the result of decrementing r is zero; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte
21

NOTES: This is a privileged instruction.

The status pins indicate a special I/O reference.

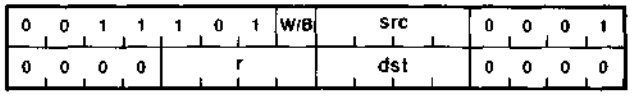
SINIR
Special Input, Increment and Repeat

```
SINIR  dst,src,r                dst: IR
SINIRB                          src: IR
```

INSTRUCTION FORMAT: (F6.4)

	dst	src	r
	IR	IR	R

W/B:



OPERATION:
```
dst <- src
AUTOINCREMENT dst {+1 if byte, +2 if word}
r <- r - 1
repeat until r = 0
```

This instruction is used for block input of strings of data, typically status information from the Memory Management Unit. The contents of the I/O location addressed by the source register are loaded into the memory location addressed by the destination register. I/O addresses are always 16 bits. The destination register is then incremented by one if SINIRB, or by two if SINIR, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can input from 1 to 65536 bytes or 32768 words (the value for r must not be greater than 32768 for SINIR).

FLAGS:
```
C: Unaffected
Z: Undefined
S: Unaffected
V: Set
D: Unaffected
H: Unaffected
```

CYCLES:

Word/Byte
11+10*n

(n=number of data elements transferred)

NOTES: This is a privileged instruction.

The status pins indicate a special I/O reference.

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

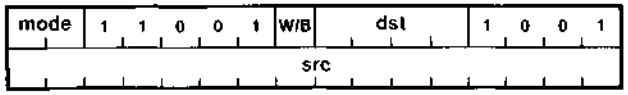
SLA
Shift Left Arithmetic

```
SLA   dst,src                    dst: R
SLAB                             src: IM
SLAL
```

INSTRUCTION FORMAT: (F5.1)

mode	dst	src
10	R	IM

W/B:

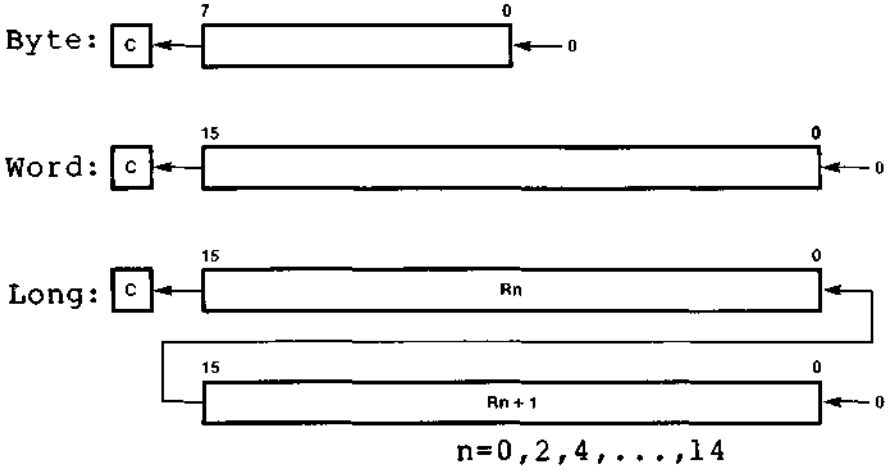


L:

OPERATION:

```
repeat src times:
dst(0) <- 0
dst(n+1) <- dst(n)   {n is 0 to msb-1}
C <- dst(msb)
```

The destination operand is shifted arithmetically left the number of bit positions specified by the source operand. For SLAB, the source is in the range 0 to 8; for SLA, the source is in the range 0 to 16; for SLAL, the source is in the range 0 to 32. A shift of zero positions does not affect the destination; however, the flags are set according to the destination value. The least significant bit of the destination is filled with 0, and the C flag is loaded from the sign bit of the destination. This instruction performs a signed multiplication of the destination by a power of two with overflow indication. The source operand may be omitted from the assembly language statement and thus defaults to the value 1.



SLA
Shift Left Arithmetic

FLAGS:
- C: Set if the last bit shifted from the destination was 1; cleared otherwise
- Z: Set if the result is zero; cleared otherwise
- S: Set if the result is negative; cleared otherwise
- V: Set if arithmetic overflow occurs, that is, if the sign of the destination changed during shifting; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte	Long	
13+3*n	13+3*n	(n=number of bit positions, where 0 is equivalent to 1)

NOTE: The "src field" is encoded in the instruction format as the 16-bit two's complement positive value of the source operand. For each operand size, the operation is undefined if the source operand is not in the range specified above.

EXAMPLE: If register pair RR2 contains %1234ABCD, the statement

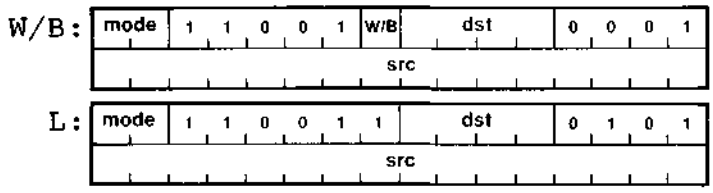
```
SLAL  RR2,#8
```

will leave the value %34ABCD00 in RR2 and clear the carry flag.

SLL
Shift Left Logical

```
SLL   dst,src                    dst: R
SLLB                             src: IM
SLLL
```

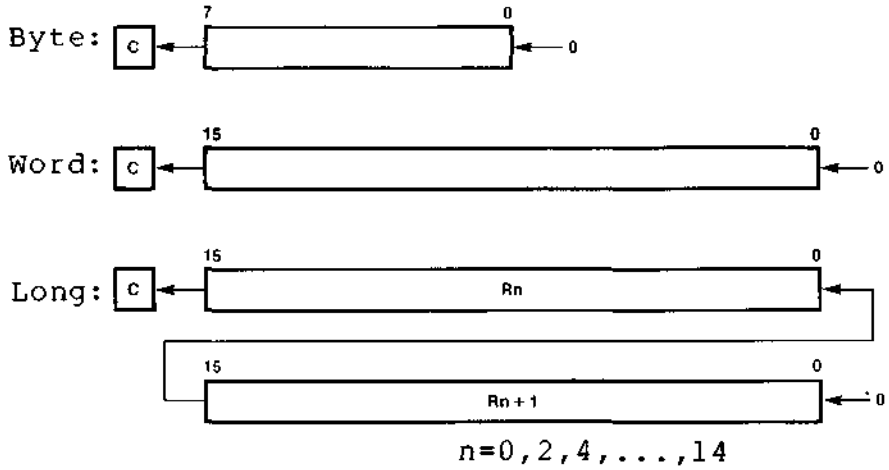
INSTRUCTION FORMAT: (F5.1)



mode	dst	src
10	R	IM

```
OPERATION:      repeat src times:
                dst(0) <- 0
                dst(n+1) <- dst(n)   {n is 0 to msb-1}
                C <- dst(msb)
```

The destination operand is shifted logically left the number of bit positions specified by the source operand. For SLLB, the source is in the range 0 to 8; for SLL, the source is in the range 0 to 16; for SLLL, the source is in the range 0 to 32. A shift of zero positions does not affect the destination; however, the flags are set according to the destination value. The least significant bit of the destination is filled with 0, and the C flag is loaded from the most significant bit of the destination. This instruction performs an unsigned multiplication of the destination by a power of two. The source operand may be omitted from the assembly language statement and thus defaults to the value 1.



SLL
Shift Left Logical

FLAGS:
- C: Set if the last bit shifted from the destination was 1; cleared otherwise
- Z: Set if the result is zero; cleared otherwise
- S: Set if the most significant bit of the result is set; cleared otherwise
- V: Undefined
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte	Long	
13+3*n	13+3*n	(n=number of bit positons, where 0 is equivalent to 1)

NOTE: The "src field" is encoded in the instruction format as the 16-bit two's complement positive value of the source operand. For each operand size, the operation is undefined if the source operand is not in the range specified above.

EXAMPLE: If register R3 contains %4321 (0100001100100001), the statement

```
SLL  R3,#1
```

will leave the value %8642 (1000011001000010) in R3 and clear the carry flag.

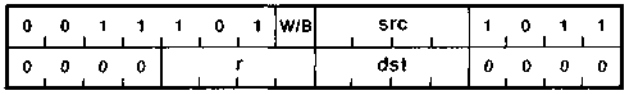
SOTDR
Special Output, Decrement and Repeat

```
SOTDR  dst,src,r                dst: IR
SOTDRB                          src: IR
```

INSTRUCTION FORMAT: (F6.4)

dst	src	r
IR	IR	R

W/B:



OPERATION:
```
dst <- src
AUTODECREMENT src {-1 if byte, -2 if word}
r <- r - 1
repeat until r = 0
```

This instruction is used for block output of strings of data, typically information to the Memory Management Unit. The contents of the memory location addressed by the source register are loaded into the I/O location addressed by the destination register. I/O addresses are always 16 bits. The source register is then decremented by one if SOTDRB, or by two if SOTDR, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can output from 1 to 65536 bytes or 32768 words (the value for r must not be greater than 32768 for SOTDR).

FLAGS:
```
C: Unaffected
Z: Undefined
S: Unaffected
V: Set
D: Unaffected
H: Unaffected
```

CYCLES:

Word/Byte
11+10*n (n=number of data elements transferred)

NOTES: This is a privileged instruction.

The status pins indicate a special I/O reference.

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

SOTIR
Special Output, Increment and Repeat

```
SOTIR  dst,src,r                dst: IR
SOTIRB                          src: IR
```

INSTRUCTION FORMAT: (F6.4)

dst	src	r
IR	IR	R

W/B:

OPERATION:

```
dst <- src
AUTOINCREMENT src {+1 if byte, +2 if word}
r <- r - 1
repeat until r = 0
```

This instruction is used for block output of strings of data, typically information to the Memory Management Unit. The contents of the memory location addressed by the source register are loaded into the I/O location addressed by the destination register. I/O addresses are always 16 bits. The source register is then incremented by one if SOTIRB, or by two if SOTIR, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can output from 1 to 65536 bytes or 32768 words (the value for r must not be greater than 32768 for SOTIR).

FLAGS:
- C: Unaffected
- Z: Undefined
- S: Unaffected
- V: Set
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte
11+10*n (n=number of data elements transferred)

NOTES:

This is a privileged instruction.

The status pins indicate a special I/O reference.

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

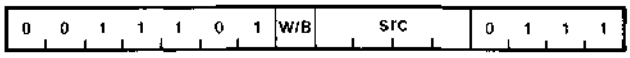
SOUT
Special Output

SOUT dst,src
SOUTB

dst: DA
src: R

INSTRUCTION FORMAT: (F7.2)

W/B:



dst	src
DA	R

OPERATION: dst <- src

The contents of the source operand are loaded into the destination. I/O addresses are always 16 bits. This instruction is used to load information into the Memory Management Unit.

FLAGS: No flags affected

CYCLES:

Word/Byte
12

NOTES: This is a privileged instruction.

The status pins indicate a special I/O reference.

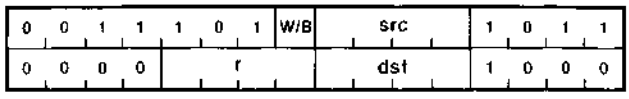
SOUTD
Special Output and Decrement

```
SOUTD  dst,src,r                dst: IR
SOUTDB                          src: IR
```

INSTRUCTION FORMAT: (F6.4)

	dst	src	r
	IR	IR	R

W/B:



OPERATION:
```
dst <- src
AUTODECREMENT src {-1 if byte, -2 if word}
r <- r - 1
```

This instruction is used for block output of strings of data, typically information to the Memory Management Unit. The contents of the memory location addressed by the source register are loaded into the I/O location addressed by the destination register. I/O addresses are always 16 bits. The source register is then decremented by one if SOUTDB, or by two if SOUTD, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS:
- C: Unaffected
- Z: Undefined
- S: Unaffected
- V: Set if the result of decrementing r is zero; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte
21

NOTES: This is a privileged instruction.

The status pins indicate a special I/O reference.

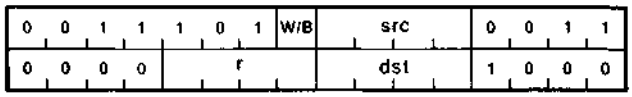
SOUTI
Special Output and Increment

```
SOUTI  dst,src,r                dst: IR
SOUTIB                          src: IR
```

INSTRUCTION FORMAT: (F6.4)

dst	src	r
IR	IR	R

W/B:



OPERATION:
```
dst <- src
AUTOINCREMENT src {+1 if byte, +2 if word}
r <- r - 1
```

This instruction is used for block output of strings of data, typically information to the Memory Management Unit. The contents of the memory location addressed by the source register are loaded into the I/O location addressed by the destination register. I/O addresses are always 16 bits. The source register is then incremented by one if SOUTIB, or by two if SOUTI, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

FLAGS:
- C: Unaffected
- Z: Undefined
- S: Unaffected
- V: Set if the result of decrementing r is zero; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Word/Byte
21

NOTES: This is a privileged instruction.

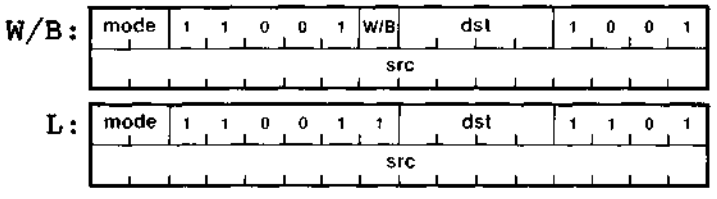
The status pins indicate a special I/O reference.

SRA
Shift Right Arithmetic

```
SRA   dst,src                dst: R
SRAB                         src: IM
SRAL
```

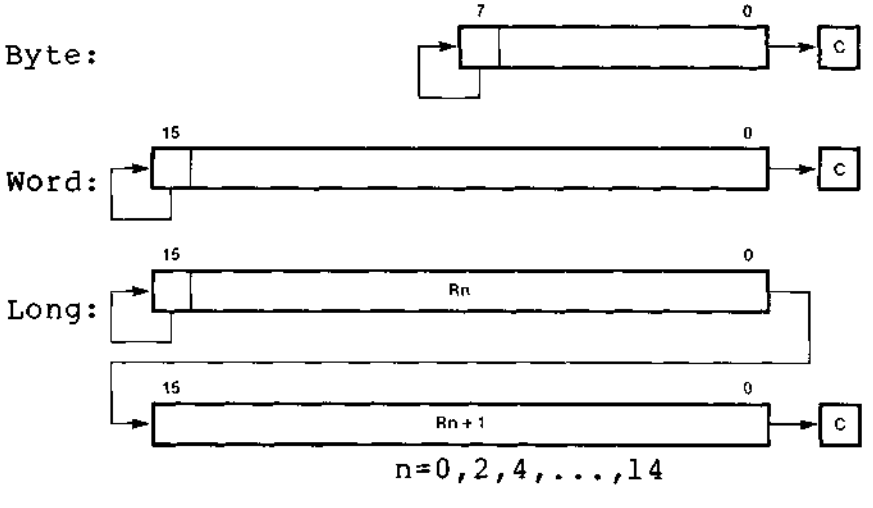
INSTRUCTION FORMAT: (F5.1)

mode	dst	src
10	R	IM



OPERATION:
```
repeat src times:
dst(msb) <- dst(msb)
dst(n) <- dst(n+1)   {n is 0 to msb-1}
C <- dst(0)
```

The destination operand is shifted arithmetically right the number of bit positions specified by the source operand. For SRAB, the source is in the range 0 to 8; for SRA, the source is in the range 0 to 16; for SRAL, the source is in the range 0 to 32. A shift of zero positions does not affect the destination, however, the flags are se according to the destination value. The sign bit of the destination is replicated, and the C flag is loaded from bit 0 of the destination. The source operand may be omitted from the assembly language statement and thus defaults to the value 1.



FLAGS: C: Set if the last bit shifted from the destination was 1; cleared otherwise
Z: Set if the result is zero; cleared otherwise
S: Set if the result is negative; cleared otherwise
V: Cleared
D: Unaffected
H: Unaffected

CYCLES:

Word/Byte	Long	
13+3*n	13+3*n	(n=number of bit positions, where 0 is equivalent to 1)

NOTE: The "src field" is encoded in the instruction format as the 16-bit two's complement negative value of the source operand. For each operand size, the operation is undefined if the source operand is not in the range specified above.

EXAMPLE: If register RH6 contains %3B (00111011), the statement

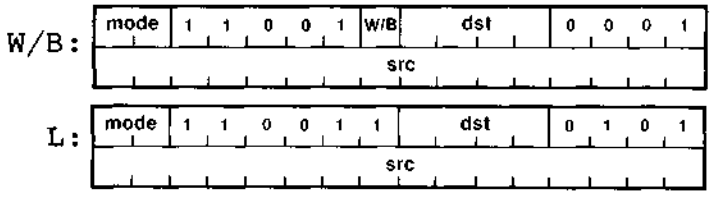
```
SRAB  RH6,#2
```

will leave the value %0E (00001110) in RH6 and set the carry flag.

SRL
Shift Right Logical

```
SRL   dst,src                dst: R
SRLB                         src: IM
SRLL
```

INSTRUCTION FORMAT: (F5.1)

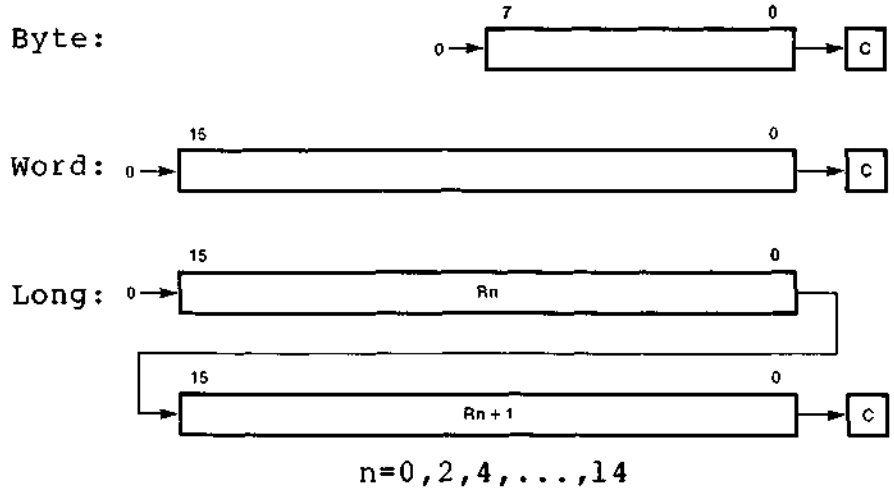


mode	dst	src
10	R	IM

OPERATION:

```
repeat src times:
dst(msb) <- 0
dst(n) <- dst(n+1) {n is 0 to msb-1}
C <- dst(0)
```

The destination operand is shifted logically right the number of bit positions specified by the source operand. For SRLB, the source operand is in the range 0 to 8; for SRL, the source is in the range 0 to 16; for SRLL, the source is in the range 0 to 32. A shift of zero positions does not affect the destination, however, the flags are se according to the destination value. The most significant bit of the destination is filled with 0, and the C flag is loaded from bit 0 of the destination. This instruction performs an unsigned division of the destination by a power of two. The source operand may be omitted from the assembly language statement and thus defaults to the value of 1.



n=0,2,4,...,14

FLAGS:. C: Set if the last bit shifted from the destination was 1; cleared otherwise
Z: Set if the result is zero; cleared otherwise
S: Set if the most significant bit of the result is set; cleared otherwise
V: Undefined
D: Unaffected
H: Unaffected

CYCLES:

Word/Byte	Long	
13+3*n	13+3*n	(n=number of bit positions, where 0 is equivalent to 1)

NOTE: The "src field" is encoded in the instruction format as the 16-bit two's complement negative value of the source operand. For each operand size, the operation is undefined if the source operand is not in the range specified above.

EXAMPLE: If register R0 contains %1111 (0001000100010001), the statement

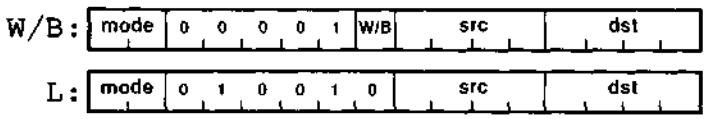
```
SRL  R0,#6
```

will leave the value %0044 (0000000001000100) in R0 and clear the carry flag.

SUB
Subtract

```
SUB   dst,src                dst: R
SUBB                         src: R, IM, IR, DA, X
SUBL
```

INSTRUCTION FORMAT: (F2.1)



mode	dst	src
10	R	R
00	R	IM (src field=0)
00	R	IR (src field<>0)
01	R	DA (src field=0)
01	R	X (src field<>0)

OPERATION: dst <- dst - src

The source operand is subtracted from the destination operand and the result is stored in the destination. The contents of the source are not affected. Subtraction is performed by adding the two's complement of the source operand to the destination operand.

FLAGS:
- C: Cleared if there is a carry from the most significant bit; set otherwise, indicating a "borrow"
- Z: Set if the result is zero; cleared otherwise
- S: Set if the result is negative; cleared otherwise
- V: Set if arithmetic overflow occurs, that is, if the operands were of opposite signs and the sign of the result is the same as the sign of the source; cleared otherwise
- D: SUB,SUBL - unaffected; SUBB - set
- H: SUB,SUBL - unaffected; SUBB - cleared if there is a carry from the most significant bit of the low-order four bits of the result; set otherwise, indicating a "borrow"

CYCLES:

src	Word/Byte			Long		
	NS	SS	SL	NS	SS	SL
R	4	--	--	8	--	--
IM	7	--	--	14	--	--
IR	7	--	--	14	--	--
DA	9	10	12	15	16	18
X	10	10	13	16	16	19

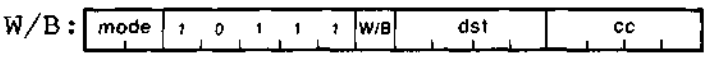
EXAMPLE: If register R0 contains %0344, the statement

```
SUB  R0,#%AA
```

will leave the value %029A in R0.

TCC cc,dst dst: R
TCCB

INSTRUCTION FORMAT: (F1.4)



mode: 10 dst: R

OPERATION: if cc is true then dst(0) <- 1

This instruction is used to create a Boolean data value based on the condition code resulting from a previous operation. The condition code specified by "cc" is tested. If the condition is true, then the least significant bit of the destination is set; otherwise it is unaffected. See section 3.2.1 for a list of condition codes. All other bits in the destination are unaffected.

FLAGS: No flags affected

CYCLES:

Word/Byte
5

EXAMPLE: If register R1 contains 0, and the Z flag is set, the statement

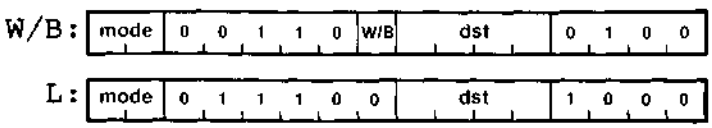
```
TCC  EQ,R1
```

will leave the value 1 in R1.

TEST
Test

```
TEST  dst                          dst: R, IR, DA, X
TESTB
TESTL
```

INSTRUCTION FORMAT: (F1.1)



mode	dst
10	R
00	IR
01	DA (dst field=0)
01	X (dst field<>0)

OPERATION: dst OR 0

The destination operand is tested (logically ORed with zero), and the appropriate flags set accordingly, which may then be used for arithmetic and logical conditional jumps. The contents of the destination are not affected.

FLAGS:
- C: Unaffected
- Z: Set if the result is zero; cleared otherwise
- S: Set if the most significant bit of the result is set; cleared otherwise
- P: TEST - unaffected; TESTL - undefined; TESTB - set if parity of the result is even; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

dst	Word/Byte			Long		
	NS	SS	SL	NS	SS	SL
R	7	--	--	13	--	--
IR	8	--	--	13	--	--
DA	11	12	14	16	17	19
X	12	12	15	17	17	20

EXAMPLE: If register R5 contains %FFFF (1111111111111111), the statement

```
TEST  R5
```

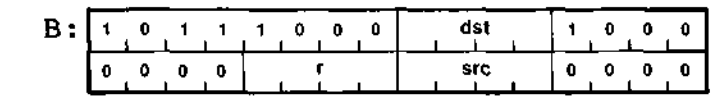
will set the S flag, clear the Z flag, and leave the other flags unaffected.

TRDB
Translate and Decrement

TRDB dst,src,r

dst: IR
src: IR

INSTRUCTION FORMAT: (F6.4)



dst	src	r
IR	IR	R

OPERATION:
```
dst <- src[dst]
AUTODECREMENT dst {-1}
r <- r - 1
```

This instruction is used to translate a string of bytes from one code to another code. The contents of the location addressed by the destination register (the "target byte") is used as an index into a table of translation values whose lowest address is contained in the source register. The index is computed by adding the target byte to the address contained in the source register. The addition is performed following the rules for addressing mode arithmetic, with the target byte treated as an unsigned 8-bit value extended with high-order zeros. The sum is used as the address of an 8-bit translation value within the table which replaces the original contents of the location addressed by the destination register.

The destination register is then decremented by one, thus moving the the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

Because the 8-bit target byte is added to the source register to obtain the address of a translation value, the table may contain 256 bytes. A smaller table size may be used where it is known that not all possible 8-bit target byte values will occur.

FLAGS:
- C: Unaffected
- Z: Undefined
- S: Unaffected
- V: Set if the result of decrementing r is zero; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Byte
25

TRDB
Translate and Decrement

NOTE: The original contents of register RH1 is lost, and is replaced by an undefined value.

EXAMPLE: If register R6 contains %4001, the byte at location %4001 contains 3, register R9 contains %1000, the byte at location %1003 contains %AA, and register R12 contains 2, the statement

```
TRDB  @R6,@R9,R12
```

will leave the value %AA in location %4001, the value %4000 in R6, and the value 1 in R12. R9 will not be affected.

TRDRB
Translate, Decrement and Repeat

TRDRB dst,src,r

dst: IR
src: IR

INSTRUCTION FORMAT: (F6.4)

dst	src	r
IR	IR	R

OPERATION:
```
dst <- src[dst]
AUTODECREMENT dst {-1}
r <- r - 1
repeat until r = 0
```

This instruction is used to translate a string of bytes from one code to another code. The contents of the location addressed by the destination register (the "target byte") is used as an index into a table of translation values whose lowest address is contained in the source register. The index is computed by adding the target byte to the address contained in the source register. The addition is performed following the rules for addressing mode arithmetic, with the target byte treated as an unsigned 8-bit value extended with high-order zeros. The sum is used as the address of an 8-bit translation value within the table which replaces the original contents of the location addressed by the destination register.

The destination register is then decremented by one, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can translate from 1 to 65536 bytes.

Because the 8-bit target byte is added to the source register to obtain the address of a translation value, the table may contain 256 bytes. A smaller table size may be used where it is known that not all possible 8-bit target byte values will occur.

FLAGS:
C: Unaffected
Z: Undefined
S: Unaffected
V: Set
D: Unaffected
H: Unaffected

TRDRB
Translate, Decrement and Repeat

CYCLES: Byte
11+14*n (n=number of data elements translated)

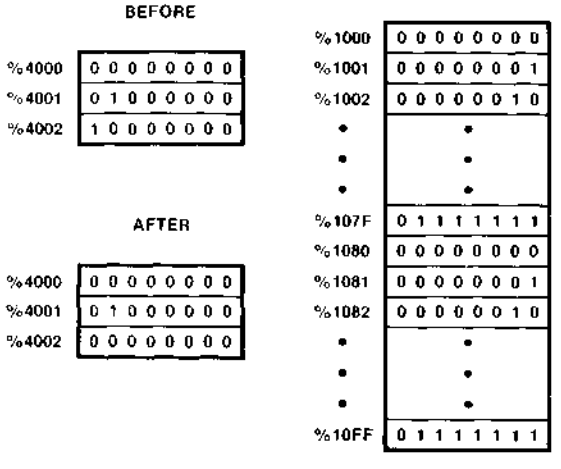
NOTES: The original contents of register RH1 is lost, and is replaced by an undefined value.

This instruction can be interrupted after each execution of the basic operation. The program counter of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

EXAMPLE: If register R6 contains %4002, the bytes at locations %4000 through %4002 contain the values %00, %40, %80, respectively, register R9 contains %1000, the translation table from location %1000 through %10FF contains 0,1,2,...,%7F,0,1,2,...,%7F (the second zero is located at %1080), and register R12 contains 3, the statement

```
TRDRB  @R6,@R9,R12
```

will leave the values %00, %40, %00 in locations %4000 through %4002, respectively. Register R6 will contain %3FFF, and R12 will contain 0. R9 will not be affected.

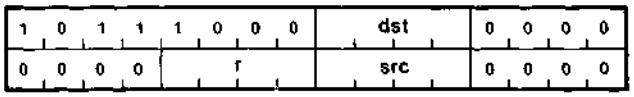


TRIB
Translate and Increment

TRIB dst,src,r

dst: IR
src: IR

INSTRUCTION FORMAT: (F6.4)



dst	src	r
IR	IR	R

OPERATION:
dst <- src[dst]
AUTOINCREMENT dst {+1}
r <- r - 1

This instruction is used to translate a string of bytes from one code to another code. The contents of the location addressed by the destination register (the "target byte") is used as an index into a table of translation values whose lowest address is contained in the source register. The index is computed by adding the target byte to the address contained in the source register. The addition is performed following the rules for addressing mode arithmetic, with the target byte treated as an unsigned 8-bit value extended with high-order zeros. The sum is used as the address of an 8-bit translation value within the table which replaces the original contents of the location addressed by the destination register.

The destination register is then incremented by one, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

Because the 8-bit target byte is added to the source register to obtain the address of a translation value, the table may contain 256 bytes. A smaller table size may be used where it is known that not all possible 8-bit target byte values will occur.

FLAGS:
C: Unaffected
Z: Undefined
S: Unaffected
V: Set if the result of decrementing r is zero; cleared otherwise
D: Unaffected
H: Unaffected

CYCLES:

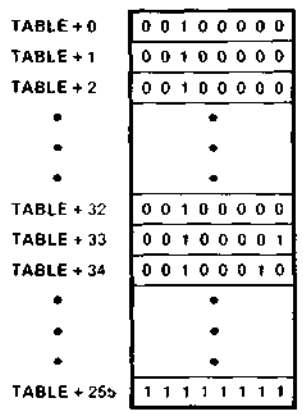
Byte
25

TRIB
Translate and Increment

NOTE: The original contents of register RH1 is lost, and is replaced by an undefined value.

EXAMPLE: This instruction is used in a "loop" of instructions which translate a string of data from one code to any other desired code, but an intermediate operation on each data element is required. The following sequence translates a string of 1000 bytes to the same string of bytes, with all ASCII "control characters" (values less than 32, see Appendix) translated to the "blank" character (value=32). A test, however, is made for the special character "return" (value=13) which terminates the loop. The translation table contains 256 bytes. The first 33 (0-32) entries all contain the value 32, and all other entries contain their own index in the table, counting from zero.

```
        LD      R3,#1000        !Initialize counter!
        LD      R1,#STRING      !Load start addresses!
        LD      R2,#TABLE
LOOP:
        CPB     @R1,#13         !Check for return character!
        JR      EQ,DONE         !Exit loop if found!
        TRIB    @R1,@R2,R3      !Translate next byte!
        JR      NOV,LOOP        !Repeat until counter = 0!
DONE:
```



TRIRB
Translate, Increment and Repeat

TRIRB dst,src,r

dst: IR
src: IR

INSTRUCTION FORMAT: (F6.4)

	dst	src	r
	IR	IR	R

OPERATION:
```
dst <- src[dst]
AUTOINCREMENT dst {+1}
r <- r - 1
repeat until r = 0
```

This instruction is used to translate a string of bytes from one code to another code. The contents of the location addressed by the destination register (the "target byte") is used as an index into a table of translation values whose lowest address is contained in the source register. The index is computed by adding the target byte to the address contained in the source register. The addition is performed following the rules for addressing mode arithmetic, with the target byte treated as an unsigned 8-bit value extended with high-order zeros. The sum is used as the address of an 8-bit translation value within the table which replaces the original contents of the location addressed by the destination register.

The destination register is then incremented by one, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can translate from 1 to 65536 bytes.

Because the 8-bit target byte is added to the source register to obtain the address of a translation value, the table may contain 256 bytes. A smaller table size may be used where it is known that not all possible 8-bit target byte values will occur.

FLAGS:
C: Unaffected
Z: Undefined
S: Unaffected
V: Set
D: Unaffected
H: Unaffected

CYCLES:

Byte
11+14*n (n=number of data elements translated)

TRIRB
Translate, Increment and Repeat

NOTES: The original contents of register RH1 is lost, and is replaced by an undefined value.

This instruction can be interrupted after each execution of the basic operation. The program counter of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

EXAMPLE: The following sequence of instructions can be used to translate a string of 80 bytes from one code to another. The pointers to the string and the translation table are set, the number of bytes to translate is set, and then the translation is accomplished.

```
LD      R1,#STRING
LD      R2,#TABLE
LD      R3,#80
TRIRB   @R1,@R2,R3
```

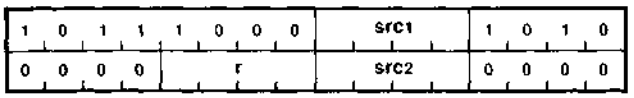
TRTDB
Translate, Test and Decrement

TRTDB srcl,src2,r

srcl: IR
src2: IR

INSTRUCTION FORMAT: (F6.4)

	srcl	src2	r
	IR	IR	R



OPERATION:
RH1 <- src2[srcl]
AUTODECREMENT srcl {-1}
r <- r - 1

This instruction is used to scan a string of bytes testing for bytes with special meaning. The contents of the location addressed by the first source register (the "target byte") is used as an index into a table of translation values whose lowest address is contained in the second source register. The index is computed by adding the target byte to the address contained in the second source register. The addition is performed following the rules for addressing mode arithmetic, with the target byte treated as an unsigned 8-bit value extended with high-order zeros. The sum is used as the address of an 8-bit value within the table which is loaded into register RH1. The Z flag is set if the value loaded into RH1 is zero; otherwise the Z flag is cleared. The contents of the locations addressed by the source registers are not affected.

The first source register is then decremented by one, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

Target byte values which have corresponding zero translation table entry values are to be scanned over, while target byte values which have corresponding non-zero translation table entry values are to be detected. Because the 8-bit target byte is added to the second source register to obtain the address of a translation value, the table may contain 256 bytes. A smaller table size may be used where it is known that not all possible 8-bit target byte values will occur.

TRTDB
Translate, Test and Decrement

FLAGS:
- C: Unaffected
- Z: Set if the translation value loaded into RH1 is zero; cleared otherwise
- S: Unaffected
- V: Set if the result of decrementing r is zero; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Byte
25

EXAMPLE: If register R6 contains %4001, the byte at location %4001 contains 3, register R9 contains %1000, the byte at location %1003 contains %AA, and register R12 contains 2, the statement

```
TRTDB  @R6,@R9,R12
```

will leave the value %AA in RH1, the value %4000 in R6, and the value 1 in R12. Location %4001 and register R9 will not be affected.

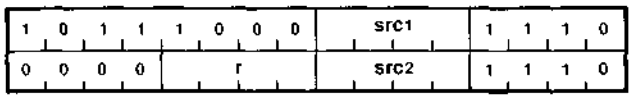
TRTDRB
Translate, Test, Decrement and Repeat

TRTDRB srcl,src2,r

srcl: IR
src2: IR

INSTRUCTION FORMAT: (F6.4)

srcl	src2	r
IR	IR	R



OPERATION:
```
RH1 <- src2[srcl]
AUTODECREMENT srcl {-1}
r <- r - 1
repeat until RH1 <> 0 or r = 0
```

This instruction is used to scan a string of bytes testing for bytes with special meaning. The contents of the location addressed by the first source register (the "target byte") is used as an index into a table of translation values whose lowest address is contained in the second source register. The index is computed by adding the target byte to the address contained in the second source register. The addition is performed following the rules for addressing mode arithmetic, with the target byte treated as an unsigned 8-bit value extended with high-order zeros. The sum is used as the address of an 8-bit value within the table which is loaded into register RH1. The Z flag is set if the value loaded into RH1 is zero; otherwise the Z flag is cleared. The contents of the locations addressed by the source registers are not affected.

The first source register is then decremented by one, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until either the Z flag is clear, indicating that a non-zero translation value was loaded into RH1, or until the result of decrementing r is zero. This instruction can translate and test from 1 to 65536 bytes.

Target byte values which have corresponding zero translation table entry values are to be scanned over, while target byte values which have corresponding non-zero translation table entry values are to be detected. Because the 8-bit target byte is added to the second source register to obtain the address of a translation value, the table may contain 256 bytes. A smaller table size may be used where it is known that not all possible 8-bit target byte values will occur.

TRTDRB
Translate, Test, Decrement and Repeat

FLAGS: C: Unaffected
Z: Set if the translation value loaded into RH1 is zero; cleared otherwise
S: Unaffected
V: Set if the result of decrementing r is zero; cleared otherwise
D: Unaffected
H: Unaffected

CYCLES:

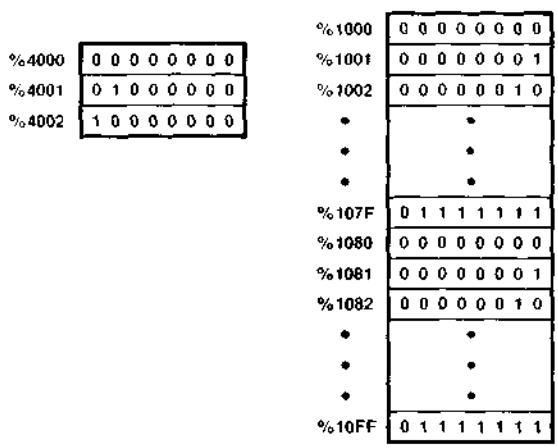
Byte
11+14*n (n=number of data elements tested)

NOTE: This instruction can be interrupted after each execution of the basic operation. The program counter of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

EXAMPLE: If register R6 contains %4002, the bytes at locations %4000 through %4002 contain the values %00, %40, %80, respectively, register R9 contains %1000, the translation table from location %1000 through %10FF contains 0,1,2,...,%7F,0,1,2,...,%7F (the second zero is located at %1080), and register R12 contains 3, the statement

```
TRTDRB  @R6,@R9,R12
```

will leave the value %40 in RH1 (which was loaded from location %1040). Register R6 will contain %4000, and R12 will contain 1. R9 will not be affected.



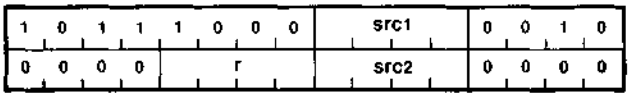
TRTIB
Translate, Test and Increment

TRTIB srcl,src2,r

srcl: IR
src2: IR

INSTRUCTION FORMAT: (F6.4)

srcl	src2	r
IR	IR	R



OPERATION:
RH1 <- src2[srcl]
AUTOINCREMENT srcl {+1}
r <- r - 1

This instruction is used to scan a string of bytes testing for bytes with special meaning. The contents of the location addressed by the first source register (the "target byte") is used as an index into a table of translation values whose lowest address is contained in the second source register. The index is computed by adding the target byte to the address contained in the second source register. The addition is performed following the rules for addressing mode arithmetic, with the target byte treated as an unsigned 8-bit value extended with high-order zeros. The sum is used as the address of an 8-bit value within the table which is loaded into register RH1. The Z flag is set if the value loaded into RH1 is zero; otherwise the Z flag is cleared. The contents of the locations addressed by the source registers are not affected.

The first source register is then incremented by one, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one.

Target byte values which have corresponding zero translation table entry values are to be scanned over, while target byte values which have corresponding non-zero translation table entry values are to be detected. Because the 8-bit target byte is added to the second source register to obtain the address of a translation value, the table may contain 256 bytes. A smaller table size may be used where it is known that not all possible 8-bit target byte values will occur.

TRTIB
Translate, Test and Increment

FLAGS:
- C: Unaffected
- Z: Set if the translation value loaded into RH1 is zero; cleared otherwise
- S: Unaffected
- V: Set if the result of decrementing r is zero; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Byte
25

EXAMPLE: This instruction is used in a "loop" of instructions which translate and test a string of data, but an intermediate operation on each data element is required. The following sequence outputs a string of 72 bytes, with each byte of the original string translated from its 7-bit ASCII code (see Appendix) to an 8-bit value with odd parity. A test, however, is made for the special character "return" (value=%D) which terminates the loop. The translation table contains 128 bytes, which assumes that the most significant bit of each byte in the string is always zero. Each entry contains its own index in the table in the low-order seven bits, with the most significant bit of the value set or cleared to give odd parity for the entire value.

```
     LD      R4,#72        !Initialize counter!
     LD      R5,#STRING    !Load start addresses!
     LD      R6,#TABLE
LOOP:
     CPB     @R5,#%D       !Check for return character!
     JR      EQ,DONE       !Exit loop if found!
     TRTIB   @R5,@R6,R4    !Translate next byte!
     OUTB    PORTN,RH1     !Output with odd parity!
     JR      NOV,LOOP      !Repeat until counter=0!
DONE:
```

TABLE+0	10000000
TABLE+1	00000001
TABLE+2	00000010
TABLE+3	10000011
TABLE+4	00000100
TABLE+5	10000101
TABLE+6	10000110
TABLE+7	00000111
TABLE+8	00001000
TABLE+9	10001001
•	•
•	•
•	•

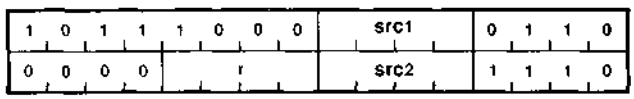
TRTIRB
Translate, Test, Increment and Repeat

TRTIRB srcl,src2,r

srcl: IR
src2: IR

INSTRUCTION FORMAT: (F6.4)

	srcl	src2	r
	IR	IR	R



OPERATION:
```
RH1 <- src2[srcl]
AUTOINCREMENT srcl {+1}
r <- r - 1
repeat until RH1 <> 0 or r = 0
```

This instruction is used to scan a string of bytes testing for bytes with special meaning. The contents of the location addressed by the first source register (the "target byte") is used as an index into a table of translation values whose lowest address is contained in the second source register. The index is computed by adding the target byte to the address contained in the second source register. The addition is performed following the rules for addressing mode arithmetic, with the target byte treated as an unsigned 8-bit value extended with high-order zeros. The sum is used as the address of an 8-bit value within the table which is loaded into register RH1. The Z flag is set if the value loaded into RH1 is zero; otherwise the Z flag is cleared. The contents of the locations addressed by the source registers are not affected.

The first source register is then incremented by one, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until either the Z flag is clear, indicating that a non-zero translation value was loaded into RH1, or until the result of decrementing r is zero. This instruction can translate and test from 1 to 65536 bytes.

Target byte values which have corresponding zero translation table entry values are to be scanned over, while target byte values which have corresponding non-zero translation table entry values are to be detected. Because the 8-bit target byte is added to the second source register to obtain the address of a translation value, the table may contain 256 bytes. A smaller table size may be used where it is known that not all possible 8-bit target byte values will occur.

TRTIRB
Translate, Test, Increment and Repeat

FLAGS:
- C: Unaffected
- Z: Set if the translation value loaded into RH1 is zero; cleared otherwise
- S: Unaffected
- V: Set if the result of decrementing r is zero; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

Byte
11+14*n (n=number of data elements tested)

NOTE: This instruction can be interrupted after each execution of the basic operation. The program counter of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

EXAMPLE: The following sequence of instructions can be used to scan a string of 80 bytes, testing for special characters as defined by corresponding non-zero translation table entry values. The pointers to the string and translation table are set, the number of bytes to scan is set, and then the translation and testing is accomplished. The Z and V flags can be tested after the operation to determine if a special character was found, and whether the end of the string has been reached. The translation value loaded into RH1 might then be used to index another table, or to select one of a set of sequences of instructions to execute next.

```
          LD        R4,#STRING
          LD        R5,#TABLE
          LD        R6,#80
          TRTIRB    @R4,@R5,R6
          JR        NZ,SPECIAL
END_OF_STRING:      .
                    .
                    .
SPECIAL:
          JR        OV,LAST_CHAR_SPECIAL
                    .
                    .
                    .
LAST_CHAR_SPECIAL:
```

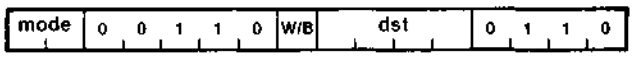
TSET
Test and Set

TSET dst
TSETB

dst: R, IR, DA, X

INSTRUCTION FORMAT: (F1.1)

W/B:



mode	dst
10	R
00	IR
01	DA (dst field=0)
01	X (dst field<>0)

OPERATION: S <- dst(msb)
dst(0:msb) <- 111...111

Tests the most significant bit of the destination operand, copying its value into the S flag, then sets the entire destination to all 1 bits. This instruction provides a locking mechanism which can be used to synchronize software "processes" which require exclusive access to certain data or instructions at one time.

FLAGS:
C: Unaffected
Z: Unaffected
S: Set if the most significant bit of the destination was 1; cleared otherwise
V: Unaffected
D: Unaffected
H: Unaffected

CYCLES:

dst	Word/Byte NS	SS	SL
R	7	--	--
IR	11	--	--
DA	14	15	17
X	15	15	18

NOTE: During the execution of this instruction, $\overline{\text{BUSRQ}}$ is not honored in the time between loading the destination from memory and storing the destination to memory. For systems with one processor, this ensures that the testing and setting of the destination will be completed without any intervening accesses. This instruction should not be used to synchronize software "processes" residing on separate processors where the destination is a shared memory location, since this locking mechanism cannot be guaranteed to function correctly with multi-processor accesses.

TSET
Test and Set

EXAMPLE: A simple mutually-exclusive "critical region" may be implemented by the following sequence of statements:

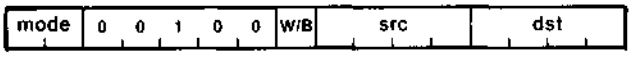
```
ENTER:
        TSET SEMAPHORE
        JR   MI,ENTER      !loop until resource
              .             controlled by SEMAPHORE
              .             is available!
              .
        !Critical Region--only one software process
         executes this code at a time!
              .
              .
              .
        CLR  SEMAPHORE     !release resource con-
                            trolled by SEMAPHORE!
```

XOR
Exclusive OR

```
XOR   dst,src                    dst: R
XORB                             src: R, IM, IR, DA, X
```

INSTRUCTION FORMAT: (F2.1)

W/B:



mode	dst	src
10	R	R
00	R	IM (src field=0)
00	R	IR (src field<>0)
01	R	DA (src field=0)
01	R	X (src field<>0)

OPERATION: dst <- dst XOR src

The source operand is logically EXCLUSIVE ORed with the destination operand and the result is stored in the destination. The contents of the source are not affected. The EXCLUSIVE OR operation results in a one bit being stored whenever the corresponding bits in the two operands are different; otherwise, a zero bit is stored.

FLAGS:
- C: Unaffected
- Z: Set if the result is zero; cleared otherwise
- S: Set if the most significant bit of the result is set; cleared otherwise
- P: XOR - unaffected; XORB - set if parity of the result is even; cleared otherwise
- D: Unaffected
- H: Unaffected

CYCLES:

src	Word/Byte		
	NS	SS	SL
R	4	--	--
IM	7	--	--
IR	7	--	--
DA	9	10	12
X	10	10	13

EXAMPLE: If register RL3 contains %C3 (11000011) and the source operand is the immediate value %7B (01111011), the statement

```
XORB  RL3,#%7B
```

will leave the value %B8 (10111000) in RL3.

3.4 UNIMPLEMENTED INSTRUCTIONS

As explained in Section 1.6.4, certain binary instruction codes are defined by the Z8000 architecture, but are not currently implemented by the hardware. The following opcodes (listed by the first instruction word in hexadecimal where "x" means any value) will cause an Unimplemented Instruction trap if an attempt is made to execute them:

0Exx
0Fxx
4Exx
4Fxx
8Exx
8Fxx

NOTE

These opcodes are reserved for future extensions to the Z8000 architecture and must not be used by the programmer. To provide for compatibility with Zilog hardware and software systems, the results of attempted execution of these instructions is undefined, other than the occurrence of an Unimplemented Instruction trap.

Section 4
Structuring a Z8000 Program

4.1 INTRODUCTION

This section introduces the high-level PLZ/ASM statements described in Section 5. The structuring of programs and the concepts of module linkage and relocation are discussed.

4.2 PROGRAM STRUCTURE

4.2.1 Modules

A Z8000 PLZ/ASM program consists of one or more separately-coded and assembled modules. These modules are combined into an executable program using the module linkage and relocation facilities of an operating system such as Z-80 RIO. One of the modules should include a "main program"; that is, a GLOBAL procedure whose name is supplied to the linking facility as the program's entry point.

PLZ/ASM modules are made up of high-level and assembly-language statements that either declare or define data or perform some action. The assembly-language statements described in Section 3 are action statements. In general, data definition equates an identifier with a fixed value or type, whereas data declaration equates an identifier with a variable and associates a type with it.

Data, labels, or procedures can be shared between modules by declaring them to be GLOBAL in one module and declaring them as EXTERNAL in other modules that reference them. Data, labels, or procedures declared as INTERNAL to a module may be referenced only within that module. See Section 4.2.5 for more explanation of the scope of these objects.

The following example is the skeleton of a module showing two procedures, the second of which is a main program.

```
bubble_sort MODULE

   CONSTANT
     false  := 0              !Constant definition!
     true   := 1

   EXTERNAL
     list ARRAY [10 WORD]     !External variable
                               declared in another module

   INTERNAL                   !First procedure!
     sort                     !Procedure name!
       PROCEDURE
            .
            .
            .
     END sort                 !End first procedure!

   GLOBAL                     !Second procedure!
     main
       PROCEDURE
       ENTRY
         LD  R0, #9*2
         CALL sort
         RET
     END main                 !End second procedure!

END bubble_sort               !End of module!
```

4.2.2 Procedures

A procedure declaration defines an executable portion of a module (including both action and data statements). It also associates an identifier with that block of code so that it can be activated by the assembly-language CALL statement. (See the "CALL sort" statement in the preceding example).

Every procedure declaration also has a scope associated with it. It can be:

- GLOBAL (the procedure can be called from other modules);
- INTERNAL (it can be called only within the current module);
- EXTERNAL (it is called from the current module, but is declared as GLOBAL in another module).

A procedure declaration can include LOCAL variables or label declarations as shown in the following example. LOCAL variables and labels can be referenced only within the procedure in which they are declared.

```
INTERNAL                    !Procedure scope!
  sort                      !Procedure name!
    PROCEDURE
      LOCAL
        switch BYTE         !Local variable!
    ENTRY
    DO                      !DO loop introduced!
     .
     .                      !(action statements)!
     .
    OD                      !End DO loop!
  END sort                  !End of procedure!
```

Note in this example that the keyword ENTRY is used to separate the declarations of LOCAL variables from the executable action statements making up the body of the procedure. The example also carries the previous "bubble-sort module" example to the first inner level of detail--the procedure. At the same time it introduces another program structuring element--the DO loop.

4.2.3 DO Loops

The description of the DJNZ instruction in Section 3 pointed out that it is used primarily for loop control. DO loops perform a similar function, but in a more structured, high-level manner. The statements between the keywords DO and OD are executed repeatedly until control is diverted through a loop control statement: REPEAT, EXIT, or one of the assembly-language branch statements. The EXIT statement causes execution to continue at the first statement following the innermost DO loop containing the EXIT, while REPEAT statement causes execution to continue at the first statement of the innermost DO loop containing the REPEAT.

A DO statement does not introduce a new scope. It contains only action statements.

The following example enlarges upon the original "bubblesort" skeleton and also introduces the IF statement.

```
DO
  LDB   switch, #false
  CLR   R1
  DO                          !Note nested DO loop!
    CP   R1, R0
    IF UGE THEN EXIT FI       !IF statement at inner level!
     .
     .
     .
    INC   R1, #2
  OD                          !End nested DO loop!
CPB   switch, #false
IF . . . FI                   !IF statement at outer level!
OD                            !End DO loop!
```

Note in this example that DO loops can contain IF statements or other DO loops. IF statements can also have DO loops or other IF statements nested within them.

4.2.4 IF Statements

The general form of the IF statement is:

```
IF condition-code  THEN  action1
          ELSE  action2
FI
```

The IF statement specifies that all statements between THEN and ELSE (action1) be executed if the condition specified is true. Otherwise, the statements between ELSE and FI (action2) are executed. The ELSE clause can be omitted, in which case no action is performed when the specified condition is false. Like the DO loop, the IF statement contains only executable statements.

The condition codes specified are the same as for the assembly language JP and JR instructions (section 3.2.1). As in the case of DO loops versus the DJNZ instruction, one can set up conditional statements using only assembly-language instructions, but the high-level IF statement may lead to a more structured program.

The following example shows a brief IF statement followed by a longer one.

```
IF  UGE  THEN  EXIT  FI
    .
    .
    .
IF UGT THEN
   LDB    switch, #true
   LD     list(R1), R6
   LD     list(R2), R4
FI
```

4.2.5 Scope

The scope of a variable, label, or procedure, refers to that portion of the program in which the object is recognized and handled in keeping with its declaration. When declaring an object for use solely within the current module or procedure, its use elsewhere is of no concern. Even though the same identifier is used in another scope, it refers to a different object.

In general, a scope can be an entire executable program, a module, or a procedure. An identifier is accessible in a scope S if it is either

- Declared in S or in the module containing S, or
- Declared EXTERNAL in the module containing S.

New identifiers are declared by their appearance in a variable, procedure, constant, or type declaration, or by their use as a label within the module. If a label identifier is not declared explicitly, it is assumed to have an INTERNAL scope and thus is accessible throughout the module in which it is defined. The scope of a label identifier may explicitly be declared to be either GLOBAL, INTERNAL, or LOCAL. In addition, a special form of statement labels is discussed in Section 5.3.5, which always have a LOCAL scope and therefore cannot be declared explicitly.

New identifiers introduced in a variable, procedure, or label declaration are accessible only within the newly-established scope as determined from the declaration class specified: GLOBAL, EXTERNAL, INTERNAL or LOCAL. A new identifier cannot be the same as an identifier already accessible in the current scope.

For variable and label declarations, GLOBAL specifies that the variable is declared in the current module but may be used in other modules. GLOBAL variables have a scope of the entire executable program. EXTERNAL specifies that the variable is used in the current module but declared as GLOBAL in another module. INTERNAL specifies that the variable is declared in the current module and is accessible only within the current module. LOCAL specifies that the variable can be accessed only inside the procedure in which it is declared. GLOBAL, EXTERNAL, and INTERNAL can only be specified at the module level; LOCAL can only be used at the procedure level.

For variables declared with type RECORD, the scope of a record field is the module or procedure in which it is defined. Note that this implies that record fields must have names that are unique within their scope.

Procedure declarations must be either GLOBAL, EXTERNAL, or INTERNAL, as defined for variables above. The name given to the procedure as part of the procedure declaration is recognized in the entire scope of the enclosing module.

Constant identifiers are defined using the CONSTANT class, and can be defined only at the module level. The scope of a constant identifier is the module scope, and it cannot be used outside the module unless it is redefined.

Type identifiers are defined using the TYPE class. They can be defined only at the module level and are recognized only within the scope of the module.

An identifier used in a scope and not declared in that scope is said to be free in that scope. Any identifier that is free in the scope of a procedure must be declared in the enclosing module. Procedures do not explicitly import variables into their scope as modules do (via the EXTERNAL declaration).

4.2.6 Summary

The preceding sections were intended only as an introduction to program structuring. The detailed formats of the declarations used to define the program structure are listed in Section 5. Even without those details, however, the following module should be comprehensible at this point. If any statement looks foreign, reread the part of Section 3 or 4 that explains its operation.

An example of a complete bubble_sort module for sorting a 10-word array is as follows:

```
bubble_sort MODULE                          !Module declaration!

  CONSTANT                                  !Constant declarations!
    false  := 0
    true   := 1

  EXTERNAL
    list ARRAY [10 WORD]

  INTERNAL                                  !Procedure declaration!
    sort
      PROCEDURE
        LOCAL                               !Local variable declaration!
          switch BYTE                       !Loop control switch!
        ENTRY                               !Begin executable part!
        DO
          LDB  switch, #false               !Initialize switch!
          CLR  R1                           !Clear array pointer i!
          DO
            CP  R1, R0                      !Done?!
            IF UGE THEN EXIT FI
            LD  R2, R1                      !Initialize pointer j!
            INC R2, #2                      !j = i+1 (double for words)!
            LD  R4, list(R1)
            LD  R6, list(R2)
            CP  R4, R6                      !If list[i] > list[j] then!
            IF UGT THEN                     !exchange to bubble largest!
              LDB switch, #true             !number to top of array!
              LD  list(R1), R6
              LD  list(R2), R4
            FI
            INC  R1, #2                     !Advance pointer (double for
                                             words)!
          OD                                !End nested DO loop!
          CPB switch, #false                !Test switch!
          IF EQ THEN RET FI
        OD                                  !End outer DO loop!
      END sort                              !End of procedure!

  GLOBAL                                    !New procedure declaration!
    main                                    !Program entry procedure!
      PROCEDURE
      ENTRY                                 !No LOCALs!
        LD  R0, #9*2                        !Initialize loop control!
                                            !Double for word array!
        CALL sort                           !Call sort procedure!
        RET
    END main                                !End of main procedure!

END bubble_sort                             !End of module!
```

4.3 RELOCATABILITY

The Z8000 PLZ/ASM assembler produces relocatable object modules. Essentially, this frees the programmer from memory management concerns during program development (since object code can be relocated in memory) and also allows programs to be developed in modules whose addresses are resolved automatically when the modules are linked.

Modular program development offers numerous advantages to the developer. Complex programs can be divided into several smaller tasks and assigned to a development team, should schedule constraints require quick completion. An entire program need not be delayed while one module is awaiting development, since modules can be separately tested and assembled. A change affecting a single module will not have a ripple effect through the entire program and will require reassembly of the affected module only. In short, several simple programs are generally easier to write, test, and debug than one large, complex program.

The relocatability feature of the Z8000 assembler is supported by the $ABS, $REL, and $SECTION assembler directives. These directives determine whether programs are assembled in relocatable mode or not, and where data and action statements are to be loaded into memory. An operating system program called a linker relocates object modules and resolves inter-module references.

These assembler directives are summarized below. Other directives are listed in Appendix C, and are described in detail in the Z8000 Assembler User's Guide.

4.3.1 Sections

In addition to the logical structuring provided by modules and procedures, it is possible to divide a program into sections which can be mapped into various areas of memory when the program is linked or loaded for execution. For example, the programmer may choose to group a set of data structures and the procedures which manipulate them together in the same module. But it may also be desirable to physically separate the object code for the procedures from the data in a system where read-only memory is used for the procedures and read/write memory is used for the data.

The assembler allows a program to be arbitrarily divided into named sections. Each section might be allocated to a different address space; for instance, code or data memory. In the segmented Z8000, each section might be mapped into a different segment, or several sections might be combined from different modules into the same segment. A single module may contain

several sections, each of which will be allocated a different area in memory. Alternatively, the portions of a single section may be spread through several modules and the portions will be automatically combined into a single area by the linker. Sections provide the programmer with complete control over the mapping of a program into the address spaces of the Z8000 processor.

Usually the full generality of arbitrary sections is not needed for a particular application. By default, the assembler automatically creates two sections for each module, one for all the data variables and the other for all procedures. The data section is given the same name as the module name, with the suffix "_D" (or "_d") appended. The procedure section is given the same name as the module name, with the suffix "_P" (or "_p") appended. If the first character of the module name is uppercase, then an uppercase suffix is used; otherwise, a lowercase suffix is used.

For example, the data section "bubble_sort_d" will be created for the sample module in Section 4.2.6, and contains only the BYTE named "switch". The program section "bubble_sort_p" contains all the code in the module.

The $SECTION assembler directive can be used to override the default sectioning. This directive causes all following statements to be associated with a symbolic identifier which can later be mapped into a particular memory area. It has the format:

```
$SECTION  identifier
```

The $SECTION directive remains in effect until either another $SECTION directive is encountered, the $SDEFAULT directive is encountered which returns to the default section scheme described above, or the end of the module is found.

NOTE

Because the assembler automatically calculates the displacement used in the Relative addressing mode (Section 2.4.6), the target address must be in the same section and module as the instruction which uses Relative Addressing.

4.3.2 Location Counter Control

The assembler keeps track of the "location" of the current statement with a location counter, just as an executing program does with its program counter. There is a location counter associated with each section in a program. The counter value

represents a 16-bit offset within the current section. The offset may be either an absolute value, or it may represent a relocatable value which may be adjusted depending on where the module's portion of the section is finally allocated at link or load time. Relocatable and absolute portions of a section may be specified in the same module or program.

If the $ABS assembler directive has been specified, the location counter reflects the absolute location of the current statement; if $REL has been specified, the counter reflects the relocatable offset of the statement. If neither is specified, the counter defaults to relocatable offset 0 at the beginning of a module. The location counter symbol "$" may be used in any expression, and represents the address of the first byte of the current instruction.

For the majority of programming tasks, one need not specify either $ABS or $REL in the source program. Program location can be carried out more appropriately at link time or load time. If one desires complete control over program location at assembly time, however, the $ABS directive will force that part of the program to reside at a specific location.

4.3.3 Modes of Arithmetic Expressions

All arithmetic expressions have one of three modes associated with them: absolute, relocatable, or external.

An absolute expression consists of one or more constants, constant identifiers, or absolute labels combined with arithmetic or logical operators. The difference between two relocatable expressions is also considered to be absolute.

```
JP    IRQ_VECTOR          !Where IRQ_VECTOR is an absolute
                           label!
ADD   R1, #K*3            !Where K is a constant identifier!
```

A relocatable expression is one containing exactly one identifier subject to relocation after assembly. The expression can be extended by adding or subtracting an absolute expression. Plus and minus are the only operators allowed, however.

```
JP    Z, LOOP+2           !Where LOOP is a relocatable
                           label!
```

An external expression is one containing exactly one external identifier, possibly extended by adding or subtracting an absolute expression. An external identifier is one that is used in the current module but defined in another module (Section 4.2.5). The value of an external identifier is not known until the modules are linked.

```
LD    R0, timer_count        !External if timer_count was
                              defined outside current module!
```

In the following summary, "AB" stands for an absolute value or expression, "RE" stands for a relocatable value or expression, and "EX" is an external identifier or expression. "operator" is one of the standard arithmetic or logical operators (+, -, *, LOR, LAND, etc.).

"AB" is defined as either:

- an absolute identifier
- AB "operator" AB
- +AB, -AB, or LNOT AB
- RE - RE

"RE" is defined as either:

- a relocatable identifier
- RE + AB
- AB + RE
- RE - AB
- +RE

"EX" is defined as either:

- an external identifier
- EX + AB
- AB + EX
- EX - AB
- +EX

Certain mode combinations are not permitted in PLZ/ASM. For example, a relocatable expression that does not result in simple relocation is invalid. Simple relocation means that only a single relocation factor need be added to a relocatable value when the assembled program is relocated.

```
JP    Z, LOOP1 + 8           !VALID -- relocation factor is
                              added only once!

JP    Z, LOOP1 + LOOP2       !INVALID -- relocation factor
                              would have to be added twice!
```

In general, the second example would be invalid no matter which arithmetic or logical operator was used to combine LOOP1 and LOOP2. The one exception is subtraction. Suppose LOOP2 is the label of the first statement following a procedure named LOOP1. The statement

```
LD    R5, #(LOOP2 - LOOP1)
```

could be used to find the length of the LOOP1 procedure. The difference between two relocatable labels (which must be in the same section) is always absolute, regardless of where the module is relocated to.

Other invalid mode combinations are:

- A relocatable expression multiplied or divided by an absolute expression:

```
JP    Z, LOOP1*4    !INVALID!
```

- A relocatable expression subtracted from an absolute expression (although the reverse is allowed):

```
DJNZ  R5, LOOP1 - 8          !VALID!
DJNZ  R5, 8 - LOOP1          !INVALID!
```

- An external expression combined with a relocatable expression, and vice-versa:

```
ADD   R6, EXTERNAL_NUMB + LOOP1      !INVALID!
JP    Z, LOOP1 - EXTERNAL_NUMB       !INVALID!
```

Section 5
PLZ/ASM High-Level Statements

5.1 Z8000 SOURCE PROGRAM STATEMENTS

The majority of code in a Z8000 PLZ/ASM program will normally be the assembly-language instructions described in Section 3. Typically, the source program will also include some high-level statements and assembler directives.

High-level statements perform two basic functions:

- They introduce program structures (modules, procedures, DO loops, and IF statements);
- They declare and define data.

Assembler directives control the mode of assembly (absolute or relocatable), determine where object code is to be stored in memory, and specify the form of assembler output. These directives are embedded in the source program and are always preceded by a dollar sign ($) (see Appendix C).

In the descriptions of high-level statements in this section the following notational conventions are used:

- Keywords are shown as all capital letters: MODULE
- Parameters shown in lowercase letters represent items to be replaced by actual data or names: module_identifier
- Optional items are enclosed in square brackets: [local_declaration]
- Possible repetition of an item is indicated by appending a "+" (to signify one or more repetitions) or an "*" (to signify zero or more repetitions) to the item: declaration*
- Other special characters shown in statement and command formats such as :=, (), and [] will be enclosed in single quotes and must be written as shown. The special symbol ":=" means "is defined as" or "is assigned".

For example:

```
CONSTANT
  constant_identifier ':=' constant_expression

RECORD  '[' identifier+ type ']'
```

5.2 PROGRAM STRUCTURING STATEMENTS

5.2.1 Module Declaration

A Z8000 PLZ/ASM program module consists of a sequence of data and procedure declarations. These declarations are bounded by the module declaration statement and the end-of-module statement. The format of a module declaration is:

```
module_identifier MODULE
  declaration*
END module_identifier
```

where

module_identifier	conforms to the rules for identifiers (Section 2.2.1).
declaration	is zero or more data and/or procedure declarations (Sections 5.2.2 and 5.3).

Example:

```
sine_computation MODULE
  .
  .
  .
END sine_computation
```

5.2.2 Procedure Declaration

A procedure declaration defines an executable part of a program and associates an identifier with it so that it can be activated by the assembly-language CALL statement.

The procedure heading specifies the identifier naming the procedure. This identifier labels the first instruction in the procedure, and can be used as any other program label. The scope of the procedure identifier may be either GLOBAL, INTERNAL, or EXTERNAL. If the procedure is declared EXTERNAL, then only the procedure identifier is given, since the actual definition of the procedure occurs in some other module. A procedure declaration may also include local variable declarations. These variables

are recognized only within the procedure in which they are declared.

The format of the procedure declaration is:

```
procedure_identifier PROCEDURE
  [LOCAL
    [variable_identifier+ type]*]*
  [ENTRY
    action_statement*]
END procedure_identifier
```

where

identifier	conforms to the rules for identifiers (Section 2.2.1).
type	is BYTE, SHORT_INTEGER, INTEGER, WORD, LONG, LONG_INTEGER, LABEL, ARRAY, RECORD or a user_defined type (Sections 5.3.2 through 5.3.4).
action_statement	is zero or more assembly-language, DO, IF, REPEAT, or EXIT statements, or a combination of these statements.

The keyword ENTRY is used to separate the local variable declarations from the executable part of the procedure, and must be used whenever any action statements are specified.

Example:

```
EXTERNAL
  print PROCEDURE

GLOBAL
  suml BYTE

  add_routine PROCEDURE
    LOCAL
      sum  BYTE
    ENTRY
      LDB   RH2,suml
      ADDB  RH2,#72
      LDB   sum,RH2
      CALL print
      RET
  END add_routine
```

Note that the RET instruction precedes the END statement. If RET is not present, control will fall through to the statement following the END statement.

5.2.3 DO Statement

DO loops provide a framework for performing actions repetitively. The statements between the DO and OD keywords are executed repeatedly until control is diverted through a loop control statement.

The only way the execution flow of a DO loop can be diverted is by encountering an assembly-language branch instruction (DJNZ, JP, JR, CALL, CALR, RET, or IRET) or an EXIT or REPEAT statement (described below).

The format of the DO loop is:

```
[label]*
DO
  action_statement*
OD
```

where

label	conforms to the rules for labels (Sections 2.2.1 and 5.3.5) and is used to identify the DO block for use with multi-level EXIT and REPEAT statements.
action_statement	is zero or more assembly-language, DO, IF, REPEAT, or EXIT statements, or a combination of these statements.

The assembler automatically inserts a single unconditional jump instruction at the OD keyword which branches back to the DO keyword. Either a JR or JP is generated depending on the range of the loop; JR is used whenever possible.

The EXIT statement causes execution to continue at the first statement following the innermost DO...OD block containing the EXIT. The EXIT statement may be further qualified by a label indicating a specific DO...OD block from which to exit. Its format is:

```
EXIT [FROM label]
```

where

label	conforms to the rules for labels (Sections 2.2.1 and 5.3.5).

The assembler automatically inserts a single unconditional jump to the instruction following the indicated OD keyword. Either a JR or JP is generated depending on the range of the EXIT, with a JR used whenever possible.

The REPEAT statement causes execution to continue at the first statement of the innermost DO...OD block containing the REPEAT. It can also be qualified by a label indicating a specific DO...OD block to which execution is to proceed. Its format is:

```
REPEAT [FROM label]
```

where

label	conforms to the rules for labels (Sections 2.2.1 and 5.3.5).

The assembler automatically inserts a single unconditional jump instruction at the REPEAT statement which branches to the indicated DO keyword. Either a JR or JP is generated depending on the range of the REPEAT, with a JR used whenever possible.

Example:

```
LOOP1:  DO
          ADD R0, @R1
          INC R5, #2
          CP  R5, #limit1
          IF EQ THEN EXIT FI
          DO
            ADD R2, @R3
            INC R6, #2
            CP  R6, #limit2
            IF GT THEN REPEAT FROM LOOP1 FI
          OD
        OD
```

5.2.4 IF Statement

The IF statement specifies that the statements between the keywords THEN and ELSE (or between THEN and FI if the ELSE clause is omitted) are to be executed if the specified condition code is true. If the condition is false and the ELSE clause is present, the statements between ELSE and FI are executed. If the condition is false and the ELSE clause is omitted, execution continues with the statement following FI.

The format of the IF statement is:

```
IF   condition_code
  THEN  action1_statement*
 [ELSE  action2_statement*]
FI
```

where

condition_code	is Z, NZ, C, NC, PL, MI, EQ, NE, OV, NOV, PE, PO, LE, LT, GT, GE, ULE, ULT, UGT, or UGE (Section 3.2.1).
action1_statement	is performed if "condition_code" is true and consists of zero or more assembly-language, DO, IF, REPEAT, or EXIT statements or a combination of these statements.
action2_statement	is performed if "condition_code" is false and consists of zero or more assembly-language, DO, IF, REPEAT, or EXIT statements or a combination of these statements.

The assembler automatically inserts a single conditional jump instruction just before the THEN keyword, which branches to either the ELSE clause, if present, or to the FI keyword if not. The conditional jump has an opposite sense from the condition code given; for instance, "IF OV THEN" generates a "JP NOV" instruction. The opposing condition pairs are: Z-NZ, C-NC, PL-MI, EQ-NE, OV-NOV, PE-PO, LE-GT, LT-GE, ULE-UGT, and ULT-UGE. If the ELSE clause is present, a single unconditional jump is inserted just before the ELSE clause which branches to the FI keyword.

For each of the jump instructions, either a JR or JP is generated depending on the range of the IF statement, with a JR used whenever possible.

Example:

```
IF NZ THEN
  LD  counter, #1
FI

IF GT
  THEN SETFLG C; RET
  ELSE RESFLG C; RET
FI
```

5.2.5 IF-CASE Statement

The IF-CASE statement is an extension of the IF statement. It allows the user to select from a series of actions depending on the contents of a selector register. The case whose list contains a match with the contents of the selector register is performed. An ELSE clause can be used to specify alternative statements to be executed if no match occurs. If no ELSE is specified and no match occurs, the statement following the FI keyword is executed next.

The IF-CASE statement has the format:

```
IF   selector_register
        [CASE  expression+  THEN action_statement*]+
        [ELSE  action_statement*]
FI
```

where

selector_register	is the designator for a single byte, word, or long word register (Section 2.4.2).
expression	is any expression which is valid as an operand in a Compare instruction.
action_statement	is zero or more assembly-language, DO, IF, REPEAT, or EXIT statements or a combination of these statements.

The assembler automatically inserts a Compare instruction (CP, CPB, or CPL) and a conditional jump for each list element. For the last expression in a CASE clause, a "jump NE" to the next CASE is generated (or to the ELSE clause, if present, or if not present, to the FI keyword for the last CASE). Either a JR or JP instruction is used depending on the range of the CASE clause, with a JR used whenever possible. When there is more than one expression in a single CASE clause, all but the last expression generate a "JR EQ" to the start of the action statements associated with the CASE clause. Therefore, the number and size of expressions for a single CASE clause must not exceed the range of the first JR instruction for that clause.

Example:

```
IF R5
  CASE #1, R4 THEN CALL control_1
  CASE #2, @R3 THEN CALL control_gt_1
  ELSE CALL control_gt_4
FI
```

5.3 DEFINING DATA

Data (constants and variables) must be defined or declared so that it can be referenced accordingly. In general, data definition associates an identifier with a fixed value or type. Data declaration introduces an identifier as the name of a variable and associates a scope and type with it. The following three statements are used to define and declare data:

- The constant-definition statement (CONSTANT), which associates a constant identifier with a fixed value;
- The type-definition statement (TYPE), which associates a type identifier with a fixed type;
- The variable declaration which associates a variable identifier with a scope, type, and (optionally) an initial value.

Constant identifiers are assumed to have INTERNAL scope. Constants have no explicit type and are represented as 32-bit values.

Type identifiers appearing in type-definitions are also assumed to have INTERNAL scope. No scope or initial value can be specified in type-definition statements. They are used primarily to categorize data or provide a template for structured data.

The variable declaration allows a variable identifier to be associated with any specific scope, type, or initial value (within the limits of the variable's scope).

5.3.1 Constant Definition

A constant definition associates an identifier with a constant expression. Since this value must be determinable at assembly time, any identifier appearing in the expression must be previously defined.

Constant identifiers have no explicit type and are always represented as 32-bit values. And, since constants are defined at the module level, they have a scope of the module in which they are defined (INTERNAL). Constants to be used in other modules must be redefined in those modules.

The format of the constant-definition statement is:

```
CONSTANT
  [constant_identifier ':=' constant_expression]*
```

where

constant_identifier	conforms to the rules for identifiers (Section 2.2.1).
constant_expression	conforms to the rules for constant expressions (Section 2.3.2).

Example:

```
CONSTANT
  minus        :=  -1
  count        :=  10
  NEG_COUNT    :=  minus*count
  A_SYMB       :=  'A'
  MODULUS      :=  256
```

5.3.2 Data Types

Data types are associated with variables either to indicate the size of the values the variables can represent or to identify a name as a label. Simple data types indicate whether a variable can hold an 8-bit, 16-bit or 32-bit value; structured data types provide a template of storage for collections of simple variables. The label type is used to declare the scope of labels explicitly.

Data types can be directly associated with variable identifiers in variable declarations (Section 5.3.4), or they may be associated with variables indirectly using a user-defined "type identifier" to specify the type. The latter is an identifier that has been previously associated with a type in a TYPE statement (Section 5.3.3). In the following example, the variable CHAR is associated with the simple type BYTE, allowing CHAR to be used as a type identifier in subsequent type definitions:

```
TYPE
  CHAR BYTE
  letter CHAR
  digit CHAR
```

Simple Types. The basic data type is either a standard simple data type or a simple data type defined by the user in a type definition (like CHAR in the example just given). The standard data types are:

SHORT_INTEGER or BYTE	An 8-bit quantity whose value can be signed (-128 to +127) or unsigned (0 to 255). This value may also represent a single character from the ASCII character set.
INTEGER or WORD	A 16-bit quantity whose value can be signed (-32768 to +32767) or unsigned (0 to 65535).
LONG_INTEGER or LONG	A 32-bit quantity whose value can be signed (-2,148,483,648 to +2,148,483,647) or unsigned (0 to 4,296,967,296).

The values of simple variables (that is, variables defined with simple types) are interpreted as signed or unsigned depending on their use in assembly-language instructions.

```
TYPE
  CHAR  BYTE
  small_value  BYTE
  large_value  LONG_INTEGER
  letter  CHAR
```

Structured Types. Structured types are defined by indicating the structuring method to be used and the types of all elements within the selected structure. Two structuring methods are available: ARRAY and RECORD.

Array Structures. An array structure is a collection of variable elements, each of which has the same type. When referenced, the identifier associated with the ARRAY type refers to the entire array structure. Arrays with N elements are indexed from 0 to N-1; for example, a 10-element array has index 0 as the first element and index 9 as the last element.

Individual elements within an array can be accessed in several different ways. A particular element's address may be calculated at run-time, for instance, by specifying an indexed address mode. At assembly-time, a particular element's address can be specified by an expression containing the array identifier and a fixed offset or by an array identifier followed by one or more constant expressions enclosed within square brackets. In the latter case, each constant expression represents an index for the particular "dimension" of the array, and the assembler's calculation of the desired element's address may involve an implicit multiplication by the size of each dimension or by the size (in bytes) of the element type (see Section 5.3.4).

Example:

```
TYPE
   STRING ARRAY [26 BYTE]
INTERNAL
   alpha STRING
!The array identifier "alpha" is defined as a 26-byte
 array!
         .
         .
         .
LD    R5, #0
LDB   alpha(R5), #'A'
LDB   alpha+1, #'B'
LDB   alpha[2], #'C'
!The first element of array alpha now contains 'A', the
 second element contains 'B', and the third element
 contains 'C'!
```

Array definition and initialization are explained in detail in Section 5.3.4.

Record Structures. A record structure is a collection of named fields. Unlike array elements, record fields are not required to have the same type. For example, a record structure named "strobe" might have a BYTE field named "pin" and a SHORT_INTEGER field named "voltage".

```
TYPE
    strobe RECORD [pin BYTE  voltage SHORT_INTEGER]
INTERNAL
    s1  strobe
```

As this example indicates, a RECORD type definition specifies an identifier and a type for each of its fields, as well as an identifier for the record structure itself. Individual fields can be referenced subsequently by specifying a record variable name followed by a period '.' and the field name.

```
LDB  RL5, s1.voltage
```

The scope of a record identifier is specified in the variable declaration in which it is introduced (or it is implicitly INTERNAL if introduced in a TYPE statement). The scope of record field identifiers is the module or procedure in which they are introduced; in other words, all field identifiers must be unique in their entire scope. Record definition is explained in detail in Section 5.3.4.

Label Type. An identifier with type LABEL can only be used as a statement label. The LABEL type declaration is used primarily to explicitly specify a label's scope. The scope of a label is assumed to be the module in which it appears (that is, INTERNAL).

Therefore, statement labels which are to be accessible throughout the module need not be declared, although they can be explicitly declared INTERNAL for documentation purposes. If a label is to have GLOBAL, EXTERNAL, or LOCAL scope, however, it must be declared explicitly in a label declaration statement.

```
GLOBAL
   TRIG_FUNCTION LABEL
```

Label declaration is explained in detail in Section 5.3.5.

5.3.3 Type Definition

The type-definition statement associates an identifier with a fixed type. These identifiers are assumed to have a scope of the current module (that is, INTERNAL scope).

Types are used primarily to categorize data and informally associate attributes or properties with the value of the data. Types ARRAY and RECORD provide an abbreviated template for structured data storage. Z8000 PLZ/ASM allows arbitrary association of types and data values with no type-compatibility restrictions.

The format of the type definition statement is:

```
TYPE
  [type_identifier type]*
```

where

type_identifier	conforms to the rules for identifiers (Section 2.2.1).
type	is BYTE, SHORT_INTEGER, INTEGER, WORD, LONG, LONG_INTEGER, a previously-defined type identifier, or is an ARRAY or RECORD type definition.

Example:

```
TYPE
  limit1  BYTE
  limit2  LONG

  LIST    ARRAY  [128 BYTE]
  matrix  ARRAY  [10 10 SHORT_INTEGER]
  patient RECORD [age height weight BYTE
                  room WORD
                  sex BYTE]

  char        BYTE
  letter      char
  digit       char
```

5.3.4 Variable Declaration

The variable declaration statement is used to declare the type of variable identifiers and optionally to define their initial values. Its basic format is:

```
variable_identifier+ type [':=' initial_value]
```

This format is limited, however, by the scope and type specified.

Initialization. There are several rules and special symbols used for initializations in general. When a single declaration contains more than one variable identifier, the corresponding initialization values are listed within square brackets. A simple variable is initialized using a constant expression. A structured variable is initialized using a "constructor", which is simply a list of values enclosed by square brackets, with each level of nesting within the structure denoted by a matching set of brackets. A special symbol, '...', indicates that the immediately preceding value or constructor is to be repeated for the rest of the variables at the current level of nesting. The special symbol '?' can be used as a placeholder in a list of initial values for simple variables or components of simple type within a structure, and indicates that the corresponding simple variable remains unassigned. An empty constructor, '[]', indicates that the corresponding structured variable remains unassigned.

Scope. The scope of the variable declaration can be GLOBAL, EXTERNAL, INTERNAL, or LOCAL (Section 4.2.5). The first three can be specified only in variable declarations at the module level. LOCAL can be specified only at the procedure level.

An EXTERNAL variable declaration cannot include an initial value. The type of the variable can be any simple or structured type, or a previously-specified type identifier. Consequently, the format of the variable declaration in this case reduces to:

```
variable_identifier+ type
```

Example:

```
EXTERNAL
  counter WORD
  input, output ARRAY [72 BYTE]
  customer_name STRING
```

Simple Variable Declaration. Simple variables are variables whose type is BYTE, SHORT_INTEGER, WORD, INTEGER, LONG, LONG_INTEGER, or a previously specified simple type identifier (Section 5.3.2). Simple variables which are GLOBAL, INTERNAL, or LOCAL may optionally be given an initial value.

Simple variables are initialized in one of two ways: with a single constant expression, or with a list of constant expressions enclosed in square brackets. In the first case, only one variable may appear in the declaration and is initialized to the constant value. If a list is supplied, the variables are initialized in left-to-right order from the initial-value list. The initial-value list may have fewer items than the variable-identifier list, but an error results if the initial-value list is longer.

Example:

```
INTERNAL
  HUE  BYTE
  limit WORD := %FFFF
  total, subtotal BYTE := [0...]
  A, B, X  BYTE  := ['A', 'B', 'X']
  D, E, F  BYTE  := [0, 1]
!D=0, E=1, F is still undefined!
```

Array Variable Declaration. An array variable is one whose type is ARRAY. An array variable declaration has the format:

```
array_identifier+  ARRAY '[' dimension+ type ']'
        [':=' initial_value]
```

or:

```
array_identifier+ array_type [':=' initial_value]
```

where

array_identifier	conforms to the rules for identifiers (Section 2.2.1).
dimension	specifies the number of dimensions in the array structure and the number of elements in each dimension. The dimension(s) must be one or more constant expression(s) (Section 2.3.2) or a single asterisk (*), as detailed below.
type	is BYTE, SHORT_INTEGER, INTEGER, WORD, LONG, LONG_INTEGER, a previously defined type identifier, or is an ARRAY, or RECORD type definition.
array_type	is a previously-defined ARRAY type identifier.

initial_value	is a bracketed list of constant expressions, or a character sequence as detailed below.

Several array identifiers may appear in a single declaration, and may optionally be initialized if they are declared GLOBAL, INTERNAL, or LOCAL.

Array structures are initialized in left-to-right order from the initial values supplied and in row-major sequence (that is, in the sequence of ascending memory addresses). For example, a 3x3 matrix would be ordered in the following sequence:

```
[0,0] [0,1] [0,2] [1,0] [1,1] [1,2] [2,0] [2,1] [2,2]
```

Array structures are initialized by a bracketed list of constant expressions. If the initial-value list contains "n" items, the first "n" elements of the array structure are initialized (in row-major sequence). The number of constants supplied cannot exceed the number of elements in the array structure.

Example:

```
INTERNAL
  matrix ARRAY [10 10 LONG_INTEGER]
  list   ARRAY [10 BYTE]  := [0,1,0,0,1]
  TABLE  ARRAY [4 BYTE]   := ['T', 'O', 'D', 'S']
  ONEDIM1, ONEDIM2  ARRAY [2 BYTE] := [[1...][2...]]
```

Normally, each dimension specified in the array declaration must be a constant expression so that variable upper bounds are prohibited. The sole exception is an array declaration initializing a one-dimensional array structure. In this case, '*' is specified as the dimension and the length of the list is determined by the number of items in the initialization list. When the '*' feature is used, array structures can be initialized in two ways: with a bracketed list of constants as described above, or with a character sequence (enclosed in single quote marks). In the latter instance, the array elements must be type BYTE, SHORT_INTEGER, or a user-defined 8-bit type and each byte is initialized to a single character value.

Example:

```
INTERNAL
  list ARRAY [* BYTE]  := [0,1,0,0,1]
!This array is only 5 bytes;  "list" array in last
 example is 10 bytes, although only 5 are initialized!

  TABLE ARRAY [* BYTE]  := 'TODS'
!Compare to TABLE array in last example!
```

Record Variable Declaration. A record variable is a variable whose type is RECORD. A record variable declaration specifies an identifier for the record as a whole and an identifier and type for each field within the record. A field type may be ARRAY or RECORD, as well as a simple type.

A record variable declaration has the format:

```
record_identifier+ RECORD '[' [field_identifier+ type]+ ']'
        [':=' initial_value]
```

or:

```
record_identifier+ record_type
        [':=' initial_value]
```

where

identifier	conforms to the rules for identifiers (Section 2.2.1).
type	is BYTE, SHORT_INTEGER, WORD, INTEGER, LONG, LONG_INTEGER, a previously-defined type identifier, or is an ARRAY or RECORD type definition.
record_type	is a previously-defined RECORD type identifier.
initial_value	is a bracketed list of constant expressions.

Several record identifiers may appear in a single declaration, and may optionally be initialized if declared GLOBAL, INTERNAL, or LOCAL. Each record field is initialized in left-to-right order from the values given in the initial list. A list of "n" values enclosed by square brackets may be given, with the first "n" fields being initialized. Having more constants than the total number of record fields is flagged as an error.

Example:

```
GLOBAL
  person  RECORD   [age, height, weight  BYTE
                    birth RECORD [day, month, year BYTE]
                    salary WORD]

  MSG  RECORD      [length BYTE char ARRAY [50 BYTE]] :=[0[0]]
                    !length field and first byte of char array
                    are initialized to zero!
```

If an array or record appears within another array or record, then this nesting is represented by enclosing each level of initialization values within square brackets. Notice that in this case, the last record fields or array elements at each level

do not have to be specified. Furthermore, if more than one structured variable identifier appears in a single declaration, then the part of the initial value list corresponding to each structured variable must also be enclosed by square brackets.

```
TYPE
    PATIENT RECORD [ ROOM WORD
                     BIRTH RECORD [DAY,MO,YR BYTE]
                     SEX  BYTE]
INTERNAL
    FEMALE ARRAY [100 PATIENT] :=[[?,[],'F']...]
       !only the SEX field of each record is initialized!
```

Alignment. If the current location counter value is even, then any data variable or instruction may start at that address. However, if the current location counter value is odd, then only 8-bit values may start at that address. The assembler automatically aligns simple variables so that word (WORD or INTEGER) or long word (LONG or LONG_INTEGER) variables are located at even addresses. This is accomplished by inserting a byte with an undefined value when the location counter value is odd, thus forcing the word or long word to start at the next higher even address. Instructions within a procedure are also automatically aligned.

Structured variables are, in general, automatically aligned; that is, all array or record variables are forced to start at an even address (an undefined byte value is inserted if the location counter was odd). An exception to this rule is made for arrays whose elements are any 8-bit type (based on BYTE or SHORT_INTEGER), which do not have to be aligned on an even address boundary.

It is the programmer's responsibility to ensure that all variables within a structure are aligned properly. The programmer should be careful to align record fields on their appropriate address boundaries relative to the start of the record. This can be accomplished by defining "filler" fields where necessary to force the alignment of the following field.

5.3.5 Label Declaration

The label declaration statement specifies that an identifier is used in the program as a statement label. It cannot be used for any other purpose within its defined scope. The format of the label declaration is:

```
label_declaration+ LABEL
```

where

label_identifier	conforms to the rules for identifiers in Section 2.2.1.

Note that the colon (:) that follows a label identifier when it appears in an executable statement (Section 2.2.1) is not included when the label is identified in a label declaration statement. Note also that labels cannot be given an initial value (that is, the label declaration cannot be used to assign absolute addresses).

A label can have GLOBAL, EXTERNAL, INTERNAL, or LOCAL scope (Section 4.2.5). If a label is used in an executable statement without being declared in a label declaration statement, it is assumed to be INTERNAL to the module in which it appears. This default scope can be overridden by explicitly declaring the scope of the label.

Notice that a label with LOCAL scope must be declared in the procedure declaration before the ENTRY keyword; that is, before the label is used or defined, whereas a label with GLOBAL, EXTERNAL, or INTERNAL scope may be used before it is either declared or defined. To allow the programmer to avoid pre-declaring LOCAL labels, Z8000 PLZ/ASM provides a special form of statement labels which are always of LOCAL scope and cannot be declared explicitly. The form of a special label is a dollar sign "$", followed immediately by any valid decimal number, and it can be used in the same manner as a regular statement label identifier, except that its scope is always limited to the procedure in which it is defined.

Example:

```
GLOBAL
  STEP3   LABEL

   process10  PROCEDURE      !Procedure has GLOBAL scope!
      LOCAL
        a,b,c  BYTE
        STEP2  LABEL
      ENTRY
         STEP1: ...          !STEP1 has INTERNAL scope!
           .
           .
           .
         STEP2: ...          !STEP2 is LOCAL to
           .                  "process10"!
           .
           .
         STEP3: ...          !STEP3 has GLOBAL scope!
           .
           .
           .
         $1: ...             !$1 is LOCAL to "process10"!
           .
           .
           .
END process10
```

5.3.6 SIZEOF Operator

Z8000 PLZ/ASM includes a special unary operator, SIZEOF, which operates on type identifiers to determine the size (in bytes) of a variable field. Although SIZEOF can be used with any type, it is useful for sizing arrays, records, and user-defined types.

```
TYPE
  char    BYTE
  digit   char
  matrix  ARRAY [10 10 WORD]
  patient RECORD [height weight BYTE
                  room WORD]
    .
    .
    .

  LD  R0, #SIZEOF digit                 !R0 contains 1!
  LD  R1, #SIZEOF matrix                !R1 contains 200!
  LD  R1, #(SIZEOF matrix/
            SIZEOF WORD)                !R1 contains 100!
  LD  R2, #SIZEOF patient               !R2 contains 4!
  LD  R3, #SIZEOF patient.weight        !R3 contains 1!
```

Appendix A
Instruction Set Summary

TO BE PUBLISHED

Appendix B
High-Level Statement Summary

The following list summarizes the high-level keywords and their uses.

ARRAY	One of the two structured variable types. Used in type definition and variable declaration statements.
BYTE	One of the simple variable types. Used in type definition and variable declaration statements.
CASE	Used in IF conditional-execution statement. Instruction(s) following CASE definition are executed if one of the specified values matches selector register.
CONSTANT	Introduces constant definition(s).
DO	Introduces DO loop.
ELSE	Used in IF conditional-execution statement. Statements between ELSE and FI are executed if the specified condition is false.
END	Module or procedure terminator.
ENTRY	Marks beginning of action-statement part of a procedure.
EXIT	Loop control statement used to control execution flow of a DO loop.
EXTERNAL	Specifies that variables and/or procedures defined as GLOBAL in another module will be used in the current module.
FI	IF statement terminator.
FROM	Used in conjunction with EXIT and REPEAT loop control statements.

GLOBAL	Declares variables and/or procedures to have a scope of the entire executable program.
IF	Introduces IF statement. Code following IF-THEN is executed if the specified condition is true.
INTEGER	One of the simple variable types. Equivalent to WORD.
INTERNAL	Declares variables and/or procedures to have a scope of the current module only.
LABEL	Used to declare statement label scope explicitly.
LOCAL	Declares variables to have a scope of the current procedure only.
LONG	One of the simple variable types. Used in type definition and variable declaration statements.
LONG_INTEGER	One of the simple variable types. Equivalent to LONG.
MODULE	Introduces a module.
OD	DO loop terminator.
PROCEDURE	Introduces a procedure.
RECORD	One of the two structured variable types. Used in type definition and variable declaration statements.
REPEAT	Loop control statement used to control execution flow of a DO loop.
SHORT_INTEGER	One of the simple variable types. Equivalent to BYTE.
THEN	Used in IF conditional-execution block. The statements between THEN and ELSE (or THEN and FI if ELSE is omitted) are executed if the specified condition is true.

TYPE	Introduces type definition(s).
WORD	One of the simple variable types. Used in type definition and variable declaration statements.

The remainder of this appendix contains the complete grammar for Z8000 PLZ/ASM. In this grammar:

- Keywords are shown as all uppercase letters;
- Required special characters are enclosed in 'single quotes';
- Optional items are enclosed in [square brackets];
- Possible repetition of an item is shown by appending a "+" (to signify one or more repetitions) or "*" (to signify zero or more repetitions) to the item;
- Parentheses are used to group items to be repeated;
- A vertical bar "|" signifies that an alternative follows.

PLZ/ASM GRAMMAR - MODULE SYNTAX

```
module                  => module_identifier MODULE
                               declarations*
                           END module_identifier

declarations            => constants
                        => types
                        => globals
                        => internals
                        => externals

constants               => CONSTANT
                              constant_definition*

types                   => TYPE
                              type_definition*

globals                 => GLOBAL
                              var_or_proc_declaration*

internals               => INTERNAL
                              var_or_proc_declaration*

externals               => EXTERNAL
                              restricted_var_or_proc_declaration*

constant_definition     => constant_identifier
                              ':=' expression

expression              => arithmetic_expression
                              [rel_op arithmetic_expression]*

arithmetic_expression   => term [add_op term]*

term                    => factor [mult_op factor]*

factor                  => unary_operator factor
                        => '(' expression ')'
                        => SIZEOF type_identifier
                        => constant_identifier
                        => label
                        => variable
                        => number
                        => character_constant
```

```
character_constant      => character_sequence

type_definition         => type_identifier type

type                    => simple_type
                        => structured_type

simple_type             => BYTE
                        => WORD
                        => LONG
                        => SHORT_INTEGER
                        => INTEGER
                        => LONG_INTEGER
                        => simple_type_identifier

structured_type         => array_type
                        => record_type

array_type              => ARRAY '[' expression+ type ']'
                        => array_type_identifier

record_type             => RECORD '[' field_declaration+ ']'
                        => record_type_identifier

field_declaration       => field_identifier+ type

var_or_proc_declaration
                        => variable_declaration
                        => procedure_declaration

variable_declaration    => variable_noinitial_declaration
                        => variable_initial_declaration
                        => label_declaration

restricted_var_or_proc_declaration
                        => identifier+ type
                        => procedure_identifier PROCEDURE

variable_noinitial_declaration
                        => identifier+ type
```

```
variable_initial_declaration
                        => identifier simple_type
                           ':=' initial_value
                        => identifier identifier+ simple_type
                           ':=' '[' initial_value*
                           ['...'] ']'
                        => identifier structured_type
                           ':=' constructor
                        => identifier identifier+ structured_type
                           ':=' '[' constructor*
                           ['...'] ']'
                        => identifier ARRAY
                           '[' '*' simple_type ']'
                           ':=' '[' initial_value+ ']'
                        => identifier ARRAY
                           '[' '*' simple_type ']'
                           ':=' character_sequence+

constructor             => '[' initial_component*
                           ['...'] ']'

initial_component       => initial_value
                        => constructor

initial_value           => expression
                        => '?'

label_declaration       => label_identifier+ LABEL

variable                => identifier
                        => array_variable
                        => record_variable

array_variable          => array_designator
                           '[' expression+ ']'

array_designator        => array_identifier
                        => record_variable
                        => array_variable

record_variable         => record_designator
                           '.' field_identifier

record_designator       => record_identifier
                        => array_variable
                        => record_variable
```

```
procedure_declaration  => procedure_identifier
                          PROCEDURE
                             locals*
                          [ENTRY
                             statement*]
                          END procedure_identifier

locals                 => LOCAL
                             variable_declaration*

statement              => [label ':'] statement
                       => loop_statement
                       => exit_statement
                       => repeat_statement
                       => if_statement
                       => select_statement
                       => assembler_instruction

loop_statement         => DO
                                statement*
                          OD

label                  => label_identifier
                       => '$' decimal_constant

exit_statement         => EXIT [FROM label]

repeat_statement       => REPEAT [FROM label]

if_statement           => IF condition_code THEN statement*
                               [ELSE statement*] FI

select_statement       => IF selector_register
                               select_element+
                               [ELSE statement*] FI

selector_register      => register

select_element         => CASE expression+
                               THEN statement*

rel_op                 => '=' | '<>' | '<' | '>' | '<=' | '>='

add_op                 => '+' | '-' | LOR | LXOR

mult_op                => '*' | '/' | LAND | MOD | SHL | SHR

unary_operator         => '+' | '-' | LNOT

assembler_instruction  => operation operand*

operation              => Z8000_instruction
```

```
operand                 => register
                        => indirect_register
                        => immediate
                        => indexed_address
                        => based_address
                        => based_indexed_address
                        => direct_address
                        => relative_address
                        => condition_code
                        => flags
                        => int

register                => single_register
                        => double_register
                        => low_byte_register
                        => high_byte_register
                        => quad_register
                        => special_register

single_register         => R0 | R1 | R2 | R3 | R4 | R5 |
                           R6 | R7 | R8 | R9 | R10 | R11 |
                           R12 | R13 | R14 | R15

double_register         => RR0 | RR2 | RR4 | RR6 |
                           RR8 | RR10 | RR12 | RR14

low_byte_register       => RL0 | RL1 | RL2 | RL3 |
                           RL4 | RL5 | RL6 | RL7

high_byte_register      => RH0 | RH1 | RH2 | RH3 |
                           RH4 | RH5 | RH6 | RH7

quad_register           => RQ0 | RQ4 | RQ8 | RQ12

indirect_register       => '@' register

immediate               => '#' expression
                        => '#' SEG expression
                        -> '#' OFFSET expression

indexed_address         => address_designator
                           '(' register ')'

based_address           => register '(' immediate ')'

based_indexed_address   => register '(' register ')'

direct_address          => address_designator

relative_address        => address_designator
```

```
address_designator          => '|' seg_address_designator '|'
                            => seg_address_designator

seg_address_designator      => expression
                            => '<<' expression '>>' expression

condition_code              => Z | NZ | EQ | NE | MI | PL |
                               C | NC | OV | NOV | LT | GE |
                               LE | GT | ULT | UGE | ULE | UGT |
                               PE | PO

special_register            => FCW | FLAGS | NSP | NSPOFF |
                               NSPSEG | PSAP | PSAPOFF |
                               PSAPSEG | REFRESH

flags                       => C | Z | S | P | V

int                         => VI | NVI
```

```
Z8000_instruction     => ADC | ADCB | ADD | ADDB | ADDL |
                         AND | ANDB | BIT | BITB | CALL |
                         CALR | CLR | CLRB | COM | COMB |
                         COMFLG | CP | CPB | CPL | CPD |
                         CPDB | CPDR | CPDRB | CPI | CPIB |
                         CPIR | CPIRB | CPSD | CPSDB | CPSDR |
                         CPSDRB | CPSI | CPSIB | CPSIR |
                         CPSIRB | DAB | DBJNZ | DEC | DECB |
                         DI | DIV | DIVL | DJNZ | EI | EX |
                         EXB | EXTS | EXTSB | EXTSL | HALT |
                         IN | INB | INC | INCB | IND | INDB |
                         INDR | INDRB | INI | INIB | INIR |
                         INIRB | IRET | JP | JR | LD | LDA |
                         LDAR | LDB | LDCTL | LDCTLB |
                         LDD | LDDB | LDDR | LDDRB | LDI |
                         LDIB | LDIR | LDIRB | LDK | LDL |
                         LDM | LDPS | LDR | LDRB | LDRL |
                         MBIT | MREQ | MRES | MSET | MULT |
                         MULTL | NEG | NEGB | NOP | OR |
                         ORB | OTDR | OTDRB | OTIR | OTIRB |
                         OUT | OUTB | OUTD | OUTDB | OUTI |
                         OUTIB | POP | POPL | PUSH | PUSHL |
                         RES | RESB | RESFLG | RET | RL |
                         RLB | RLC | RLCB | RLDB | RR | RRB |
                         RRC | RRCB | RRDB |SBC | SBCB | SC |
                         SDA | SDAB | SDAL | SDL | SDLB |
                         SDLL | SET | SETB | SETFLG | SIN |
                         SINB | SIND | SINDB | SINDR | SINDRB |
                         SINI | SINIB | SINIR | SINIRB | SLA |
                         SLAB | SLAL | SLL | SLLB | SLLL |
                         SOTDR | SOTDRB | SOTIR | SOTIRB |
                         SOUT | SOUTB | SOUTD | SOUTDB |
                         SOUTI | SOUTIB | SRA | SRAB | SRAL |
                         SRL | SRLB | SRLL | SUB | SUBB |
                         SUBL | SWAP | TCC | TCCB | TEST |
                         TESTB | TESTL | TRDB | TRDRB | TRIB |
                         TRIRB | TRTDB | TRTDRB | TRTIB |
                         TRTIRB | TSET | TSETB | XOR | XORB
```

PLZ/ASM GRAMMAR - LEXICAL SYNTAX

```
PLZ_text                => separator* [id_constant_text]
                             (separator+ id_constant_text)*

id_constant_text        => identifier
                        => constant

separator               => delimiter_text
                        => special_symbol

identifier              => letter (letter  |  digit  |  '_')*

constant                => number
                        => character_sequence

number                  => decimal_constant
                        => hex_constant
                        => octal_constant
                        => binary_constant

decimal_constant        => digit+

hex_constant            => '%' hex_digit+

octal_constant          => '%(8)' oct_digit+

binary_constant         => '%(2)' bin_digit+

character_sequence      => ''' string_text+ '''

string_text             => string_char
                        => special_string_text

string_char             => any_character_except_%_or_'

special_string_text     => '%' special_string_char
                        => '%' hex_digit hex_digit

special_string_char     => 'R' | 'L' | 'T' | 'P' | 'Q' | '%' |
                           'r' | 'l' | 't' | 'p' | 'q'

letter                  => 'A' | 'B' | ... | 'Z' |
                           'a' | 'b' | ... | 'z'

bin_digit               => '0' | '1'

oct_digit               => '0' | '1' | '2' | '3' |
                           '4' | '5' | '6' | '7'
```

```
digit                   => '0' | '1' | '2' | '3' | '4' |
                           '5' | '6' | '7' | '8' | '9'

hex_digit               => '0' | '1' | '2' | '3' | '4' |
                           '5' | '6' | '7' | '8' | '9' |
                           'A' | 'B' | 'C' | 'D' | 'E' | 'F' |
                           'a' | 'b' | 'c' | 'd' | 'e' | 'f'

special_symbol          => '+' | '-' | '*' | '/' | '...' |
                           '?' | ':' | '.' | '&' | '#' |
                           '[' | ']' | '(' | ')' | '<>' |
                           '=' | '<' | '>' | '<=' | '>=' |
                           '<<' | '>>' | '|'

delimiter_text          => delimiter
                        => comment

comment                 => '!' any_character_except_! '!'

delimiter               => ';' | space | ',' |
                           tab | formfeed | linefeed |
                           carriage_return
```

Appendix C
Assembler Directives and Extended Instructions

C.1 ASSEMBLER DIRECTIVES

The following is a summary of some of the assembler directives used to control the operation of the Z8000 assembler. Other directives are explained in detail in the Z8000 Assembler User's Guide.

These directives can be embedded in the source program, and always start with a dollar sign "$", immediately followed by the particular directive and then any operands. For example, a programmer may want to fix the address for a procedure:

```
$ABS 12
```

The assembler directives are:

$ABS [location]	Specifies that the object code produced is to be absolute. If "location" is specified, the location counter will be set to that value and the assembler will begin assigning absolute addresses from that location. "Location" must be a constant expression. $ABS remains in effect until a $REL directive is encountered.
$REL [location]	Specifies that the object code produced is to be relocatable. If "location" is specified, it is a relocatable offset from the beginning of the current section. "Location" must be a constant expression. $REL remains in effect until $ABS directive is encountered. $REL 0 is the default at the start of module assembly.
$SDEFAULT	Cancels the effect of the $SECTION assembler directive and restores default memory assignment. Data declarations reside in the data section and procedure declarations reside in the program section.

$SECTION identifier	Causes the object code produced to be associated with a symbolic identifier which can be used later for mapping into one of the memory areas. Remains in effect until either another $SECTION directive is encountered, or until a $SDEFAULT directive restores default memory assignment.

C.2 EXTENDED INSTRUCTIONS

The following three "extended" instructions may be placed within a procedure as a one-operand assembly language instruction and produce an arbitrary byte, word, or long word value.

BVAL constant_expression	Defines a byte value to be located at the current location counter. Can be used to "create" Z8000 instructions or data.
WVAL constant_expression	Defines a word value to be located at the current location counter. Can be used to "create" Z8000 instructions or data.
LVAL constant_expression	Defines a long word value to be located at the current location counter. Can be used to "create" Z8000 instructions or data.

Appendix D
Reserved Words and Special Characters

D.1 RESERVED WORDS

Certain special symbols are reserved for Z8000 PLZ/ASM and can not be redefined as symbols by the programmer. These are the names of operators, condition codes, register symbols, assembly language instructions, and high-level statement keywords. The specific reserved words are listed below.

NAMES OF OPERATORS

LAND	OFFSET
LNOT	SEG
LOR	SHL
LXOR	SHR
MOD	SIZEOF

STATUS FLAGS AND CONTROL BITS

C	V
NVI	VI
P	Z
S	

CONDITION CODES

C	LE	NE	PE	UGT
EQ	LT	NOV	PL	ULE
GE	MI	NZ	PO	ULT
GT	NC	OV	UGE	Z

CONTROL REGISTER SYMBOLS

FCW	PSAP
FLAGS	PSAPOFF
NSP	PSAPSEG
NSPOFF	REFRESH
NSPSEG	

ASSEMBLY LANGUAGE INSTRUCTIONS

ADC	CPSD	IND	LDM	POPL	SETB	SOUTIB
ADCB	CPSDB	INDB	LDPS	PUSH	SETFLG	SRA
ADD	CRSDR	INDR	LDR	PUSHL	SIN	SRAB
ADDB	CPSDRB	INDRB	LDRB	RES	SINB	SRAL
ADDL	CPSI	INI	LDRL	RESB	SIND	SRL
AND	CPSIB	INIB	MBIT	RESFLG	SINDB	SRLB
ANDB	CPSIR	INIR	MREQ	RET	SINDR	SRLL
BIT	CPSIRB	INIRB	MRES	RL	SINDRB	SUB
BITB	DAB	IRET	MSET	RLB	SINI	SUBB
CALL	DBJNZ	JP	MULT	RLC	SINIB	SUBL
CALR	DEC	JR	MULTL	RLCB	SINIR	TCC
CLR	DECB	LD	NEG	RLDB	SINIRB	TCCB
CLRB	DI	LDA	NEGB	RR	SLA	TEST
COM	DIV	LDAR	NOP	RRB	SLAB	TESTB
COMB	DIVL	LDB	OR	RRC	SLAL	TESTL
COMFLG	DJNZ	LDCTL	ORB	RRCB	SLL	TRDB
CP	EI	LDCTLB	OTDR	RRDB	SLLB	TRDRB
CPB	EX	LDD	OTDRB	SBC	SLLL	TRIB
CPL	EXB	LDDB	OTIR	SBCB	SOTDR	TRIRB
CPD	EXTS	LDDR	OTIRB	SC	SOTDRB	TRTDB
CPDB	EXTSB	LDDRB	OUT	SDA	SOTIR	TRTDRB
CPDR	EXTSL	LDI	OUTB	SDAB	SOTIRB	TRTIB
CPDRB	HALT	LDIB	OUTD	SDAL	SOUT	TRTIRB
CPI	IN	LDIR	OUTDB	SDL	SOUTB	TSET
CPIB	INB	LDIRB	OUTI	SDLB	SOUTD	TSETB
CPIR	INC	LDK	OUTIB	SDLL	SOUTDB	XOR
CPIRB	INCB	LDL	POP	SET	SOUTI	XORB

When defining symbols, users should also avoid the forms Rn, RHn, RLn, RRn, and RQn where n is a number from 0 to 15.

EXTENDED INSTRUCTIONS

BVAL
LVAL
WVAL

HIGH-LEVEL STATEMENT KEYWORDS

ARRAY	END	GLOBAL	LONG	REPEAT
BYTE	ENTRY	IF	LONG_INTEGER	SHORT_INTEGER
CASE	EXIT	INTEGER	MODULE	THEN
CONSTANT	EXTERNAL	INTERNAL	OD	TYPE
DO	FI	LABEL	PROCEDURE	WORD
ELSE	FROM	LOCAL	RECORD	

D.2 SPECIAL CHARACTERS

The list of special characters below includes delimiters and special symbols. The difference between them is that delimiters have no semantic significance (for example, two PL2/ASM tokens can have any number of blanks separating them), whereas special symbols do have semantic meaning (for example, # is used to indicate an immediate value).

The class of delimiters includes the space (blank), tab, form feed, line feed, carriage return, semicolon (;), and comma (,). The comment construct enclosed in exclamation points (!) is also considered a delimiter.

The special symbols and their uses are as follows:

+	Binary addition; unary plus.
-	Binary subtraction; unary minus.
*	Unsigned multiplication; dimension specifier for list (one-dimensional array) initialization.
/	Unsigned division.
:	Label terminator.
:=	Constant and variable initialization.
%	Nondecimal number base specifier; special character specifier within quoted character sequence.
#	Immediate data specifier.
@	Indirect address specifier.
$	Current contents of location counter; specifies special LOCAL statement labels; precedes assembler directives.
[]	Enclose components of ARRAY or RECORD definition; enclose index part of ARRAY reference; enclose initialization values.
()	Enclose expressions selectively; enclose octal or binary number base indicator; enclose index part of indexed, based, and based indexed address.
.	Separates RECORD name from field name in RECORD field reference.

<	"Less than" operator
>	"Greater than" operator.
=	"Equal" operator.
<>	"Not equal" operator.
<=	"Less than or equal" operator.
>=	"Greater than or equal" operator.
?	Placeholder in initialization lists.
...	Repetition symbol in initialization lists.
<< >>	Denotes segmented address.
| |	Enclose short offset segmented address.

Appendix E
Hex-ASCII Table

The following table lists the ASCII character set and the hexadecimal representation for each character code.

GRAPHIC OR CONTROL	HEX CODE	GRAPHIC OR CONTROL	HEX CODE
NUL	00	(	28
SOH	01	)	29
STX	02	*	2A
ETX	03	+	2B
EOT	04	'	2C
ENQ	05	-	2D
ACK	06	.	2E
BEL	07	/	2F
BS	08	0	30
HT	09	1	31
LF	0A	2	32
VT	0B	3	33
FF	0C	4	34
CR	0D	5	35
SO	0E	6	36
SI	0F	7	37
DLE	10	8	38
DC1	11	9	39
DC2	12	:	3A
DC3	13	;	3B
DC4	14	<	3C
NAK	15	=	3D
SYN	16	>	3E
ETB	17	?	3F
CAN	18	@	40
EM	19	A	41
SUB	1A	B	42
ESC	1B	C	43
FS	1C	D	44
GS	1D	E	45
RS	1E	F	46
US	1F	G	47
(space)	20	H	48
!	21	I	49
"	22	J	4A
#	23	K	4B
$	24	L	4C
%	25	M	4D
&	26	N	4E
'	27	O	4F

GRAPHIC OR CONTROL	HEX CODE	GRAPHIC OR CONTROL	HEX CODE
P	50	h	68
Q	51	i	69
R	52	j	6A
S	53	k	6B
T	54	l	6C
U	55	m	6D
V	56	n	6E
W	57	o	6F
X	58	p	70
Y	59	q	71
Z	5A	r	72
[	5B	s	73
\	5C	t	74
]	5D	u	75
↑	5E	v	76
	5F	w	77
←	60	x	78
a	61	y	79
b	62	z	7A
c	63	{	7B
d	64	|	7C
e	65	} (ALT MODE)	7D
f	66	~	7E
g	67	DEL (RUB OUT)	7F

DOCUMENT CHANGE NOTICE

DATE: 07-02-79

DCN NUMBER: E3-3055-01

PUBLICATION NUMBER: 03-3055-01

TITLE: Z8000 PLZ/ASM Assembly Language
Programming Manual

PREVIOUS DCN'S BY NUMBER: None

EFFECTIVE DATE: 07-02-79

This Document Change Notice provides changed pages for the publication specified above. The changed pages will remain in effect for subsequent releases unless specifically amended by another DCN or superseded by a publication revision. Replace the following pages in your manual with the enclosed pages:

vii	4-1
	4-7
3-19	
3-35	5-15
3-55	5-17
3-59	5-19
3-77	
3-123	B-3
3-127	B-7
3-129	B-9
3-131	
3-133	

Changes are indicated by a vertical line in the right-hand margin.

NOTE: Please file this DCN at the back of the manual to provide a record of changes.